Generator-Based Design of Custom Systems-on-Chip for Numerical Data Analysis

by

Alon Amid

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Engineering – Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Summer 2021

Generator-Based Design of Custom Systems-on-Chip for Numerical Data Analysis

Abstract

Generator-Based Design of Custom Systems-on-Chip for Numerical Data Analysis

by

Alon Amid

Doctor of Philosophy in Engineering – Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Co-chair

Professor Krste Asanović, Co-chair

With the end of Dennard scaling and the subsequent demise of Moore's Law, the continuous demand for higher computing performance and efficiency is increasingly met through specialization of digital processors. In particular, numerical data processing and machine-learning applications incur high computational costs but often have common computational structures, acting as prime targets for hardware customization. Specialization of digital designs is accompanied by substantial non-recurring engineering (NRE) costs, which limit the proliferation of customized designs. This work presents tools and methodologies for the development of custom systems-on-chip (SoCs) for numerical data analysis applications. An integrated generator-based framework for SoC development is demonstrated through SoC customization and hardware/software co-design for numerical data analysis and machine-learning applications. The development of full-system support from hardware accelerators through system software leads to the identification of several co-design opportunities for increasing accelerator utility in custom SoCs. Specifically, we demonstrate the development of high-performance custom software library implementations to support accelerated numerical data analysis on custom SoCs developed using the Chipyard integrated generator-based framework for custom SoC design. We further identify a need to provide support for processing of a high variety of matrix shapes and sizes in SoC deep learning accelerator matrix engines for accelerated processing of numerical data analysis workloads, and demonstrate up to a 1.25× improvement in the utilization of a matrix engine on small and rectangular matrices through hardware-managed static scheduling, dynamic scheduling, and hardware-managed commutative micro-threading.

Contents

List of Figures

List of Tables

Acknowledgments

There are so many people to thank and acknowledge as part of the experience of a PhD. It is often said that a PhD is an individual endeavour, but I believe that in the modern research age it is rare to find any type of work that can succeed without teamwork and collaboration.

I have been fortunate to work with world-class faculty at Berkeley; professors who challenged my way of thinking, but also introduced me to new ideas and perspectives on problem-solving and system trade-offs. I would especially like to thank my supervisors Bora Nikolić and Krste Asanović, as well as Yakun Sophia Shao, Ion Stoica, Kurt Keutzer, Jim Demmel, Randy Katz, and Jonathan Ragan-Kelly who have helped, collaborated, and advised on multiple projects and ideas. I would also like to thank Prof. Bin Yu for serving on the dissertation committee, and for her contributions to my understanding of applied statistics and data science.

As a person with interests spanning across the EE/CS divide, I was lucky to be affiliated with several research centers and labs within the Berkeley EECS department: ASPIRE, ADEPT, and BWRC. I would like to thank the staff who helped ensure that research operations run smoothly, and addressed any issue or question I had with incredible speed and diligence. I could not have completed the work in this dissertation without Tami, Ria, Roxanna, Kosta, and the incredible Candy Corpus. I would also like to thank the research staff who help maintain and expand the research infrastructure that was used in this dissertation - Jim Lawson, Chick Markley and James Dunn. When speaking of people who hold the EECS department together, it would be hard to imagine a Berkeley EECS PhD experience without Shirley Salanio - the guardian angel of EECS graduate students.

I would like to give special thanks to my main partners in crime and leaders of the Chipyard and FireSim projects, Sagar Karandikar, David Biancolin, Abraham Gonzalez and Jerry Zhao. I have learned so much about the computer architecture research community, open-source project management, and software development from them, as well as about what it takes to successfully engage in academic projects as a team, promote open-source projects, and support stable software releases and development cycles. These are invaluable skills that no paper or course could teach me. I would like to give another special thanks to my go-to technical guru, Albert Ou, who would always answer even my most trivial questions at any hour of the day (or night). I've learned countless debugging techniques from Albert, and I was always amazed by his breadth of knowledge across the entire spectrum of the computing stack. I'd also like to thank Hasan Genc, the leader of the Gemmini project, for countless hours of discussions on the phone, and for navigating the project through often challenging paths.

I thank my past and present colleagues, collaborators, and most importantly, friends, in the BAR, ADEPT, ASPIRE, and BWRC labs: Colin Schmidt, John Wright, Albert Magyar, Adam Izraelevitz, Donggyu Kim, Howard Mao, Martin Maas, Nathan Pemberton, Albert Ou, Jenny Huang, Sam Steffl, Seah Kim, Ameer Haj-Ali, Vignesh Iyer, Pranav Prakash, Harrison Liew, Daniel Grubb, Sean Huang, Ben Keller, Pi-Feng Chiu, Andrew Waterman, Jack Koenig, Chris Celio, Eric Love, Arya Reais-Parsi, Dan Fritchman, Dayeol Lee, Dima Nikiforov, James Dunn, Chick Markley, Jim Lawson, Lisa Wu Wills, David Kohlbrenner, Shweta Shinde, Paul Rigge, Keertana Settaluri, and Stevo Bailey. The people of BAR have been more than just colleagues, but friends who I've had the fortune of traveling with across California, in the mountains and on the beaches, in basketball arenas and airplane boneyards, and on countless Zoom lunches throughout the COVID-19 pandemic.

Throughout my time at Berkeley, I was fortunate enough to be an active member of several organizations, including the Electrical Engineering Graduate Student Association (EEGSA), and its CSGSA counterpart. While I initially joined simply as a favor to my friend David Fridovich-Keil to help with some finance and paperwork as a treasurer, I ended up finding an opportunity to make tangible changes impacting the EECS graduate student community and graduate student experience. I was fortunate enough to work with, and learn from, change-makers and friends such as Sylvia Herbert, Josh Sanz, Kelly Fernandez, Justin Yim, Avi Pandey, Sidney Buchbinder, Gabe Fierro, Kristina Monakhova, Kevin Laeufer, Eldon Schoop, Jake Sporrer, Laura Brink, Mike Danielczuk, Saavan Patel, Hani Gomez, Matt and Carolyn Matl, Mauricio Bustamante, Ryan Kaveh, and so many countless others who volunteer their time in order to make Berkeley EECS a better place for current and future graduate students, and make sure that EECS graduate students have an experience that goes beyond just research work.

I also found a welcoming home within the graduate student community of Berkeley Hillel. Whether it was weekly Shabbat dinners, holiday parties, or retreats to the mountains, I always felt the backing and support of the Jewish community and enjoyed contributing back to it. I would like to thank the graduate student community who welcomed me during my first year, including David, Zach, Zoe, Andy, Todd, Mark, Shante, Rosie, Dan, Sara, and Lauren, as well as the Berkeley Hillel staff, particularly Rabbi Adam, Lauren, Molly, Emily, and Danielle.

Looking back at the beginning of this experience, I probably would not have ended up in Berkeley if my friend Eyal Cidon did not convince me to apply for a PhD program in the US. Together with Jenya Pergament, having the support of two high school friends pursuing PhDs in EE and CS in the Bay Area was invaluable, and created an environment where I was always able to get a taste of home when I needed it. While we ended up pursuing our PhDs in rival universities across the bay, I could not have asked for a better high school class (Hebrew Reali School of Haifa class of 2007). Equally as important are my undergraduate professors at Technion who led me to this path, and kept in communication throughout my time in Berkeley, especially Avinoam Kolodny and Ran Ginosar. I'm also grateful to Ruth and Prof. Bob Brayton, who graciously hosted me when I first arrived in Berkeley. Having a place to land when I arrived relieved a lot of stress of the transatlantic relocation, and I will forever be grateful to them for hosting me during that time.

Finally, I would like to thank my family back home. My parents, Gil and Yael, my "fun car" sisters Gal and Maayan, and my newest family member Michael, who were always available for a phone call, video chat, or "half-way" meeting somewhere in Europe. I've had the privilege of growing up in a house with both a love for computers from a very young age,

as well as a sympathetic ear regarding any personal or professional concerns. I always know my family has my back with any choice I make or any problem I have, and that knowledge and feeling can overcome any distance.

This work was supported in part by the Defense Advanced Research Projects Agency through the Circuit Realization at Faster Timescales Program under Grant HR0011-16-C0052 and the Real Time Machine Learning Program under contract FA8650-20-2-7006; in part by the Advanced Research Projects Agency-Energy, U.S. Department of Energy, under Award DEAR0000849; in part by the NSF CCRI Award 2016662; in part by the NSF CISE SHF Award 1955450; and in part by ASPIRE and ADEPT Lab industrial sponsors and affiliates. The views and opinions of the author expressed herein do not necessarily state or reflect those of the U.S. Government or any agency thereof.

Chapter 1

Introduction

Throughout the past decade, the computing industry has been slowly coming to terms with the reality and challenges of the end of Moore's law. This ending of Moore's law is due to a mix of technological factors, such as the limits of planar lithography-based atomic-scale devices and increased power density resulting from the end of Dennard scaling, as well as economic factors, such as the cost of sub-micron process technologies. Nevertheless, increasingly efficient computing remains a necessity, with the recent big data and machine learning revolutions acting as prime drivers for high-performance computing. Different communities within the computing world have been pursuing multiple research and development efforts in order to continue and meet the increasing computing requirements of the growing software industry. These can generally be classified into three main categories:

1. Scale-Out: Distribute the workload across many machines.
2. Device and Packaging Physics: New devices, process technologies, packaging technologies, or computing physics such as carbon nanotubes, quantum computing, phase-change memory, memristors, etc.
3. Specialization: Specialized hardware for particular applications with limited use-cases.

Each of these approaches provides different capabilities that enable increased computing performance and efficiency, but which also add constraints and assumptions regarding algorithms, software, and fabrication, that limit their usage scenarios when compared with traditional general-purpose computing.

Scale-out computing derives both its power and its limits from the properties of parallelism. Workloads which expose sufficient parallelism theoretically enable unlimited scaling, but at the same time are still limited by Amdahl's law. This ability to scale horizontally benefits from programming models which can generalize to arbitrary parallel programs, flexible application-dependent scaling, and mature software infrastructure platforms such as MPI, MapReduce, Spark, Ray, and DASK [243, 72, 284, 193, 219]. However, this same ability to scale can also incur a large power and energy cost, as synchronization and other overheads of parallel execution can result in less-than-optimal utilization of the underlying computation resources and large amounts of data movement.

Device and Packaging physics provides new properties and capabilities such a non-volatility, low-power operation, sustainability, and tighter device integration across multiple dimensions. Nevertheless, there are still many open questions in terms of cost, yield, reliability, endurance, size, determinism, and other properties of such devices. Furthermore, such devices will require new design infrastructure, which will take time to mature, as well as new software abstractions that will be able to utilize these new device properties. While advances in device physics provides a long-term prospect for more efficient computing, and packaging innovations have the potential to enable better modularity and scalability, resolving the associated open questions will take time.

Hardware specialization provides the benefits of higher energy efficiency and performance improvements beyond simple parallelism [257], but is limited by programming models, developer productivity and challenging software stacks. Hardware specialization can be performed by using current technologies and tools, and provides immediate impact for a select number of applications. However, it requires an equally expensive investment in software support for each unique instance of hardware. Some examples of specialized hardware include digital signal processors (DSPs), artificial-intelligence (AI) accelerators, field-programmable gate arrays (FPGAs), and graphics processing units (GPUs).

The future of computing will require a combination of these different approaches: scale-out systems with specialized and heterogeneous hardware across different nodes and new device physics supporting custom hardware for lower power based on the specialized properties of each node. In this dissertation, we focus on the hardware-specialization aspects of computing, with a particular interest in machine-learning and data-analysis applications as target workloads and driving applications.

1.1 Hardware Specialization

While application-specific integrated circuits (ASICs) have existed since the early days of the microelectronics industry, with common applications in signal processing and communication, specialized hardware has taken the computing industry by storm in recent years. These include mobile application processors, graphics accelerators for gaming and movie production, video transcoders, and AI accelerators. In contrast to traditional ASICs, these specialized processors often involve complex software ecosystems, requiring various levels of programmability and integration with larger and more complex systems. These specialized processors require significant design and fabrication resources, but in return they demonstrate orders-of-magnitude better performance and energy efficiency compared to the general-purpose processor alternative.

In the academic community, specialized accelerators are often designed from scratch and tailored for a particular application or algorithm to demonstrate maximal efficiency of performance through tailored data paths and control paths. This approach is often taken in industry as well, although subsequent generations of the accelerator may build upon the first generation, allowing for some amortization of first-generation design costs.

Nevertheless, the development costs of this approach to specialization are extensive. The design of specialized hardware is a cumbersome process with immense non-recurring engineering (NRE) costs. The drivers of these NRE costs are diverse: they include architecture and detailed design, prototyping, verification, validation, physical design, IP qualification, software development, and much more. A contemporary custom chip can cost well over a hundred million dollars in NRE, and this means that only very high-volume chips are likely to amortize and justify their NRE.

Recent trends in open specifications and open-source software and hardware are helping reduce some of these costs. Free and open instruction set architectures (ISAs) such as RISC-V serve as a substrate for both hardware and software to build upon while providing the freedom for extension and specialization. Similarly, open-source hardware and software efforts help distribute the burden of development and maintenance of common building blocks which do not provide competitive or innovative advantages. This allows for the resources invested in specialization and customization to be re-directed towards innovation rather than boilerplate development.

The state-of-the-art in hardware customization and specialization has primarily focused on methods and tools for hardware description and synthesis. For example, high-level synthesis (HLS) for both ASICs and FPGAs [59, 240] is a tool-based method to rapidly generate hardware descriptions from software programs. However, HLS tools still require non-negligible manipulation of the high-level software description in order to be able to generate synthesizable hardware. HLS is often a good fit for dataflow-centric hardware with limited control-flow, limiting it to only a subset of the specialized hardware spectrum, leaving open the challenge of full-system customization. Alternative high-level hardware-description frameworks such as Bluespec System Verilog [197] and Chisel [23] attempt to lower the cost of hardware customization by providing high-level hardware-description primitives and functionality while maintaining the RTL abstraction (in contrast to HLS tools, which use software abstractions). While these tools provide platforms for increased productivity in digital hardware design, their benefits remain at the hardware-description productivity level, while the majority of hardware customization costs are still incurred throughout the multiple design flows and system integration stages of hardware customization. To improve productivity at the hardware-system-level, generator-based approaches to processor and digital system design have been proposed as an additional layer on top of high-level hardware-description frameworks [198, 22, 170, 234, 126]. In these generator-based approaches, highly parameterized and modular implementations of digital designs using high-level programming language abstractions enable generation of a broad range of RTL designs. Generators describe digital designs at the RTL level in conjunction with additional functional programming and metaprogramming primitives to encode a high level of parameterization and modularity.

The advancements in hardware description, generation, and synthesis tools help increase hardware design productivity towards the desired goal of lowering the customization cost of complete systems-on-a-chip (SoC). The Oxford English Dictionary defines the word "customization" as *modify (something) to suit a particular individual or task.* As such, one key perspective in the customization of SoCs is to not start from scratch, but rather re-use components from existing SoCs. As a result, SoC design frameworks have been proposed as platforms for custom SoC design, intended to compose jointly with the aforementioned advancements in hardware developments tools [43, 26, 212]. Efficient re-use of hardware generators in combination with ad-hoc custom hardware description can significantly improve the productivity of custom digital hardware system design. In this work, we demonstrate SoC customization through composition of generators in order to lower the NRE of SoC development. This composition and customization does not end with the hardware description or IP blocks, but continues all along the SoC development flow, including verification, implementation, and software development.

1.2 Numerical Data Analysis

Numerical analysis predates the invention of modern computers by several centuries. Nevertheless, since the creation of modern electronic processors and micro-processors, numerical analysis has been a prime consumer of computing technologies and has affected many key developments in the computing industry, including the standardization of floating-point computation and parallel computing platforms.

Numerical computing has been at the core of many scientific modeling and data analysis applications from molecular and atomic dynamics to atmospheric and celestial mechanics. Computation for analysis at these scales requires immense computing capabilities which were the driving applications for the early supercomputers of the 20th century. However, high-performance numerical computing for data-driven analysis, which was once the realm of only scientists and mathematicians, has now become a necessity for a large number of commercial applications and household tasks. New professions in data science and machine-learning research are consuming ever-growing computation resources with the goal of developing data-driven models for insights and prediction. A recent refrain, highlighted in a 2017 article in *The Economist*, identifies data as "the new oil" [206]. The "big data" revolution enables new classes of applications and technologies, many of them centered around machine learning and data analysis.

Throughout the years, many techniques have been developed to extract information from large amounts of numerical data, from early classical statistics to modern deep learning. Optimization, linear modeling, dimensionality reduction, feature extraction, and neural networks require computation over large amounts of numerical data in order to obtain meaningful results. Through heavy reliance on linear algebra, these techniques have been able to utilize the large body of work on high-performance numerical computing using efficient linear-algebra algorithms to enable new tools and data-driven applications. Their high reliance on vector, matrix, and tensor data representations and linear-algebra computations, also make them amenable to hardware customization.

As such, state-of-the-art numerical computing software often originates from the requirements of scientific computing and has utilized data-parallel and vectorized computation across platforms ranging from vector supercomputers [125, 223], through packed-SIMD ISA extensions such as Intel's SSE and AVX and ARM's NEON and SVE in general purpose processors [179, 248, 191], and most recently highly parallel programmable GPUs [55, 56]. Recently, the high computational demands of matrix computations in deep-learning applications have led to the addition of specialized matrix and tensor hardware units in processor architectures designed to support high-performance numerical computing [140, 56], as well as discrete tensor accelerators dedicated only to deep learning and the processing of large amounts of data [149]. Writing software for these specialized architectures requires careful attention to data parallelization opportunities, since general-purpose programming languages and tools have yet to achieve efficient automated mapping and compilation to processor architectures beyond traditional scalar processors. State-of-the-art software libraries used for numerical data analysis often manually map common linear algebra operations to specialized architectures in order to achieve high performance and efficiency.

As general-purpose computing increasingly exhibits diminishing performance and efficiency returns with the end of Moore's law, numerical computing and numerical data analysis continue to be a driving force for high-performance computing, with the range of target platforms extending from supercomputers to mobile devices. In this work, we focus on hardware that can be used for modern deep-learning applications as well as traditional statistics, and the customization required for it to be applicable to the broad field of numerical data analysis.

1.3 Dissertation Scope and Outline

This work presents tools and methodologies for the development of custom systems-on-chip (SoCs) for numerical data analysis applications. An integrated generator-based framework for SoC development is demonstrated through SoC customization and hardware/software co-design for numerical data analysis and machine learning applications. The development of full system support from hardware accelerators through system software leads to the identification of several co-design opportunities for increasing accelerator utility in custom SoCs.

Specifically, this works builds upon the aforementioned high-productivity hardware description tools and generator-based design approaches but focuses on the system-level aspects of SoC design, which include system configuration and component integration, design-flow integrity, system-level design space exploration, and system-level HW/SW co-design. We demonstrate how re-use and integration of several key generator-based tools together with a unified configuration abstraction provide new capabilities in SoC design, from pre-silicon inception and exploration stages to complete implementation and verification flows. We demonstrate how this approach differentiates generator-based SoC design from alternative SoC design approaches through multiple levels of configurability, and a single source of design truth across the entire hardware and software development flow.

The driving applications throughout the dissertation are numerical data analysis workloads, from deep learning through classical statistics. While the primary focus of recent state-of-the-art hardware specialization has been deep learning, this work highlights a broader class of applications which are based on linear algebra computations for numerical data analysis. This broader class of applications can be supported by custom hardware for vector and matrix computations. As part of the SoC customization process, we highlight generator-based specialized hardware for matrix and vector computations, and its integration as part of a custom SoC. We demonstrate the complete design and customization process, from hardware through software, and highlight the challenges and opportunities enabled through a rapid generator-based customization process for this class of numerical data analysis driving applications. In particular, we identify several co-design opportunities for improved support for small matrix computations, which can be enabled or disabled by the hardware generator during the SoC development process based on the requirements of the custom SoC target application class.

Chapter 2 provides background on numerical data analysis applications, their computational properties and motifs, and techniques used for high-performance numerical analysis.

Chapter 3 provides background on hardware architectures used to accelerate such computational patterns found in numerical data analysis workloads and introduces the Hwacha vector unit and Gemmini matrix engine which are used to accelerate the aforementioned computational patterns within the context of this dissertation.

Chapter 4 introduces ideas and methodologies pertaining to generator-based SoC design, challenges based on previous experiences applying those ideas in practice, and methodological approaches which help address those challenges and encode them using relevant tools and design flows.

Chapter 5 presents the Chipyard integrated SoC-development framework, which is the primary vehicle used to implement the generator-based SoC design methodology, through automation and integrated tooling. This chapter describes the main components and tools of the Chipyard framework, the ways it can be applied to custom SoC design, and its impact on accessibility of education related to custom generator-based SoC design.

Chapter 6 introduces several hardware-software co-design capabilities enabled by generator-based tools within the Chipyard framework. Specifically, it demonstrates full-system design space exploration enabled by generators, debugging and optimization enabled by generator-based FPGA-accelerated simulation, and performance tuning through templated integration of automatically generated software header files.

Chapter 7 presents the customization and mapping of the numerical data analysis software stack onto SoCs designs and hardware accelerators generated by the Chipyard framework, with a particular focus on mapping the data science application stack onto the Gemmini matrix engine.

Chapter 8 presents matrix engine hardware-software co-design opportunities for numerical data analysis applications, identified and evaluated through the mapping of the numerical computing software stack onto Chipyard-generated SoC.

Chapter 9 concludes the work by summarizing the key contributions of this dissertation and suggesting opportunities for future research directions.

Chapter 2

Algorithms for Numerical Data Analysis

With the extensive availability of digital data, data science has emerged as a discipline linking together knowledge from multiple industries and academic fields in order to harness data to its fullest extent. To this end, data scientists are combining traditional statistical modeling methods with modern deep learning methods to identify new insights from data analysis and construct efficient prediction, inference and training models.

2.1 Machine Learning and Numerical Data Analysis

Data analysis enabled by modern computing capabilities has propelled the data revolution into prominence across a multitude of application domains. From anomaly detection and business intelligence (BI) extracted through immense data logs, through biological and genomics research, recommendation systems, computer vision, and natural language processing enabled through advanced modeling methods, the ability to efficiently process large numerical data sets is impacting every aspect of our lives. As the size and diversity of data sets continue to grow, the science around their processing is evolving as well.

Data science spans a wide range of data processing tasks, from data acquisition and cleaning through exploratory analysis to modeling, prediction and inference. The latter parts are often aggregated and referred to as *Machine Learning* (ML). ML is a study of data-driven algorithm development, and is often considered to be a branch of both modern statistics and computer science. Machine-learning algorithms improve and evolve through the collection and refinement of data. The discipline has grown and expanded from traditional statistical methods through data-mining and unsupervised methods to modern deep neural networks. In particular, *numerical data*, in contrast to other data forms such as strings and objects, is amenable to processing founded on the basis of mathematical and statistical algorithms. Such numerical data can be collected through sensors and simulations or be used to model other real-world objects such as images and colors. In this work, we are interested in both exploratory data analysis as well as production data modeling and machine learning pipelines.

Data-modeling pipelines are used for both model construction and training, as well as prediction and inference based on the model. Data-modeling pipelines often require several types of methods, including data preparation, feature extraction, and model construction, all in order to generate insightful data-driven models and conclusions. Each step in these pipelines may require different numerical processing methods. Starting from the source of the data, data-modeling pipelines may compute and validate basic properties of the data such as covariance matrices. Next, numerical data often requires extensive pre- and post- processing to make it amenable for efficient data analysis or training in supervised learning models. Such processing includes data transformation through parsing, permuting and cleaning. An equally important part of processing may include clustering or other dimensionality-reduction techniques to identify important features or reduce the space of classification classes. Dimensionality reduction can also be used to efficiently communicate features and gradients between remote models. The incorporation of a theoretical or formal model within the pipeline may also require solving one or more linear systems in order order to encode additional features. Finally, data passes through the final parts of the modeling pipeline which can incorporate training or inference of various machine-learning models, including linear regressions, support vector machines, deep neural networks, or any other data-driven modeling algorithm. The composition of all these tools and methods often consists of both data movement and numerical transformation of the data, and in particular, linear-algebra-based transformations of the data.

Due to the recent conflation of the terms "machine learning", "deep learning", and "deep neural networks" (DNN), in both scientific and popular literature, throughout this dissertation we will attempt to avoid the term "machine learning". We will use the term "deep learning" or "DNN" to describe machine learning based on deep neural networks, and we will use the term "data analysis" to describe exploratory data analysis as well as what may sometimes be referred to as "classical statistics" or "classical machine learning," which includes dimensionality reduction, clustering, linear models, and so on. Figure 2.1 helps illustrate the differences and overlaps between these categories of domains, with numerical data analysis being a subset of the broader data analysis domain overlapping with machine-learning and deep-learning.

2.2 Data Analysis Algorithms and Dense Linear Algebra

The foundation of numerical data analysis and classical statistics is linear algebra: weighted systems of equations, eigenvalue problems, projections, low-rank approximations, and least-squares regressions, all require extensive numerical computations on tensors, matrices, and vectors. Deep neural networks, which have been the subject of some recent divergence from

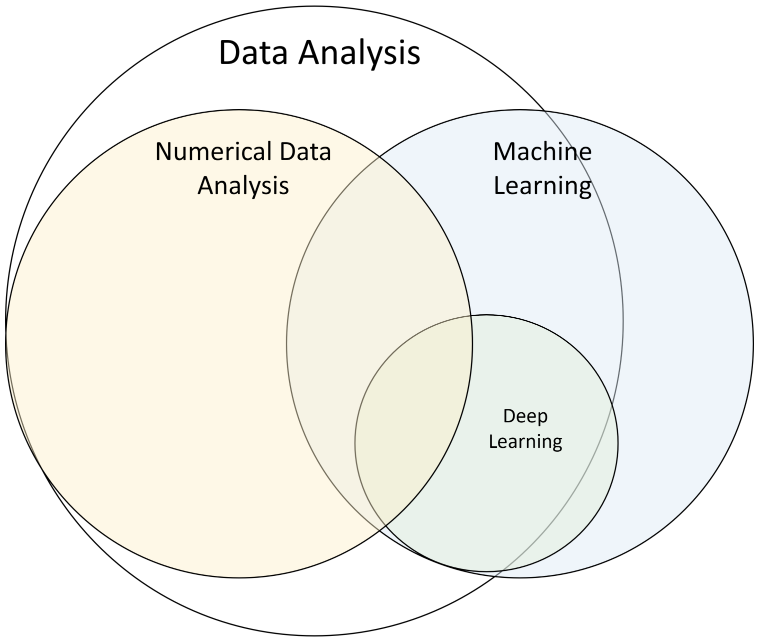


Figure 2.1: Data analysis workload categories and overlaps between them.

classical machine learning and statistical methods for data modeling, are also heavily dependent on dense linear algebra computations, to some extent even more so than classical statistics and data analysis. Numerical data sets are typically represented as vectors, matrices, and tensors. Through the manipulation of the dimensionality and geometry of numerical data, dense linear algebra methods extract abstract features, generalizations and anomalies which form the building blocks of data-driven models.

Several examples of key data analysis methods and their associated linear algebra problems include:

- Principal component analysis (PCA) - Linear dimensionality reduction by projecting data to a lower-dimensional space. Extracting and selecting the principal components of the data helps reduce the dimensionality of a high-dimensionalilty dataset while maintaining as much of the data's variation as possible. Dimensionality reduction is especially useful as part of exploratory data analysis, but can also be used as part of a data-modeling pipeline. PCA is in fact the singular value decomposition (SVD) from linear algebra, used to find the orthogonal projection of the matrix.

- Clustering - Grouping data samples into groups (clusters) through an unsupervised process based on their numerical features. Two popular forms of clustering are K-means and spectral clustering. K-means clustering partitions the data into k clusters where a data sample belongs to a cluster with the nearest mean, which is typically found through a heuristic iterative process involving several matrix multiplications. Spectral clustering constructs a similarity matrix of the samples in the dataset (based on some form of similarity), performs an eigen decomposition on the Laplacian of the similarity matrix, and performs K-means clustering on the first k eigen-vectors.

- Linear regressions - Construction of a linear model of a system based on a dataset of samples and features. Typically consists of solving a linear least-squares problem on an overdetermined matrix (with the potential additional regularization), sometimes with regularization parameters (Ridge regression, Lasso regression). The linear least-squares problems can be solved by directly solving the normal equations ($A^TAx = A^Tb$) or using matrix decompositions such as QR decomposition or SVD.

- Deep neural networks (DNNs) - Artificial neural networks with multiple layers between the input and the output, used as universal function approximators and universal classifiers. Artificial neural networks are biologically inspired computing structures, in which each layer is composed of multiple "neurons" which apply a potentially non-linear transformation on the input. Common layers in DNNs are implemented using matrix-vector multiplications, matrix-matrix multiplications, and convolutions.

- Support vector machines (SVMs) - A supervised learning algorithm used primarily for classification and regression. SVMs project the data into a higher-dimensional space in order to construct hyperplanes which can separate classes of data points with maximal margins. Training a support vector machine requires solving a quadratic problem, in which most optimization methods are based on dense linear algebra and matrix multiplications. Inference using SVMs requires summing over a simple matrix-vector or matrix-matrix multiplication related to a kernel function selected to suit the problem.

- Random projections - techniques for dimensionality reduction based on probabilistic guarantees, where points from a sufficiently high dimensional space are projected to a lower-dimensional space while maintaining the pairwise distances between the points. The projection is performed by multiplying the original data matrix of size $m \times n$ with a random matrix of size $n \times k$, where k is the desired lower dimension. The random matrix is initialized based on a known distribution (for example, Gaussian) and has normalized columns.

While this list in not exhaustive, in this work, we focus workloads that act as part of the data-modeling pipeline, including dimensionality reduction through clustering and PCA, linear modeling, and deep neural networks. Together, these techniques allow data scientists to construct and analyze complex numerical datasets with both explicit and latent properties.

2.3 High-Performance Numerical Data Analysis

The foundations of computer-based numerical data analysis lie in the lessons learned from years of research in high performance computing that have had long lasting implications on the modern data analysis software stack.

While early high-performance numerical computing libraries were designed primarily with re-usability in mind, their portability and taxonomy later enabled performance optimization techniques thanks to their well-defined interfaces and problem structures.

The high-performance computing community has identified that by partitioning linear algebra problems into "blocks" (or "tiles"), one can extract two primary advantages which assist in obtaining high performance from a parallel computer with a memory hierarchy: **parallel processing** both at the inter-block level and the intra-block level, and reduction of **communication** through a surface-to-volume effect for the ratio of operations to data movement [76]. These principles lie at the core of high-performance numerical data analysis in both hardware and software implementations.

Parallel Processing

Computer architecture literature traditionally recognizes three major types of parallelism: *instruction-level* parallelism (ILP), *thread-level* parallelism (TLP, sometimes also called task-level parallelism) and *data-level* parallelism (DLP):

- Instruction-level parallelism represents the ability to overlap the simultaneous execution of multiple instructions within a single instruction stream (or thread).
- Thread-level parallelism represents the ability to overlap the simultaneous execution of multiple independent instruction streams, implying multiple independent program counters.
- Data-level parallelism represents the ability of a single operation or instruction to be performed simultaneously on multiple elements of data (whether in a single instruction stream or multiple instruction streams).

All three types of parallelism can be exploited both through software techniques and hardware techniques, although some may be more prevalent than others. ILP is most commonly exploited in hardware implementations through pipelining, dynamic scheduling and speculation. High-performance software will extract ILP through software pipelining, loop unrolling and Very Long Instruction Word (VLIW) compilation to assist the hardware in overlapping independent instructions. TLP is most often exploited in hardware through multi-processors (sometimes called multi-core processors) and simultaneous multithreading (SMT) processors. Software will extract TLP using threading libraries such as POSIX threads (pthreads) or OpenMP. DLP can be exploited in hardware using single-instruction multiple-data (SIMD) instructions which enable static scheduling and wide datapaths. Software will utilize data-level parallelism through explicit SIMD programming, auto-vectorizing compilers (which transform loops to SIMD instructions), and high-level data-parallel programming languages. ILP, TLP and DLP are not mutually exclusive, and are typically used in tandem in order to maximize processor performance.

Communication

Algorithms generally have two types of costs: arithmetic and communication. *Communication* is the cost of moving data between levels of a memory hierarchy or over an interconnect connecting processing units. The cost of communication can often be reduced to two main components: bandwidth and latency [27]. Communication can be observed in computer architecture across various levels of the computing system: I/O, on-chip networks, off-chip networks, and the memory hierarchy. Communication-avoiding algorithms have been researched since the early 1970s-1980s [155, 89, 147] with the goal of improving performance through reduction in communication costs.

While parallel processing may help reduce the arithmetic cost of an algorithm, it often incurs an additional communications cost. For example, a lower-bound on communication in parallel matrix multiplication for the "memory-scalable" case was found to be $\Omega(n^2/\sqrt{P})$, where P is the number of processors [142]. As can be observed, in a parallel system with P processors, this means that the lower bound on the total amount of communication in the system would be $\Omega(P \times n^2/\sqrt{P}) = \Omega(\sqrt{P} \times n^2)$, which is $\sqrt{P}$ times the lower bound of communication in the single-processor memory-scalable case ($P = 1$).

Similar communication lower bounds pertaining to sparse and dense linear algebra problems have been proven in recent years [27], leading to advances in communication-optimal algorithms that are used across a variety of numerical data analysis problems. These communication-avoiding schemes can be captured both in software implementations and specialized hardware implementations which rely on high degrees of computational parallelism.

2.4 Arithmetic Intensity of Numerical Data Modeling Kernels

From a computational perspective, dense linear algebra is considered to be a compute-intensive class of workloads. A useful measure of whether a workload is likely to be compute-bound or memory-bound is the arithmetic intensity (or operational intensity) of the workload [277]. Arithmetic intensity is a property of a workload or an algorithm (as opposed to a property of the "problem", or a property of the hardware). It measures the ratio between computational (arithmetic) operations to data movement operations. As such, it measures the ratio of operations which require processing capabilities vs. operations which require memory capabilities (or memory bandwidth). The definition of an "operation", as well as the exact units used to measure arithmetic intensity, may change based on the context of the problem. In high-performance scientific computing, arithmetic intensity is often measured in floating-point operations (FLOPs[1]), while data movement is measured in bytes.

[1]Due to frequent confusion, we clarify that the notation "FLOPs" is the plural of FLOP (a **FL**oating-point **OP**eration), and measures operation count, while the notation "FLOP/s" or "FLOPS" is a measure of throughput, denoting floating-point operations per second

As a corollary, since deep-learning inference may use quantized integer computations rather than floating-point computation, the arithmetic intensity measure for deep neural networks uses integer operations (OPs) rather than floating-point operations. Similarly, some deep-learning researchers consider only the data movement of weight parameters rather than data movement of both weight and activation data [149]. In this work, we consider all necessary data movement, including DNN activation data.

Table 2.1 lists the arithmetic intensity properties of several key dense linear-algebra operations used for numerical data analysis. In particular, we focus on matrix operations and matrix decompositions, since numerical datasets are often represented in the form of matrices. We observe that most matrix operations have arithmetic intensity that grows with the size of the input matrix. There is a large body of work on increasing the arithmetic intensity of matrix decomposition algorithms and minimizing communication using blocking/tiling techniques. We further elaborate on these within the context of the LAPACK and BLAS software libraries in Chapter 7.

A particular subset of dense linear algebra and machine learning of recent interest are operations that compose deep neural networks (DNNs). DNNs have been at the forefront of advancements in computer vision, natural language processing (NLP) and additional data-driven modeling and prediction domains. They are made up of a large and diverse class of linear algebra operations ranging from convolutions, through sparse and dense matrix-vector and matrix-matrix operations, and even optimization and gradient computations as part of the training process. Table 2.2 lists several key DNN operators and their arithmetic intensity properties. Typical layers in computer vision DNNs include spatial convolutions, which perform two-dimensional or three-dimensional convolutions over the input tensors ($O_{k,p,q} = \sum_r^R \sum_s^S \sum_c^C X_{c,p+r-R/2,q+s-S/2} W_{k,c,r,s}$); fully connected layers, which multiply an input feature vector by a weight matrix ($o_i = \sum_j W_{ij} x_j + b_i$); and pooling layers, which reduce the dimensions of activation tensors through averaging, maximizing or minimizing across the size of a filter. Typical layers in NLP DNNs include embedding layers, which transform an object from a vocabulary encoded as a one-hot vector into a vector embedded within a high-dimensional space by performing a table-lookup operation; long short-term memory (LSTM) layers, which are recurrent layers that use feedback connections between features across the time dimension ($h_t = NonLinearity(W_{ih} x_t + b_{ih} + W_{hh} h_{(t-1)} + b_{hh})$) together with a set of gates to regulate the flow of information in order to control vanishing gradients; and attention layers, which build upon bi-directional LSTMs and add all-to-all relationships across input feature vectors through additional weight matrices (effectively resulting in a series of matrix multiplications) [266]. Notably, a large number of DNN operators exhibit arithmetic intensity that grows with the size of their parameters (a "surface-to-volume" effect).

Research on deep neural networks has traditionally focused on improving their inference accuracy and generalization ability, often through larger numbers of parameters and increased amount of computation. As such, compositions of operators such as spatial convolutions, fully connected layers, and attention, are continuously increasing the computational cost of deep neural networks. We note that throughout the evolution of deep neural net-

Table 2.1: Building blocks for numerical analysis and their characteristics (LAPACK 3.9.0 implementations, matrix size $m \times n$).

	FLOPs	Memory Footprint	Arithmetic Intensity
Matrix-Vector Multiplication	$2nm$	$nm + 2m$	$\frac{2n}{n+2}$
Matrix Multiplication	$2nmk$	$nm + mk + kn$	$\frac{2nmk}{nm+mk+kn}$
LU Decomposition	$mn^2 - n^3/3$	mn	$\frac{mn^2 - n^3/3}{mn}$
QR Decomposition (Blocked Householder*)	$2mn^2 - 2n^3/3$	$mn + \mathcal{O}(nb)$	$\frac{2mn^2 - 2n^3/3}{mn + \mathcal{O}(nb)}$
Cholesky Decomposition	$n^3/3$	$n^2/2$	$\frac{2}{3}n$
Singular Value Decomposition** [99]	$\Sigma : 4mn^2 - 4n^3/3$ $U, V : kRn + 2Rmn + 4Rn$	$\Sigma : mn + \mathcal{O}(m)$ $U, V : m^2 + n^2 + \mathcal{O}(4n^2)$	$\Sigma : \frac{4mn^2 - 4n^3/3}{mn + \mathcal{O}(m)}$ $U, V : \frac{kRn + 2Rmn + 4Rn}{m^2 + n^2 + \mathcal{O}(4n^2)}$
Linear System Solve	$\frac{2}{3}n^3 + 2n^2$	$n^2 + 3n$	$\frac{\frac{2}{3}n^3 + 2n^2}{n^2 + 3n}$
Linear Least Squares*	$2mn^2 - 2n^3/3 + 2mn + n^2$	$mn + nb + m$	$\frac{2mn^2 - 2n^3/3 + 2mn + n^2}{mn + nb + m}$

*The parameter b represents the block size in a blocked algorithm. We assume $m > n$.
**SVD is solved using iterative algorithms. Therefore, the number of FLOPs and memory footprint can only be estimated. We estimate the number of FLOPs based on the cost of various operations within the SVD listed in Golub and Van Luan [99], assuming k iterations, $m >> n$, and computing R singular vectors.

works, a subset of networks have focused on improving efficiency as opposed to accuracy of DNNs [280, 130]. This class of computationally efficient DNN models, which are designed to run within latency and energy constrained environments, is based on a set of computationally efficient operators with smaller parameter sets or higher arithmetic intensity. These operators include depthwise convolution, pointwise convolution, and group convolution, each of which reduces or partitions one of the dimensions of the traditional spatial convolution. The pointwise convolution is a traditional spatial convolution with a filter size of 1×1. As such, it has smaller parameter memory footprint, but also lower arithmetic intensity. A spatially separable convolution factorizes a square $(R \times R)$ convolutional filter into two vectors $(1 \times R$ and $R \times 1)$, hence reducing the parameter memory footprint and arithmetic operation count by a factor of the filter size (R). The group convolution partitions the input channels and output channels to G groups, removing dependencies between the groups. This reduces the number of compute operations and parameter size by a factor of G, but also reduces data re-use (due to the elimination of data-reuse across groups). A particular case of group convolution, in which the number of groups is equal to the number of input channels is called “depthwise convolution”. This operation, which was popularized in DNNs models designed for small memory footprints (e.g. MobileNet [127]), has very low arithmetic intensity since it effectively maximizes the reduction in data re-use in group convolutions by maximising the number of groups, which may sometimes lead to a performance degradation despite the decrease in memory footprint and arithmetic operation count. We include several such efficient operators in Table 2.2 to demonstrate the impact of operator and layer selection on the arithmetic intensity of DNN inference.

As is the case with matrix decompositions and dense linear algebra for traditional data analysis and scientific computing, we observe that the arithmetic intensity of DNN operators typically grows with the sizes of the operator parameters (number of channels and filters, and sizes of filter kernels and activation tensor dimensions). Notably, some DNN operators exhibit higher arithmetic intensity than others despite similar parameter sizes. Such observations have been used in the past to devise DNN models that execute more efficiently on various processor architectures [97]. Within the context of hardware for data-science, it is generally preferable to have operators with higher arithmetic intensity, since compute resources are typically “cheaper” to add compared to providing additional memory bandwidth. At the same time, it has also been found that preferences for high arithmetic intensity vs. smaller memory footprints are highly dependent on the type of hardware architecture, with certain high arithmetic intensity models running faster on compute accelerators while resulting in slower performance on traditional CPUs [107]. Despite this diversity of operations, the vast majority of data analysis and DNN operations are based on highly data-parallel linear-algebra computations, which exhibit relatively high levels of arithmetic intensity, ranging on the orders of tens to hundreds of operations per byte. For example, the ResNet-50 DNN model [118] has an arithmetic intensity of approximately 100 operations per byte (depending on the precision of computation) [201, 271].

2.5 Computational Motifs for Accelerated Computing

With the popularization of parallel and distributed computing across the computing spectrum, several attempts have been made at identifying and characterising fundamental computational kernels with the goal of containing and simplifying their parallelization [21, 67, 58]. Often described as computational motifs such as dwarfs or giants, these motifs (which include dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, map-reduce, and others) represent common computational kernels that can be found across a set of applications from various diverse domains such as machine learning, databases, graphics, and embedded computing.

The analysis of these motifs has led to the understanding that certain hardware design patterns indeed fit a broad set of compute kernels. Vector architectures and SIMD units

Table 2.2: Building blocks for DNNs and their characteristics

	Parameter size	OPs	Memory Footprint	Arithmetic Intensity
Spatial Convolution	$CKRS$	$2CKRSPQ$	$PQ(C+K)+RSCK$	$\frac{2CKRSPQ}{PQ(C+K)+RSCK}$
Fully Connected	$CHWKPQ$	$2CHWKPQ$	$CHW+CHWKPQ$	$\frac{2CHWKPQ}{CHW+CHWKPQ}$
Pooling	0	CHW	CHW	1
Embedding	VE	L	$LE+VE$	$\frac{L}{LE+VE}$
Attention	$4E^2$	$8LE^2+4L^2E$	$4E^2++5LE+L^2A$	$\frac{8LE^2+4L^2E}{4E^2+5LE+L^2A}$
LSTM	$16E^2+10E$	$8E^2$	$8E^2+16E$	$\frac{16E^2+10E}{8E^2+16E}$
Factorized Spatial Convolution	CKR	$2CKRPQ$	$PQ(C+K)+RCK$	$\frac{2CKRPQ}{PQ(C+K)+RCK}$
Pointwise Convolution ($D_K=1$)	CK	$2CKPQ$	$PQ(C+K)+CK$	$\frac{2CKPQ}{PQ(C+K)+CK}$
Group Convolution	$CKRS/G$	$2CKRSPQ/G$	$PQ(C+K)+RSCK/G$	$\frac{2CKRSPQ}{GPQ(C+K)+RSCK}$
Depthwise Convolution ($C=K=G$)	CRS	$2CPQRS$	$2CPQ+CRS$	$\frac{2CPQRS}{2CPQ+CRS}$
Channel Shuffle	0	0	$2CPQ$	0
Shift Convolution	CK	$2CKPQ$	$PQ(C+K)+CK$	$\frac{CKRPQ}{PQ(C+K)+RPQ}$
Channel Aggregation	$(\sum_i C_i)KRS$	$2(\sum_i C_i)KRSPQ$	$PQ(\sum_i C_i+K)+(\sum_i C_i)KRS$	$\frac{2(\sum_i C_i)KRSPQ}{PQ(\sum_i C_i+K)+(\sum_i C_i)KRS}$

*We use the dimension terminology used by NVIDIA to denote the tensor and kernel dimensions: the number of input channels C, the input height H, the input width W, the number of output channels K, the output height P, the output width Q, the filter height R, the filter width S. The embedding, LSTM, and attention layers use a different classification of parameters, in which we assume L represents the number of objects being embedded (sequence length), the parameter V represents the size of the embedding vocabulary, and the parameter E represents the embedding dimension (or input/hidden dimension in internal layers). The attention layer can also use an additional parameter to represent the number of attention heads, listed as A.

were found to be very efficient for the dense linear algebra motif, while multi-core architectures provide additional load-balancing required for the sparse linear algebra motif. These hardware architecture patterns and the associated software libraries which are able to utilize them efficiently have been integrated into modern high-performance processors and their respective software stacks.

A quote attributed to Prof. Katherine Yelick and Prof. James Demmel notes that "hardware accelerators are a method of turning a compute-bound problem into a memory-bound problem". More precisely, hardware *compute accelerators* are indeed a method of turning compute-bound problems into memory-bound problems (when sufficient computational resources are provided and utilized in the accelerator), and the majority of hardware accelerators presently available in commercial products are indeed *compute* accelerators. Compute accelerators accelerate workloads through the addition of parallel and specialized computational units for high-intensity math and bit sequence manipulation. In contrast, *memory-system acceleration* often requires speculative or prior knowledge of non-standard or data-dependent organization of data in memory. This often materializes through memory prefetching, scheduling, custom caching, and bandwidth allocation, in order to accelerate memory-bound problems. Since compute accelerators often do not require such prior or speculative knowledge of *non-standard* data organization in memory, compute accelerators are sometimes considered to be more "general" and "programmable". Domain-specific accelerators such as ray-tracing accelerators and multimedia transcoding accelerators are often a mix of compute acceleration and memory-system acceleration.

As transistor density and on-die resources continues to increase with modern process technologies (albeit, bound by power), all while memory bandwidth remains constrained by pin-counts and surface areas, providing additional on-chip compute resources is often a simpler endeavor than providing additional memory system resources. Hence, from the perspective of accelerating computational motifs, using compute accelerators has the potential to generate high return on investment due to their applicability across multiple application domains. As noted in Section 2.4, compute kernels associated with numerical data analysis algorithms generally exhibit high arithmetic intensity and fall under the category of compute-bound problems within the dense linear-algebra motif. Hence, hardware accelerators for this motif can provide performance and efficiency gains for this broad class of workloads. Further fine tuning and customization of accelerators and systems within this domain for particular subsets of algorithms can provide additional gain, while still maintaining high performance and applicability to a wide class of applications within the motif. In the following chapters, we will introduce how customization of hardware systems and accelerators and be applied to acceleration of numerical data analysis algorithms.

Chapter 3

Hardware for Numerical Data Analysis

With the growing computational requirements of numerical data analysis algorithms and the increasing amounts of data that needs to be processed, dedicated hardware that is able to take advantage of their computational properties has been designed in order to support high-performance processing. This dedicated hardware takes advantage of data-parallelism, as well as common linear algebra computational motifs found in numerical data analysis workloads.

3.1 Data Parallel Architectures

Numerical data analysis is traditionally characterized by computation on large amounts of data. This computation is often repetitive over each data element, exposing a high degree of data-level parallelism. Throughout the history of modern computing, the leading edge of high-performance computing has always included demand for data-level parallelism [20, 122]. Early supercomputers such as the CDC STAR-100 [125], Cray-1 [223], and their successors were based on vector processing instructions. These vector supercomputers extracted performance using SIMD vector processing instructions that were able to orchestrate and feed data at a high rate to the arithmetic units. This ability to feed data to the arithmetic units at a high rate required high memory bandwidth. Notably, there were some differences in memory hierarchies across vector supercomputers - for example, the CDC STAR-100 was a vector memory-memory architecture while the Cray-1 was a vector-register architecture. Vector-register architectures reduced some of the memory bandwidth requirements by enabling intermediate data reuse within vector registers.

In the 1980s-1990s data-parallel vector supercomputers fell in popularity and were mostly replaced by general-purpose CMOS superscalar microprocessors, which had much lower cost due to their much larger market volume. CMOS superscalar microprocessors were able to exploit instruction-level parallelism, enabled through high-density CMOS process technologies and on-die integration. Superscalar microprocessors eliminated the need for explicit vectorization of code, providing high performance and higher flexibility for a broader range of applications. Since data-parallelism can also be represented in the form of instruction-level parallelism, CMOS superscalar microprocessors were deemed strictly superior to vector supercomputers in cost-performance and took over their place in the high-performance computing market.

Nevertheless, instruction-level parallelism is limited by the scope and resources of the instruction windows and prediction mechanisms of the processor. With the end of Dennard scaling [73] and the emergence of the "Power wall" [21] during the 2000-2010s, the energy and power costs of increasingly wider high-frequency superscalar microprocessors had become untenable with diminishing returns on die area, resulting in a return to architectures that exploit parallelism explicitly. Thread-level parallelism (TLP) in the form of multi-core architectures, and data-parallel parallelism in the form of wider SIMD ISA extensions and GPUs, emerged as attractive solutions for continuously increasing computing demands under the constraints of bounded power envelopes.

However, multi-core machines were not a panacea for performance improvement in the post-Dennard-scaling era. Multi-core thread-parallel machines incur a cost for state and instruction management logic for each thread or instruction stream. Together with inter-thread synchronization, coherency, and communication overheads, the costs of thread-level parallelism may at times out-weigh its benefits. Given that performance improvement from explicit parallelism is bounded by Amdahl's law [8], the scaling limits of such explicitly parallel architectures depend on the type of target platform. Server processors with many independent instruction streams are able to utilize multi-core architectures with tens of independent thread-parallel cores, while energy-conscious mobile processors will typically not have more than 2-8 independent cores due to limited thread-level parallelism in mobile workloads.

Data-parallel architectures provide opportunities for additional performance and efficiency gains on a subset class of applications. Data-parallel SIMD and vector instructions provide the processor with semantic guarantees regarding the properties of the larger computations, which enable efficient micro-architectural optimization and **static** scheduling of operations. These main properties are homogeneity of operations across elements, independence of operations across elements, and regular element access patterns within a vector register file and/or memory instructions [20]. The compact nature of data-parallel instructions also benefits instruction caches and reduces instruction-fetching requirements. Their properties enable micro-architectural implementations with higher energy efficiency compared to scalar or superscalar processors. For example, a regular access pattern enables a high-bandwidth vector register file to be constructed of smaller banks with a reduced number of ports. Similarly, since operations are grouped together, a smaller number of control path wires are required, reducing both leakage and switching energy. These execution patterns further lend themselves to energy reduction thanks to a reduction in the number of instruction fetches and TLB accesses.

Data-parallel instructions and extensions have become a staple of modern ISAs, with x86 supporting packed-SIMD instructions through the SSE and AVX series of extensions, Power including the Altivec and VSX extensions, and ARM offering the NEON packed-SIMD and SVE vector extensions [248, 179, 191, 74, 105]. RISC-V implementations have included several non-standard vector extensions [46, 290, 153, 279, 230], with a proposal for a standard vector extension being developed as part of the RISC-V standardization process. Observations on the RISC-V vector extension standardization process can be found in Appendix A.

Of notable mention in discussions of data-parallel architectures is the Graphics Processing Unit (GPU). The GPU is a processor designed for offloading of highly parallel graphics computation. It originates from a different type of workstation than the CPU and was originally not intended to be used for general-purpose parallel computation. In fact, it bears no common lineage with the evolution of data-parallel computation within CPUs [122]. Nevertheless, GPUs are based on many of the same data-parallel architectural patterns that evolved within CPUs and general-purpose computing. GPUs are composed of a collection of multi-threaded SIMD processors. Hence, one can think of a GPU as a multi-threaded chip multi-processor (CMP) in which each processor is in-fact a data-parallel vector processor rather than a traditional scalar processor. As such, GPUs invest their die resources primarily into sustaining high-throughput under the assumption of abundant data-parallelism, as opposed to a more even balance between throughput and latency of operations for general-purpose computation in CPUs.

In recent years, the emergence of machine learning and dense linear algebra as key workloads, together with the availability of on-chip area resources (but limited power budget), has led to the popularization of 2-dimensional (2-D) spatial hardware computation units for operations such as matrix multiplication or convolutions. These 2-D spatial units take advantage of data-parallelism and data-reuse possibilities across dot-product computations to implement more efficient computation patterns. Together with 1-D data-parallel units, these 2-D units are now commonly found in server CPUs [140], GPUs [38, 55, 56], mobile phones, and other custom accelerators and systems [149, 233, 113]. The following sections will describe some of the properties and implementation design spaces for 1-D and 2-D data-parallel units, as well as the specific implementations used in the evaluation in this work.

3.2 1-D Data-Parallel Accelerators

Traditional data-parallel micro-architectural implementations can be generally categorized as temporal, spatial, or a mix of the two. Temporal implementations rely on an assumption of long vector lengths in order to hide memory access latency and amortize a deep execution pipeline for high throughput, while spatial implementations utilize wider datapaths to provide a spatial degree of parallelism which also enables a possibility for further instruction-level parallelism (ILP).

While early supercomputing vector machines such as Cray-1 and CDC STAR-100 relied on narrow temporal micro-architectural implementations, other more modern implementations such as the Fujitsu A64FX rely on spatial implementations (specifically, a 512-bit wide spatial implementation) to implement vector ISA extensions such as the ARM Scalable Vector ISA extension (SVE) [248].

There are many additional nuances to data-parallel implementation (superscalar, register renaming, etc.), but for simplicity, we can think of the following two design points as representing extreme ends of the temporal and spatial implementation philosophies:

1. A temporal implementation based on an in-order pipeline decoupled from the scalar process pipeline, a banked vector register file, and narrow datapath (32-64 bits). We will refer to this design point as *temporal*.

2. A superscalar out-of-order pipeline tightly coupled with the scalar processor, a multi-ported vector register file with register renaming, and a wide datapath (128-512 bits). We will refer to this design point as *spatial*.

Each design point allows for micro-architectural and software optimization using the abstracted dimension, while at the same time limiting similar optimizations across the exposed dimension. For example, temporal vector machines are able to utilize different vector element layouts within their vector register files in order to optimize register file bandwidth and pipeline utilization, while in contrast, spatial machines are constrained to specific memory layouts since the wide datapath must correspond to the respective memory layout. Conversely, temporal machines can rely on deep pipelines or multi-cycle execution which provide high-throughput on long vectors but increase the processing latency of short vectors, while spatial implementations can support low-latency processing of short vector at the cost of a potential utilization penalty. The different implementation approaches also impact the programming model: for example, spatial machines allow for simple re-interpretation of data within vector registers, since they often assume contiguous layout of data within the register. In contrast, temporal machines, which can benefit from data-layout optimizations, may require sufficient abstraction at the software level which can prevent such re-interpretation of data.

3.3 Hwacha Decoupled Vector Accelerator

Hwacha [167, 174, 173, 228] is an open-source decoupled vector accelerator developed at UC Berkeley focused on maximizing energy efficiency while remaining a favorable compiler target. In this section, we describe Hwacha as background to help understand its use within the complete system design process of generator-based SoCs for numerical data analysis, as used in Chapters 5, 6, and 7. Hwacha is a temporal vector micro-architecture, and it has been developed as a RISC-V non-standard extension that integrates with the Rocket Chip generator [22] through the Rocket Co-processor interface (RoCC). Several VLSI implementations have been taped-out in 16 nm, 22 nm, 28 nm, and 45 nm technology nodes at 1 GHz+ clock frequencies [169, 290, 153, 279, 230, 100, 229].

Similarly to traditional vector architectures, Hwacha offers a more flexible programming model than fixed-width SIMD architectures by exposing a *vector length register* to the software. This allows software to query and manipulate the vector length, achieving both portability and scalability by abstracting the actual number of compute units in the microarchitecture. The same code executes correctly on all Hwacha configurations of any hardware vector length. A stripmine loop can handle fringe elements gracefully without extra scalar code.

The user-visible register state consists of vector *data* registers (`vv0-255`), vector *predicate* registers (`vp0-15`), scalar *shared* registers (`vs0-63`), and scalar *address* registers (`va0-31`). The vector data registers are used to store vectorized data that will be processed by the machines. Scalar address registers are meant for scalar values used in address computations such as base addresses and strides, while scalar shared registers are meant for scalar values used as part of arithmetic computations. As such, scalar shared registers can be read and written within a vector-fetch block, while scalar address registers are read-only within vector-fetch blocks, simplifying support for non-speculative light-weight access-execute decoupling.

The vector data registers are the primary vector data storage elements within the Hwacha architecture. The vector register file is treated as a fine-grained configurable resource, enabling software to exchange unneeded registers or numerical precision for longer vectors. The control thread indicates the desired number of architectural registers and their element widths, and the vector unit re-partitions the physical state and maximizes the hardware vector length accordingly.

The vector predicate registers are used for predicated operations in order to provide support for vectorized control flow. The combination of simple vector predication and consensual branch instructions based on a small set of quantifier conditions suffices to efficiently handle complex control flow [171]. The instruction encoding permits all vector operations to be predicated.

Hwacha Programming

Hwacha implements a *vector-fetch* programming model [167]. In this model, vector instructions execute independently of the RISC-V control thread in self-contained *vector-fetch blocks*, launched by the control thread using a custom instruction that conveys a PC to the vector unit. This dissociation simplifies code generation and further aids microarchitectural optimizations. Through this structured division of responsibilities, the control thread should be able to complete the stripmining loop faster and be able to continue doing useful work, while the worker thread is independently executing vector instructions.

Nevertheless, this programming model also comes with challenges. Scalar data must be communicated from the control thread to the vector unit through dedicated data movement instructions or through memory. This communication generates an overhead for every vector-fetch block, which in principle should be negligible compared to the execution time of the vector-fetch block itself. However, this imposes a challenge in automatically generating and optimizing code across multiple vector-fetch blocks. This programming model has many

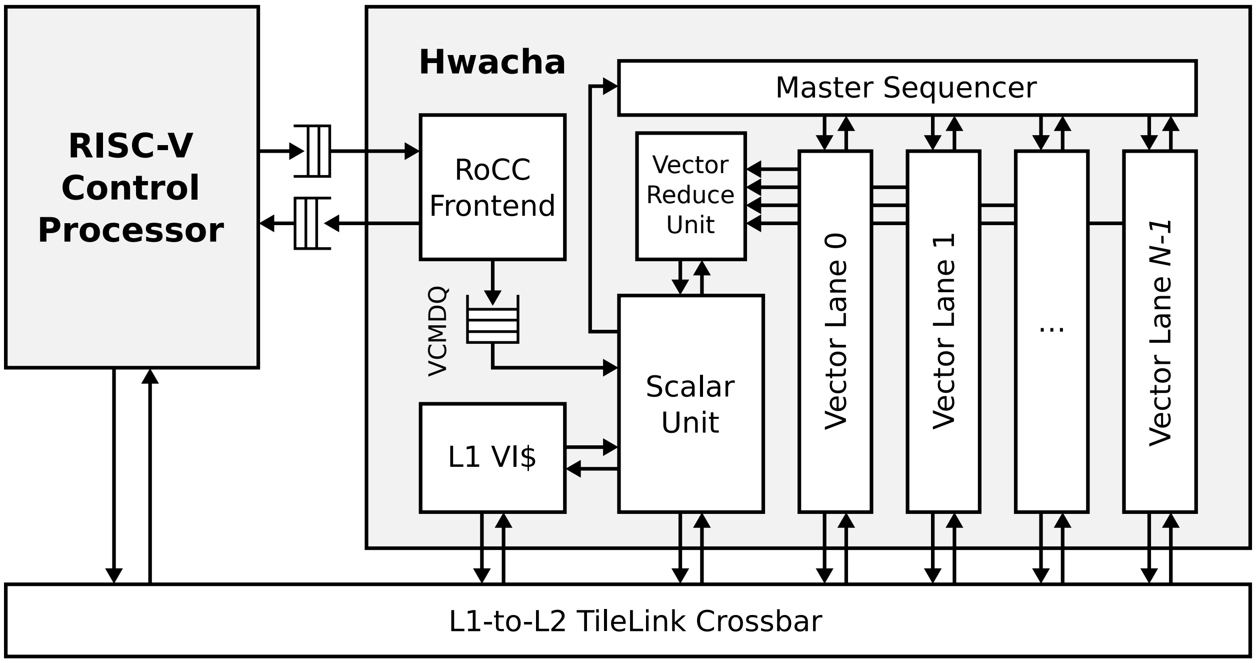


Figure 3.1: Hwacha accelerator instance with N vector lanes.

similarities to the modern relationship between CPUs and GPUs, in which the CPU acts as a control thread for GPU kernel operations. However, they differ in the cost of communication (due to the integrated nature of Hwacha within the shared memory system), as well as in the management of the private memory.

Hwacha Micro-architecture

Hwacha incorporates ideas from access/execute decoupling [242], decoupled vector architectures [85], and cache refill/access decoupling [30], applying them in a cache-coherent memory system. Extensive decoupling enables the microarchitecture to effectively tolerate long and variable memory latencies with an in-order design.

Figure 3.1 presents the high-level structure of Hwacha, implemented as a discrete co-processor with its own independent front-end. This vector-fetch decoupling relieves the control processor to resolve address calculations for upcoming vector-fetch blocks, among other bookkeeping actions, well in advance of the accelerator.

Hwacha consists of one or more replicated vector lanes, each of which is organized around a dense vector register file constructed of four 1R/1W SRAM banks as shown in Figure 3.2. The lanes employ *temporal execution* to efficiently saturate long-latency functional units by pipelining multiple elements of a vector across a number of consecutive cycles. To continuously supply operands at full bandwidth without stalls, the banks follow a systolic schedule that eliminates port conflicts. The Hwacha master sequencer schedules and distributes operations to the vector lanes, while local lane sequencers and expanders decompose instructions into smaller micro-operations and schedule them in a systolic fashion across the lane banks.

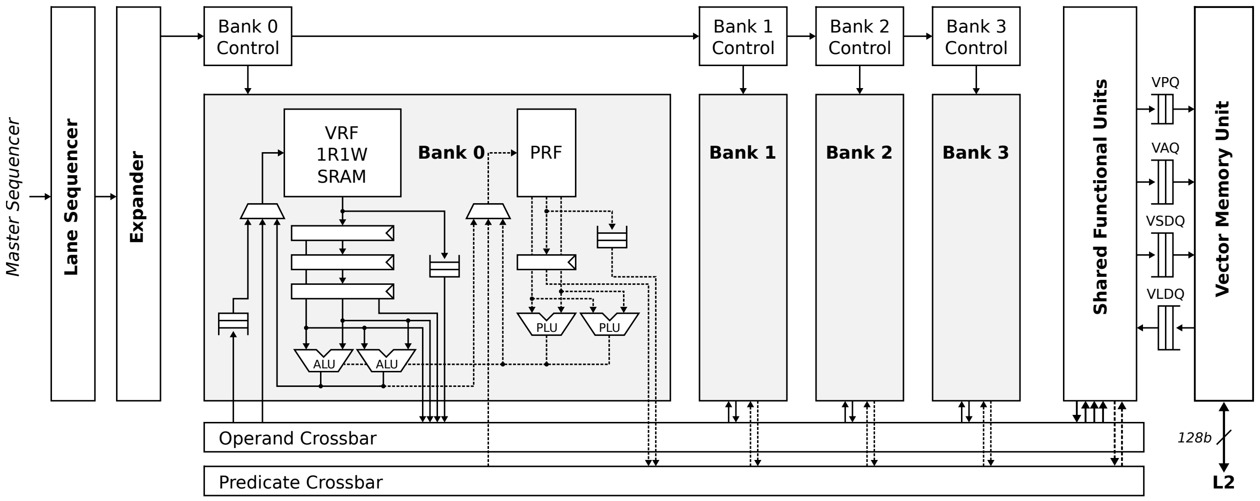


Figure 3.2: **Microarchitecture of one vector lane.** VRF = vector register file, PRF = predicate register file, ALU = arithmetic logic unit, PLU = predicate logic unit, VPQ = vector predicate queue, VAQ = vector address queue, VSDQ = vector store data queue, VLDQ = vector load data queue.

Hwacha is implemented as a generator, such that various micro-architectural parameters such as the number of lanes, vector register files sizes, sequencer slots and functional units can be configured at elaboration time to support a variety of functionality and performance targets. For example, some of the fabricated test-chip implementations of Hwacha were configurations as single-lane configurations [279], while others comprised of dual-lane configurations [230].

3.4 2-D Data-Parallel Accelerators

Matrix and tensor computations are classes of data-parallel computations that can typically exploit parallelism and data re-use across more than a single dimension. Conventional 1-D data-parallel accelerators are able to parallelize a single loop level within a nested-loop structure. In contrast, two-dimensional (2-D) data-parallel accelerators can parallelize and re-use data across two loop levels within a nested-loop structure.

For example, a matrix multiplication requires a three-level nested loop:

```
for (m=0; m < M; m++) {
  for (n=0; n < N; n++) {
    for (k=0; k < K; k++) {
      C[n, m] += A[m, k]* B[k, n];
}}}
```

If we use the `hw_for` notation to indicate a data-parallel loop that is accelerated in hardware (whether using a spatial implementation or a temporal implementation), only the innermost loop of a matrix-multiplication nested-loop implementation can be accelerated using a 1-D data-parallel accelerator:

```
for (m=0; m < M; m++) {
  for (n=0; n < N; n++) {
    hw_for (k=0; k < K; k++) {
      C[n, m] += A[m, k]* B[k, n];
  }}}
```

With 2-D data-parallel accelerators, the **two** innermost levels of the nested-loop structure can be accelerated using the 2-D data-parallel accelerator:

```
for (m=0; m < M; m++) {
  hw_for (n=0; n < N; n++) {
    hw_for (k=0; k < K; k++) {
      C[n,m] += A[m, k]* B[k, n];
  }}}
```

While there has been some recent work regarding temporal 2-D data-parallel acceleration within 1-D vector units [227], the vast majority of 2-D data-parallel acceleration research and development in recent years has focused on spatial architectures. Spatial architectures are a class of accelerator architectures that exploit high computational parallelism using direct communication between an array of relatively simple processing elements (PEs). Compared to SIMD and vector architectures, spatial architectures have relatively low on-chip memory bandwidth per PE, but they have good scalability in terms of routing resources and memory bandwidth. Spatial architectures vary with the complexity of the processing elements and the complexity of the interconnects. Processing element complexity can range from a simple single operation such as a fused multiply-add, to a fully capable processor with instruction decoding and full-fledged ALU. Interconnect complexity can range from static uni-directional single-input single-output interconnect, to complete networks-on-chip with addressing and routing capabilities to each processing element in the spatial array. This variety of spatial design points enabled by the two-dimensional space results in a spectrum of architectural options ranging from highly flexible re-configurable arrays [213, 190] to more targeted spatially unrolled fixed-function implementations.

2-D spatial arrays, and systolic arrays in particular, have been a topic of research in computer architecture for many years [161, 160], but until recently have failed to find commercial adoption in modern multiprocessors. Systolic arrays are one type of spatial architecture in which the routing between the processing elements is statically fixed to orchestrate a sequence of operations across the PEs in a “systolic” fashion. While matrix computations, and matrix multiplication in particular, have been a kernels of importance in dense linear algebra for many years, superscalar microprocessors and vector/SIMD processors have been a design point that sufficiently answers both general-purpose computing needs and occasional high-performance computing needs. Nevertheless, the increasing demands of dense linear-algebra computing in deep learning have tipped the scale, leading to adoption of spatial architectures within mobile SoCs, GPUs, and high-performance discrete accelerators for machine learning. The high-computational cost of dense linear-algebra operations such as matrix multiplication and convolutions within deep learning models (for example, convolutions constitute 90% or more of the computation in DNNs for computer vision), together with the ubiquity of deep-learning solutions across an increasingly widening range of applications such as computer vision, natural-language processing and signal processing, have led to the integration of spatial architectures within the mainstream computing hardware stack.

Systolic arrays encode in hardware the same basic principles used in high-performance computing software: reduction in communication cost while maximizing data-parallelism. By using a large number of processing elements, systolic arrays maximize data-parallelism across a large number of data elements. By communicating data only between adjacent processing elements, systolic arrays reduce the cost of communication since the wire distances are short and the data does not need to travel through a complex memory hierarchy [147, 161, 160]. Thanks to the high degree of parallelism and data reusability of convolutions and matrix-multiplication operations, these systolic or spatial architectures are a natural option for accelerating DNNs.

Spatial architectures have been integrated into silicon-based systems under a variety of system configurations: as integrated components within GPU SIMD processors [204], as discrete PCIe-attached accelerator devices [149, 189], as peripheral accelerators on SoCs [241, 246, 120], and as CPU-integrated units [140, 247, 144]. Each configuration has profound implications on software implementation as well as latency, power and energy constraints. For example, spatial architectures that are integrated within larger compute units (CPU, GPU) can perform finer-grained operations in conjunction with their host unit. In contrast, peripheral accelerators integrated on an SoC may provide energy and power efficiency but incur latency in operating-system overheads and peripheral-device management. Similarly, decoupled accelerators may provide benefits for only very large matrices due to communication and setup costs, while tightly integrated accelerators could lower this threshold.

Spatial Characteristics

Spatial architectures utilize dataflow as a means for reducing communication between processing elements. Dataflow defines an order of computation and order of data movement across a set of processing elements. Typical dataflows for a matrix-multiplication spatial architectures include input-stationary (IS), weight-stationary (WS), and output-stationary (OS). In each dataflow, one of the matrices involved in the operation (where the first matrix operand is referred to as the "input", while the second matrix operand is referred to as "weights") is statically resident in the spatial array, while the other two matrices "flow" through the two dimensions of the spatial array. The Eyeriss project [51] proposed a taxonomy for classifying convolutional neural network (CNN) accelerator dataflows according to the type of data each PE locally reuses. The four dataflows proposed in the taxonomy are weight stationary (WS), output stationary (OS), row stationary (RS), and no local reuse (NLR). In a WS dataflow, each filter weight remains stationary in the PE register file. In an OS dataflow, the accumulation of the output feature stays stationary in the PE, and the partial sums are stored in the PE register file. In a NLR dataflow the PEs do not exploit data re-use at the local register file level, but rather use inter-PE communication for input feature map re-use and partial sum accumulation. In a RS dataflow, each PE operates on one rows of filter weights and one row of the input feature map, and generates one row of partial sums. In the RS dataflow, partial sums from different PEs are then accumulated together to generate the output feature map. Evidence has suggested that different CNN layer types, shapes, and sizes, map with different levels of efficiency to different dataflows in spatial architectures [162, 14], advocating for hybrid dataflow architectures.

The memory hierarchy of a spatial architecture can be further extended beyond the register files within each PE with larger memories shared across several PEs. These larger shared memories can be managed in the form of explicitly addressed scratchpad memories or with large register file abstractions. Spatial architectures typically require higher memory bandwidth than temporal architectures due to their larger number of computational units. The structured nature of spatial architecture enables structured access patterns to these memories shared across several PEs, which can reduce their complexity and enable banked low-ported memory designs.

The interconnect within a spatial architecture determines the level of flexibility, and often the programming model, of a spatial architecture. A static interconnect limits the spatial architecture to fixed operations, fixed operation sizes, and fixed dataflows. On the other extreme, a fully dynamic interconnect enables the architecture to implement a variety of operations and operation sizes, but requires a complex routing and programming scheme. Modern spatial architectures often select a hybrid approach, which may enable limited or hierarchical interconnect reconfigurabiliy to support a diversity of operations, sizes, and dataflows.

3.5 Gemmini Spatial-Array Accelerator

Gemmini [96, 95] is an open-source generator of DNN accelerators developed at UC Berkeley, spanning across different spatial hardware architectures, programming interfaces, and system integration options. In this section, we describe Gemmini as background to help understand its use within the complete system design process of generator-based SoCs for numerical data analysis, as used in Chapters 5, 6, 7, and 8. A high-level system diagram of a Gemmini accelerator is illustrated in Figure 3.3. Gemmini provides a flexible architectural template to support a diverse set of numerical and structural microarchitecture parameters. Several VLSI implementations have been taped-out in 12 nm, 16 nm, and 22 nm technology nodes at 500 MHz-1 GHz clock frequencies. In addition, Gemmini also produces software binaries tuned for each generated hardware instance, either through a push-button programming interface via the ONNX Runtime [214, 64, 90] or a low-level, Gemmini-provided Application

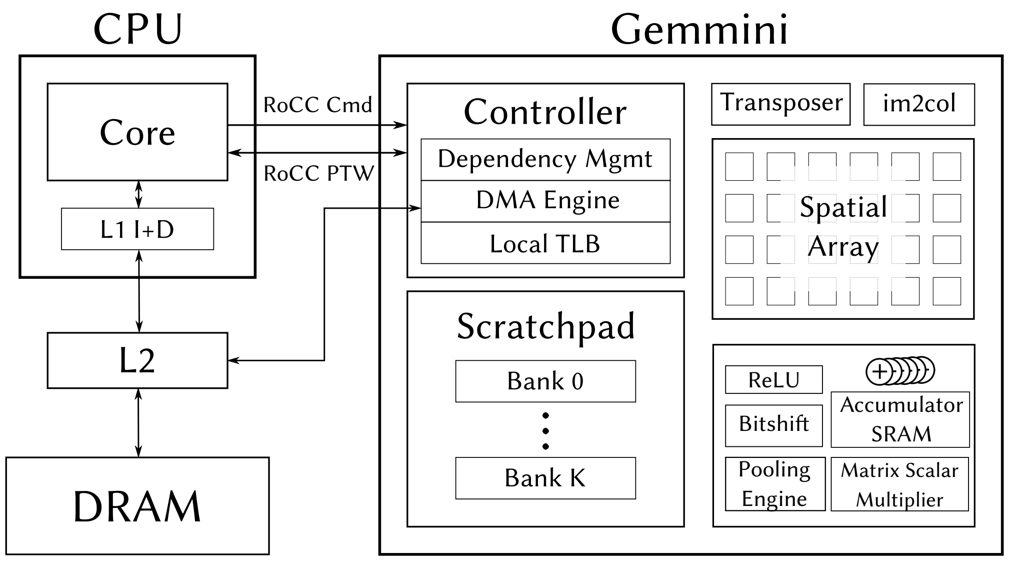


Figure 3.3: Gemmini system diagram. [96] (© 2021 IEEE).

Programming Interface (API) that is portable across Gemmini-generated accelerators.

Gemmini Micro-architecture

The foundation of Gemmini's architectural template is a spatial architecture with spatially distributed processing elements (PEs), each of which performs dot products and accumulations, with different structural parameters (e.g., dataflows or spatial architectures) and numerical parameters (*e.g.*, array size and bitwidth). The spatial array reads data from a local, explicitly managed scratchpad memory, made up of banked SRAMs, and writes results either to that same scratchpad or to local accumulator storage with a higher bitwidth than the input data. In addition, Gemmini also supports other commonly-used DNN kernels, *e.g.*, pooling, non-linear activations (ReLU or ReLU6), and matrix-scalar multiplications, through a set of configurable peripheral circuitry units. Gemmini accelerators can include fixed hardware controllers, implementing finite-state-machines controlling the execution of matrix multiplications or convolution operations. As is the case with the Hwacha vector unit, Gemmini has been developed as a RISC-V non-standard extension that integrates with the Rocket Chip generator through the RoCC interface and connects to an SoC memory system though a shared last level cache, as illustrated in Figure 3.3.

Gemmini's spatial array template is composed of a two-level hierarchy to provide flexible compositions of microarchitecture structures, as demonstrated in Figure 3.4. The spatial array is composed of a rectangle of *tiles*, where tiles are connected via explicit pipelined registers. Each individual tile can be further broken down into a rectangular array of *Processing Elements (PEs)*, where PEs in the same tile are connected combinationally without pipelining registers. Each PE performs a single multiply-accumulate (MAC) operation every cycle,

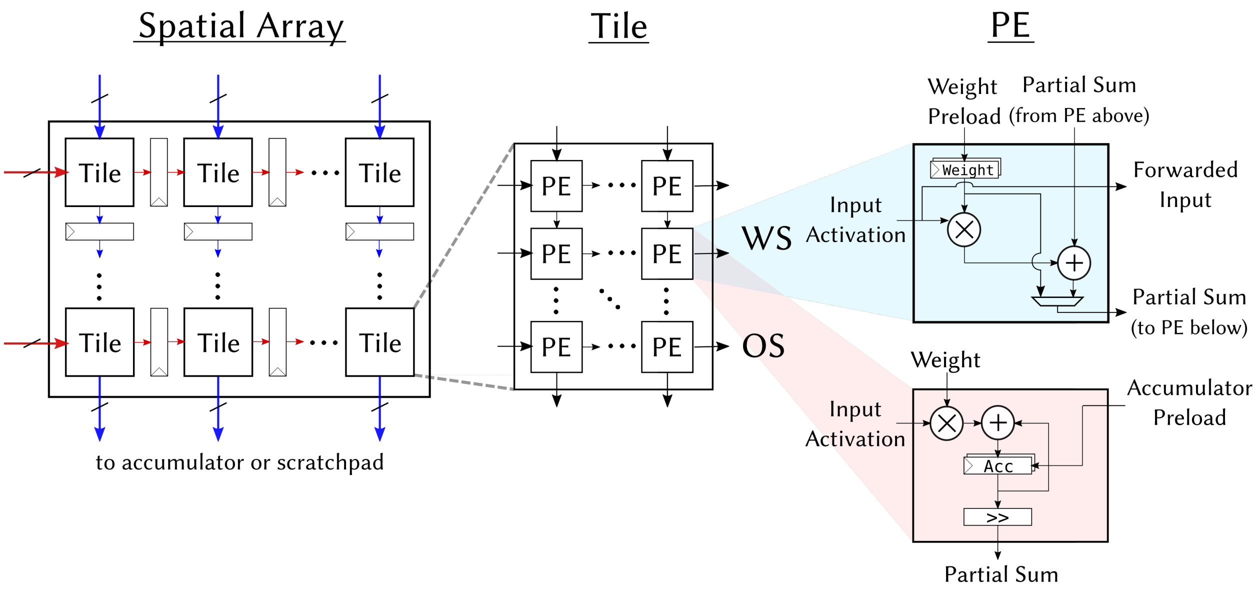


Figure 3.4: Microarchitecture of Gemmini's two-level spatial array template.

using either the weight- or the output-stationary dataflow or both. Every PE and every tile shares inputs and outputs only with its adjacent neighbors. Using this two-level hierarchy, a Gemmini instantiation can implement a systolic array architecture by configuring each tile to include only one PE. Alternatively, it can also implement parallel vector registers with combinational multiply-accumulate reduction trees by configuring fewer tiles with multiple PEs within each tile, similar to accelerators such as NVDLA [241] and DianNao [50, 52]. The default Gemmini configuration uses a systolic-array architecture, with support for a weight-stationary dataflow.

Gemmini supports a large number of parameters, allowing a high degree of customization over the compute characteristics, memory capacity, and SoC-level characteristics of any accelerator that is generated. The most relevant parameters, described in Table 3.1, configure all parts of the compute stack, from the dataflow of the spatial array to the cache hierarchy of the host CPU and the main memory. For example, the datatype of the PEs can be configured to signed integers, unsigned integers, or floats, of any size defined by the user. We highlight several of these configuration parameters:

Datatypes: DNN workloads show a large amount of variety with regards to the datatypes they operate on. For example, inference on the edge is often done with 8-bit fixed-point numbers, while training and inference in the cloud are typically performed using single-precision or low-precision floating point numbers. Recent research is also enabling the use of a variety of new floating-point representations which diverge from the standardized IEEE-754 format [251, 220, 148, 108]. Gemmini allows users to specify arbitrary new datatypes, with user-defined multiply-accumulate (MAC), rounding, and comparison functions, by leveraging the robust typeclass support in Chisel [23]. For traditional types, Gemmini provides signed and unsigned fixed-point numbers, as well as parameterizable floating-point formats

Category	Parameter	Recommended Range
Spatial Array	Mesh Rows	1–256
	Mesh Columns	1–256
	Tile Rows	1–256
	Tile Columns	1–256
	Dataflow	Weight/Output stationary, or both
Accelerator Memory	Scratchpad Capacity	256 bytes–16 MB
	Accumulator Capacity	256 bytes–8 MB
	Scratchpad Banks	1–4
	Accumulator Banks	1–4
	Accumulator Ports	1-2
Execution Schedule	ROB Entries	4–128
	Load Queue Entries	2–128
	Store Queue Entries	2–128
	Execute Queue Entries	4–128
Controller	PE Latency	0–4 cycles
	DMA Bus Width	64–256 bits
	DMA Block Size	32–64 bytes
	TLB Entries	2–64
Datatypes	Datatype	SInt/UInt/Float/User-defined
	Input Bitwidth	8–32 bits
	Output Bitwidth	8–32 bits
	Accumulator Bitwidth	16–64 bits
Operators	Multiply by Scalar	Present/Not + Width
	Transposer	Present/Not
	Pooling	Present/Not
	Im2col	Present/Not

Table 3.1: Example Gemmini generator parameters.

using the Berkeley Hardfloat library [117].

Dataflow: The dataflow of the spatial array can be fixed at elaboration-time or configured to be runtime-adjustable. Gemmini supports both output-stationary and weight-stationary spatial arrays by setting the *Dataflow* parameter, which will impact the internal configuration of the PEs. DNN workloads often exhibit a high degree of variation in layer shape and size, meaning that weight reuse might be higher in some layers, while output reuse is higher in others [162, 14]. To enable efficient computation across all these types of layers, the runtime-adjustable dataflow support in Gemmini generates PEs that are equipped with multiplexers allowing them to switch between the weight- and output-stationary modes at runtime.

Optional compute blocks: Peripheral compute blocks supporting DNN operations such as non-linear activation functions, max-pooling, transpositions, and image-to-column (im2col) transformations can be optionally left out of the generator to save area and power consumption for custom SoCs which do not require them. This enables a high level of customizability for a variety of workloads both within the DNN domain, as well as for general purpose matrix workloads.

Execution Control: Controller parameters such as the number of entries in the various queues and buffers in the system can be adjusted based on the properties of the outer memory system and the internal compute array. This allows for fine-grained tuning of the controller to achieve balance in the system between the compute array and the SoC memory system.

Gemmini Programming

Gemmini supports multiple programming options. Gemmini can be programmed using fine-grained RISC instructions, as well as through coarser-grain CISC instructions which issue their own sequence of RISC operations using finite state machines implemented in hardware. Both instruction types are issued by the host processor through the RoCC interface as part of the program's instruction stream. Like other RoCC accelerators, Gemmini supports virtual memory addressing by maintaining its own TLB as well as by communicating with the host processor page table walker through the RoCC interface.

In addition, Gemmini provides a multi-layer software flow to support different programming scenarios, illustrated in Figure 3.5. At the *high-level*, Gemmini supports a push-button software flow which reads ONNX-formatted DNN descriptions [90] and generates software binaries that will run them through the ONNX Runtime environment [64, 214]. Alternatively, at the *mid-level*, the generated accelerator can also be programmed through C/C++ APIs, with tuned functions for common DNN kernels such as convolutions fused with activation functions, residual additions, matrix multiplication, and pooling. Finally, at the *low-level*, Gemmini also provides macros allowing programmers to call the Gemmini custom instructions directly, giving the most fine-grained level of control. Gemmini generates an accompanying software header file containing various parameters, e.g. the dimensions of the spatial array, the dataflows supported, and the compute blocks that are included (such as pooling, im2col, or transposition blocks). This enables the high- and mid-level APIs to tune their template implementations based on the properties of the generated instance, and automatically take advantage of whichever Gemmini hardware blocks were chosen to be elaborated. We further elaborate on this generated header file and its usage model in Chapter 6.

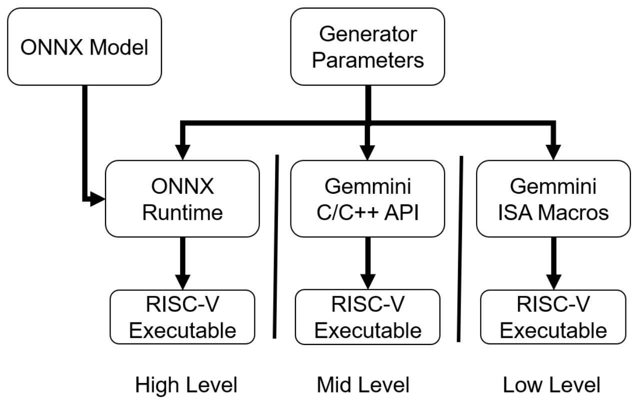


Figure 3.5: Gemmini software programming flows.

3.6 Hwacha vs. Gemmini

Hwacha and Gemmini represent examples of 1-D and 2-D data parallel accelerators. However, they differ in several aspects beyond simply the dimension of acceleration. Table 3.2 contrasts Hwacha and Gemmini across several key architectural and micro-architectural properties.

In contrast to Hwacha, which relies primarily on temporal vectorization, Gemmini is more heavily reliant on spatial acceleration and a large number of execution units. At the same time, the Gemmini controller uses hardware-managed double-buffering for hiding the latency of memory accesses, in order to be able to sustain high utilization of the spatial array of execution units. This hardware-managed double-buffering adds an additional temporal dimension of work distribution to the Gemmini controller, a temporal aspect which can only be amortized for sufficiently large matrices (large than the size of the buffers), similar to Hwacha requiring long vectors to amortize the latency of its deep pipeline. Additionally, the two accelerators use different instruction-issue and control-management schemes: Hwacha uses a vector-fetch model, in which it fetches its own instructions and is not reliant on instruction issue from the host processor, while in contrast, Gemmini does not fetch its own instructions, and is fed instructions directly from the host processor through the RoCC interface. However, Gemmini supports both coarse- and fine-grained instruction schemes, while Hwacha support only one granularity of instructions applicable to the active vector length. In the Gemmini instruction scheme, complex coarse-grain instructions are translated by Gemmini to a sequence of operations that are issued by the Gemmini controller to the execution and memory units. Alternatively, the host CPU can also directly issue simple fine-grained instructions which correspond to these same operations. While Hwacha also breaks down instructions into sequences of micro-operations, these control sequences are not programmable by the host CPU.

Within the context of numerical data analysis, it is important to note that while 1-D vector units provide superior performance on element-wise operation, 2-D spatial units provide a valuable advantage in reduction of operations. This is due to reduced communication between accumulation operations within the 2-D spatial unit. A test-chip in 22nm FinFET process technology, which included both the Hwacha vector accelerator and a very early version of a 16×16 8-bit integer configuration of the Gemmini accelerator, demonstrated that Gemmini could be at least 10× more energy-efficient and 2-5× faster than Hwacha on simple matrix-multiplication workloads [100].

Table 3.2: Gemmini vs. Hwacha Comparison

	Hwacha	Gemmini
Fast Memory	Vector Register File	Scratchpad and Accumulators
Memory Management	VMU	DMA
Data-Reuse	Pipeline	Systolic Array
Instruction Issue	Hwacha (Vector-fetch)	Host Processor (Rocket/BOOM)
Micro-op Issue	Hwacha (Sequencer)	Controller **OR** Host Processor
Scheduler	8-slots (default value, configurable generator parameter)	16-slots (default value, configurable generator parameter)
Sparsity Support	Predicate Registers	None
Datatype Support	Int8 **AND** Int16 **AND** Int32 **AND** Int64 **AND** FP16 **AND** FP32 **AND** FP64	Int8 **OR** Int16 **OR** Int32 **OR** Int64 **OR** FP16 **OR** FP32 **OR** FP64

Chapter 4

Generator-based System-on-Chip Design

In the face of the slowdown in technology scaling, sustaining improvements in system capability requires a greater use of domain-specific architectures. This era of specialization is being seen as a new golden age of computer architecture [121] but creates the challenge of escalating development costs. Differentiated architectures require productive digital system design methods for architectural exploration, system integration, verification, validation, and physical design. To reduce development costs, generator-based agile design processes for hardware development have been proposed with the goal of increasing hardware developer productivity through component re-use and modular design.

Generator-based SoC development enables extensive pre-silicon architectural exploration, validation and optimization based on complete digital RTL designs and test chips, as opposed to more abstract models used in traditional design-space analysis. This "vertically integrated" approach to hardware development, which complements further vertical integration trends across the computing stack, enables tighter design cycles between architecture and implementation through re-usable parameterized implementations.

4.1 System-on-a-Chip

The system-on-a-chip (SoC) represents the evolution of on-die integration of electronic system components. In the past, different components of a computing system such as the processor, the L1 cache, the L2 cache, peripherals, and I/O controllers were all fabricated on different chips and then composed together in packages or on boards. With the continuous evolution of integrated-circuit scaling technologies and electronic design automation (EDA) tooling, the increase in transistor density has enabled single die integration of system components that previously used to be on separate chips. On-die integration increases communication bandwidth and reduces communication costs between system components in terms of both latency and energy. As such, SoC architectures have become popular in energy-conscious use-cases such as mobile application processors and embedded devices.

The core components of an SoC are the application processor, the memory system, and I/O controllers. In modern SoCs, the central processor is often surrounded with a multi-level hierarchical memory system, connectivity peripherals (UART, SPI, USB, Bluetooth, WiFi, etc.), and additional application-specific co-processors. In some embedded SoCs the processors can be exchanged with simpler micro-controllers. Modern SoCs use component specialization in order to maximize performance and energy efficiency. Specialization starts with careful selection of the profile of the central processor. These profiles can range from simple 32-bit micro-controllers to 64-bit high-performance superscalar out-of-order processors. In multi-core SoCs, specialization can span multiple cores resulting in a mix of heterogeneous CPU cores on a single SoC, enabling runtime scheduling of applications to the core which most-closely meets the application's latency, throughput, and power requirements. In commercial products, this approach is branded as *big.LITTLE®* in ARM's commercial offering and *Hybrid Technology* in Intel's commercial offerings (with Intel Lakefield being the first such product), while Apple application processors refer to this mix as "performance cores" and "efficiency cores" (*P-cores* and *E-cores*). This heterogeneity in processors requires software to be aware of the properties of each processor in order to optimally schedule different threads and processes by the operating system of runtime. Furthermore, SoCs typically also include a series of controllers and management processors which perform system control tasks such as power management or serial link management. Each of these management cores can be specialized for their specific tasks, with different levels of support for arithmetic operation or virtual memory, generating even further heterogeneity of processors and software within the SoC.

Specialization of SoCs continues with custom accelerators and co-processors for particular computation kernels. Common accelerators include multimedia transcoders, cryptography computations, integrated graphics, image processing, and signal processing. In a complex SoC, these accelerators can act as independent subsystems using self-contained control processors and memory systems, with on-chip integration allowing for efficient offloading from the CPU while maintaining high performance and energy efficiency.

In this work, since we are interested in numerical data analysis applications, a custom SoC should enable the execution of high arithmetic-intensity kernels for data analysis with high-performance through tight on-chip integration. Such SoCs will require numerical linear algebra acceleration in addition to the standard processor stack, and can include high-performance multi-processors, data-parallel vector units, matrix multiplication units and machine learning accelerators, a coherent memory system, and I/O peripherals. However, as noted in the previous chapter, the exact composition of the SoC depends on the target platform: low power IoT devices, mobile application processors, discrete accelerator platforms, and server SoCs may all require SoCs for numerical data analysis, but under different area, throughput, latency, power, and energy constraints. For simplicity, discussion of the additional levels of complexity added by power management and other control processors within SoCs is outside the scope of this work.

4.2 Generator-based Digital Design

Generator-based approaches to hardware system design have been under development in recent years across several research labs and organizations [198, 22, 170, 234, 126, 48]. In generator-based approaches, hardware-description languages capture design methodologies as opposed to specific design instances. The description and encoding of these design methodologies requires high degrees of paramaterization and modularity. Generators describe digital designs at the RTL level in conjunction with additional primitives from functional programming and metaprogramming to encode a high level of parameterization and modularity. These highly parameterized and modular implementations of digital designs using high-level programming language abstractions enable generation of a broad range of RTL designs.

A generator-based agile design process for hardware has been proposed and demonstrated through a series of RISC-V microprocessor chips developed with small teams [170] and made available as the now widely used open-source Rocket Chip SoC generator codebase [22]. The proposed process and the Rocket Chip project were both based on the Chisel hardware-construction language [23], which is a Scala-based collection of hardware-description functions and operators which can be composed together using Scala metaprogramming and passed through an elaboration program to generate detailed RTL hardware description (in Verilog or a different format). This ability to use metaprogramming in Scala is the key to encoding hardware generators in Chisel. Unlike other high-level hardware-description approaches such as high-level synthesis (HLS), Chisel-based generators do not raise the level of abstraction of the hardware description. Hardware still needs to be explicitly described using RTL functions and operators in Chisel. However, the integration with Scala metaprogamming provides an additional framework which enables the high level of parameterization and modularity which is required of hardware generators.

The Rocket Chip SoC generator includes Rocket, an in-order RISC-V core, and supports coherent caches and standard interconnects via the TileLink protocol. It also includes a library of generator functions which provide the infrastructure of multiple levels of parameterization. Through the use of context-dependent environments (CDEs) [60], the Rocket Chip generator supports a rich parameter system implemented as a key-value dictionary passed through the hardware module hierarchy, enabling generators to make meta-programming choices based on the context of the environment. Designs captured as generators enable reuse through rich parameterization and incremental extension. For example, the parameterization of the core can direct whether it includes components for support of virtual memory, floating-point computation, or other properties which may be considered optional in some systems. The Rocket Chip generator can be used as a starting point for customized SoCs. At its core, the Rocket Chip SoC generator can be thought of as a library of SoC-generator infrastructure that can be used to compose together a system. When additional components of custom SoCs use generator-based primitives and functions from the Rocket Chip library, these components can be composed together in a flexible manner, allowing for increased customizability and a wider range of possible SoCs that can be generated.

The goal of generator-based design is to reduce the development time of custom SoCs from idea to production. Customization ideas can manifest across various levels of the design. We identify five key levels of SoC customization in which generator-based design could provide significant gains:

- Intra-core customization
- Inter-core customization
- Tightly integrated custom accelerators
- Custom peripheral accelerators
- System peripheral customization

These different levels of customization enable designers to tailor the SoC to the exact needs of a target application, allowing for energy and efficiency gains through the reduction in general-purpose support. Through intra-core customization, generators can select the minimal ISA configuration that will support the target application, from the base integer ISA for simple controllers to general-purpose Linux-supporting extensions with floating-point, atomics, and virtual memory support. Inter-core customization can enable designer to provision the SoC for a larger variety of applications through a mix of cores tailored for different operating points from high-efficiency to high-performance (with the different operating point of each core being configured through intra-core customization). Generators can help configure such mixes by providing flexible interconnect generation systems and programming collateral. The most common method to specialize SoCs is through accelerators which target a particular type of domain, with generators having to support their operation through different integration level with a host processor – from ISA extensions tightly integrated with the processor pipeline to peripheral accelerators accessed as devices and shared across multiple subsystems of the SoC. Finally, while often neglected during the conception of custom SoCs, there are many potential peripheral and I/O devices an SoC could support, and generators can help simplify the integration of such devices for each particular SoC use-case through automated device-tree generation and SoC top-level design support. Due to their ubiquitous use, generator-based peripheral devices provide ample opportunity for generator-based infrastructure sharing among SoC designers.

4.3 Generator-based Test Chips

The Rocket Chip generator has been used as a baseline for multiple RISC-V microprocessor test-chips. Customizations to test chips based on the Rocket Chip generator have been performed by adding multiple generations of vector processors, machine learning accelerators, and peripheral devices, by replacing the standard in-order core with an out-of-order core, and by using the Rocket core in various configurations as both an application core and as control cores. The design process outlined by Lee et al. [170] was gradually enhanced on the physical

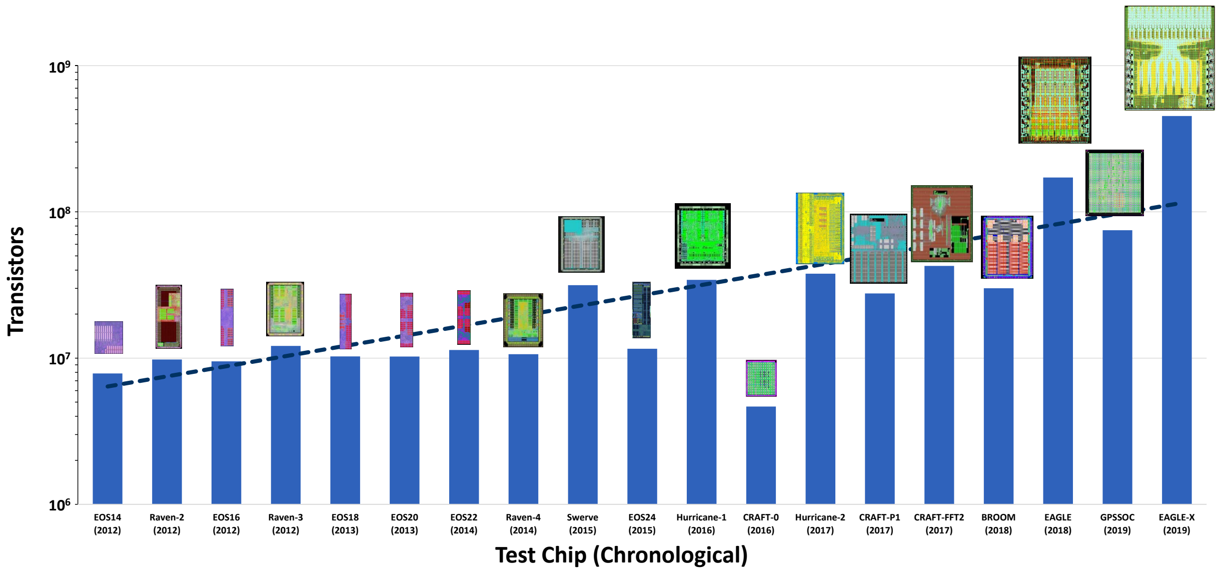


Figure 4.1: Increasing complexity of custom RISC-V SoC test chips built at Berkeley using the Rocket Chip SoC generator between 2012-2019 [13] (© 2020 IEEE).

design front to support much larger silicon dies and many more placeable instances in the SoC, resulting in increasingly complex test chips (Figure 4.1). The most recent test chips in this series are representative of modern SoCs composed of a diverse set of IP blocks and include multiple cores, accelerators, and complete analog subsystems, including high-speed serial links (SerDes), analog-to-digital converters (ADCs) and phase-locked loops (PLLs).

Generator-based design enables rapid prototyping using test-chips of different SoC architectures. However, while test chips provide the most representative evaluation of an SoC design, their integration with mixed signal and packaging concerns still incurs non-negligible costs both in terms of financial resources and engineering resources. Often times, when the primary areas of customization are within the digital domains of the SoCs, designers would like to shift larger parts of the evaluation of the customized design to pre-silicon stages of development. Therefore, extending and expanding pre-silicon evaluation capabilities using generator-based designs is a highly desired property on the path to reducing the number of test-chip iterations in a project and lowering the NRE costs of custom SoC development, while maintaining their complete implementation and fabrication properties. This dual-headed task (increased flexibility of pre-silicon evaluation while maintaining implementation of fabrication properties) exposes the main challenges facing generator-based hardware design.

4.4 Challenges of Generator-Based Agile Hardware Design

While generator-based hardware design provides a flexible approach for customization of SoCs through re-use of generator components and libraries, the practice of this approach through the aforementioned series of test chips has exposed several challenges which still need to be addressed.

Generator Versioning and Compatibility

Generators are software libraries. Most software libraries define sets of APIs and versions for other software packages to depend on. Hardware interfaces are traditionally defined through standard committees and consortiums with long standardization processes. This is in contrast to many open-source software libraries, in which APIs organically evolve with the evolution of the library. This type of organic API evolution for hardware components is not common in the hardware design ecosystem, and therefore sometimes causes challenges with the use of hardware generators. While interface definitions are common in hardware design practices, design versioning and evolving interfaces are not as common, partially due to the degree of verification required of a hardware block and due to monolithic system design practices.

Since generators encode interfaces and protocols using higher-level functions, changes in the internal implementation of the protocols and interfaces may be transparent to hardware blocks which rely on such interfaces. However, the software functions implementing the generator would need to adhere to the updated APIs. Generator versioning and version compatibility are a simple solution to this challenge. Software libraries have long been able to define version-based dependency chains through package managers.

Verification and Validation

Verification and validation are major components of the SoC design cycle. By relying on many previously verified open-source generator components such as the Rocket core, and incrementally extending SoCs based on the Rocket Chip generator, it was often possible to sidestep rigorous verification and validation steps in the aforementioned series of test chips. At the same time, the large space of design instances that can be generated by a generator makes it challenging to perform verification at the generator-level as opposed to a particular instance, due to the exponentially large space of combinations of system compositions and parameters. However, once a sufficiently large number of instances of a generator have been verified, the level of confidence in the generator increases, reducing the verification burden in certain usage scenarios. For example, in the case of test chips, each test chip has a focus on testing either a particular design feature of a module or a component of the design methodology, and therefore, the verification of a test chip can be respectively limited to a small set of functionalities.

A reliance on open-source generators shifts the verification challenge from block-level and unit-level testing to system-integration verification and validation. Rich module-level configurability in the generator-based approach allows for quick, iterative design customization, enabled by expansive configuration and parameter systems. While these parameter systems enable quick iterations across many design points, their flexible nature makes them prone to misconfiguration, underscoring the need for continuous full-system integrated validation and verification. While this parameterization makes it possible to continuously update the SoC design in an agile way with late-breaking changes, it does not inherently aid the design verification of the chip with those changes. Issues relating to on- and off-chip interfaces, clock domain crossings, third-party IP integration, and power management are all vulnerable to insufficient full-system verification coverage. Some previous test chips designed using the generator-based approach have encountered both verification and validation gaps of this type: for example, one chip's cache capacity was inadvertently reduced to half of its desired size, when the configuration was changed late in the design process to meet physical design constraints.

At the same time, generator-based design can also make it difficult to verify blocks at the unit-levels, due to system-level generator behavior. For example, the Diplomacy framework within the Rocket Chip generator [62] uses two-stage elaboration for negotiating parameters across devices on shared buses in order to generate optimal bus instances without over-provisioning of resources. As such, the generator requires the context of the full SoC in order to determine the parameters and interfaces of individual hardware blocks. This adds an additional level of challenges to unit-level testing of individual block generators, since trustworthy unit-level verification would require the context of the complete SoC [180]. This further encourages verification of generator-based designs at the system-level as opposed to the unit-level, in order to verify the true hardware interfaces that will be implemented in the system by a fully configured generator.

Work Distribution

Although design and verification executed by a small team in an agile manner has a high appeal for small companies and industrial and academic research labs, our experiences with test chips developed through an agile process also identify some challenges in execution. The increasing complexity of chips makes it difficult to parallelize and distribute effort among the small number of designers with different toolsets and design configurations, as architecture definition, RTL implementation, physical design, verification, and validation all take varying amounts of time, which is difficult to account for. Furthermore, it is difficult to maintain institutional memory of good generator-based design practices, especially in academic environments and when working with complex physical design tool flows in deeply scaled technologies.

Process Technology Transition

Transitioning between process technologies is a significant undertaking in both system and chip design. The aforementioned generator-based test-chips have been designed in several different process technologies, including IBM 45nm SOI, ST 28nm FD-SOI, TSMC 28nm, TSMC 16nm FFC, and Intel 22nm FFL. The generator-based approach simplifies process technology transition, since it allows for adjustment of design parameters through high-level descriptions. However, this flexibility does not propagate across the abstraction layers to the physical design process. Each custom test chip requires significant manual effort in mapping RTL abstractions to process-specific components (such as memory and register-file macros), as well as meeting the design rules. When generator parameters change between design iterations, a new rigid physical design script is often created. These constraints have encouraged early design freezes, sometimes before full-system integrated testing and validation. Process technology transition is a significant undertaking in chip-design - Intel's well-known "tick-tock" approach dedicated an entire chip generation to process technology transition. As deeply scaled technologies add increased complexity to process design kits (PDKs) and associated EDA tools, process technology transitions across design are becoming increasingly lengthy and often require breaking the RTL design abstraction.

4.5 Integrated Generator-Based Design Methodology

The approach proposed by Lee et al. [170] presented important pillars of generator-based agile hardware design, but requires additional refinement in order to be captured as a robust methodology. Based on the observations from multiple generator-based test-chip development cycles, we identify several steps and principles that can be captured and further distilled as part of a generator-based design methodology. Principally, inherent dependencies across agile design iterations require tighter integration of generator-based tooling and the associated development flows. Reusable hardware generators extend the design space of possible SoCs that can be tailored to a target application. The broad multi-dimensional design space shifts the balance of the design cycle, and requires further design-space exploration time as opposed to instance RTL design time. We elaborate on several core principles of a more integrated generator-based design methodology.

Component Composition

Modularity and composition are important principles in generator-based design, as well as in general hardware and software engineering. The meta-programming aspect of generator-based design raises the question of module composition between generated sub-components. One possible approach is composition at the generated-RTL level through the definition of "hard-coded" interfaces for the generated modules. With this approach, modules are generated individually, and then composed and wired together together as part of a top-level module using manual integration or a simple automated scheme. An alternative approach is composition at the generator-level, similar to the infrastructure provided by the Diplomacy [62] framework.

Composition at the generated-RTL level is fairly straightforward, as all it requires are agreed-upon interfaces between sub-components. This allows for generators of different components to be developed individually using their own unique semantics and meta-programming conventions. As long as the generated RTL adheres to the interface specification, components will be able to be composed together into a system. However, composition at the RTL-level relies on optimizations performed by the underlying synthesis tools to optimize over-provisioned interface definitions, and may also require manual intervention upon any architectural or system-level change which impacts the interface on either side of it.

In contrast, composition at the generator-level requires a holistic view of the entire system in order to enable optimizations and composition by the generator framework. The requirement for a holistic view necessitates more advanced techniques such as custom generator-level interface specification semantics and multi-stage elaboration, as implemented for example by the Diplomacy [62] framework. As a result, generator components need to adhere to these custom composition elaboration semantics, and select a single "generator manager" to perform the system-level composition.

Composition at the generator-level requires a common "standard library" across generators. This "standard library" goes beyond hardware generation languages such as Chisel, since it needs to have knowledge about basic semantics of the digital design such as synchronous design, interconnect structure and memory semantics. The Rocket Chip generator library acts as just such a standard library within the Chisel ecosystem, which can be used for processor-centric digital designs. With common generator standard libraries, generators can be designed and developed individually as long as they adhere to the semantics of a common version of the base library. These standard libraries can be versioned accordingly, assisting in appropriate versioning and version-dependency management across generators and generator base libraries. Figure 4.2 compares the role of standard libraries in generator-based hardware design to software standard libraries and the absence of standard libraries in traditional hardware design.

Generator/Instance Hybridization

The *design* of hardware generators requires a thorough understanding of RTL-based digital design considerations as well as advanced software-engineering concepts such as functional and object-oriented programming, meta-programming, code generation and code transformation. This results in a small number of experts being able to design and maintain advanced and highly parameterized hardware generators. However, the *usage* of hardware generators is significantly simplified through modularity and parameter-configuration schemes. These simpler usage models enable democratization of hardware generators to a broader audience of hardware developers.

One approach to lower the barrier-of-entry to generator-based custom SoC design is a hybridization of generators and specialized instances. In this approach, the core components

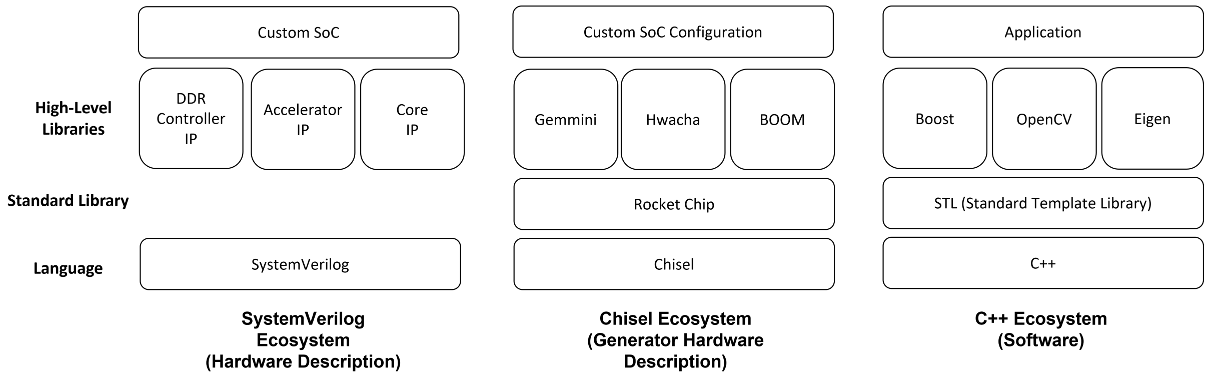


Figure 4.2: Standard libraries provide an additional semantic layer between the programming or description language and domain-specific libraries. Standard libraries provide common primitives which can help generate interfaces with more complex common semantics, and are prevalent in software programming languages, such as C++. Traditional hardware description languages such as SystemVerilog typically do not have standard libraries. Generator-based hardware design methods, which incorporate higher-level semantics than traditional hardware description languages, benefit from standard libraries such as the Rocket Chip generator, which provide common primitives that can be used by higher-level generator libraries to better communicate and compose together.

of the SoC such as CPU, memory system, buses and a subset of peripherals and controllers on the SoC are generator-based, while specialized custom components such as accelerators and additional peripherals are instance-based.

An agile generator-based methodology requires the identification of common SoC components across multiple generations and variations of the SoC, and the allocation of sparse generator design expertise to the development of the core generator infrastructure. Alternatively, generator design expertise can be extracted from the open-source domain through re-use of existing open-source generators.

Specialized single-use accelerators and peripherals can be implemented using traditional Verilog-based instance design patterns and tools, and be integrated within a generator-based SoC, as illustrated in Figure 4.3. These single-use instances can be designed using the general digital design skill-set, enabling broader adoption and participation by hardware engineering professionals. This type of Generator/Instance hybridization requires sufficient support in generator-based tooling, and sufficient knowledge of generator usage models by specialized instance designers to enable testing and verification.

Design-Space Exploration

Design-space exploration for custom chips is traditionally performed using high-level architectural modeling or spreadsheet-based back-of-the-envelope estimations. Such architectural software models are primarily written in high-level languages (Python, C++) and are in-

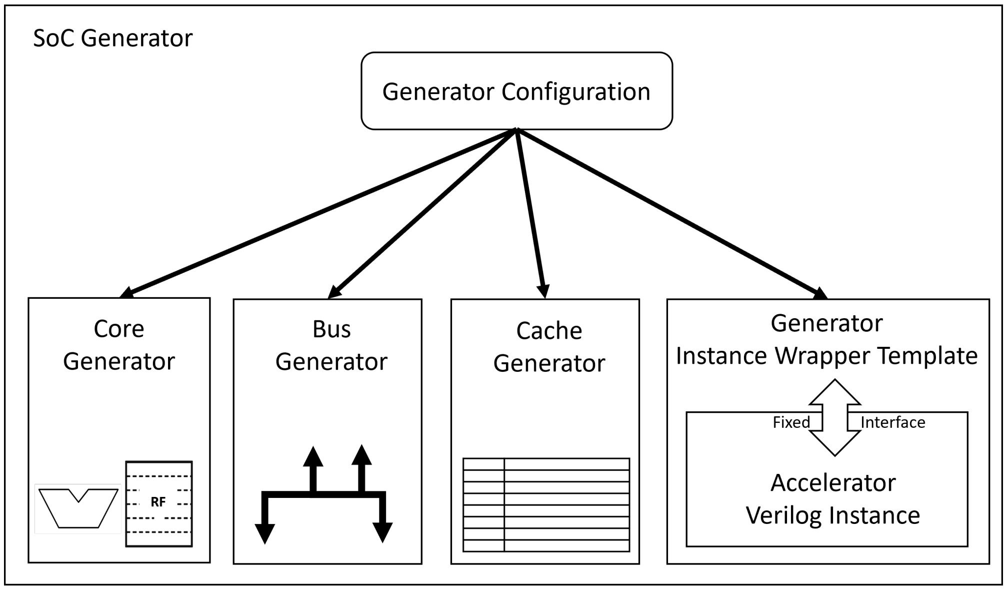


Figure 4.3: Generator/instance hybridization. Specialized single-use accelerators and peripherals can be implemented using traditional Verilog-based instance design patterns and tools, and be integrated within a generator-based SoC by using fixed-interface wrappers integrated with the generator configuration system.

tended to approximately estimate the consequences of various design choices. As a result, this type of modeling is often coarse-grained, especially with respect to power-performance tradeoffs and interactions between SoC components. At the same time, this approach to modeling is also very flexible, as high-level models provide for extensibility and fast modulation without time-costly concerns for implementation details.

In the traditional "waterfall" serial development model, architectural design-space exploration is one of the first stages of development. As such, it is allocated a fairly short amount of time, since all of the following development stages depends on its results. In particular, the implementation stage cannot start until the architectural decisions are made. This stage is especially costly during the first version of a custom SoC, since it requires exploration of a broad swath of fundamental components including processor architecture and IO choices. These choices are often made with very little knowledge of fine-grained implementation details. Consequent iterations of a similar product may build upon the fundamental design choices made by the first iteration with the additional insight of implementation details from the already-implemented previous SoC version.

With a generator-based approach, a single generator implementation can provide insights on a broader set of design points. Furthermore, the availability of open-source hardware generators enables the initial design-space exploration stage to be based on generator implementations as opposed to high-level models. This approach can lead to the identification of fine-grained performance phenomena and pathologies during pre-silicon design-space explo-

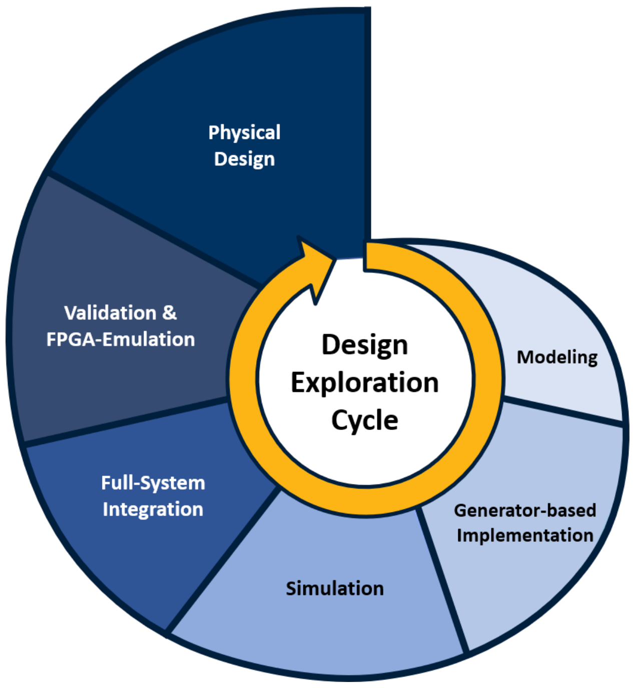


Figure 4.4: Iterative generator-based design space exploration cycle, with respect to the varying time and resource consumption of different stages of the development cycle

ration, rather than late implementation stages. An example of generator-based identification of such performance phenomena related to a Linux-based networking stack is demonstrated in the FirePerf project [151]. As a result, a generator-based methodology allocates a larger time budget to generator-based design space exploration than a traditional serial "waterfall" development methodology. This additional time budget allocation comes at the expense of later-stage implementation time budget allocations, which are reduced thanks to re-use of configurable generator components.

Pre-Silicon Validation and Verification

Traditional chip-design verification and validation processes rely on a mix of unit-testing, integration testing, coverage-based and formal verification to provide a sufficient level of pre-silicon confidence in the functionality and performance of an SoC design.

Since the verification complexity increases as the size of the design grows, end-to-end integration testing is limited in the scope and time. At the same time, individual IP components rely on self-contained unit-tests with custom test harnesses (sometimes called "Verification IP") which attempt to emulate the potential interface behavior of the surrounding system.

A generator-based approach to verification of custom SoCs advocates for *end-to-end* testing and evaluation of custom IP components. SoC components are often designed around a central host processor (the CPU) and typically require interaction with some of the SoC

I/Os. As a result, design-time end-to-end verification of IP components necessitates a sufficient level of integration with the minimal set of SoC structure blocks such as interconnect buses, I/O, memory system, and the CPU. A generator-based SoC design methodology provides this type of system support, enabling end-to-end pre-silicon verification. An SoC generator can generate a basic and minimalistic CPU and SoC structure which can be used as a verification test harness while a final CPU design point and SoC architecture have not been selected and completed. This approach enables verification based on actual target software execution during early-stage design of custom IP components in contrast to testing and verification based on artificial and synthetic signal activity. This approach does not diminish the importance of individual component unit testing and dedicated verification IP, but rather provides an additional level of verification (and to some extent, can be thought of as a variant of verification IP) which is more representative of potential software-driven behaviors of the host system. Recent efforts in co-simulation of generator-based IP provide a complementary approach to end-to-end verification by helping overcome some of the challenges of unit-level verification of individual generator-based blocks [180], helping increase the verification coverage of unit-level generator testing. However, a generator-based approach to verification would still be highlighted by complete end-to-end testing of custom IP components in order to enable to highest possible fidelity of results and verify potential system-level generator configuration consequences.

VLSI Mapping

During the era of Moore's law and Dennard scaling, the choice of selecting a fabrication process technology was fairly simple. The cost-per-transistor was constantly decreasing while each new process technology provided better power and performance profiles. Hence, for a selected application, the choice of fabrication process technology could be succinctly summarized as a strictly "greater than" relationship between performance improvement and cost increase. The cost increase of using a newer process technology was almost always smaller compared to the SoC performance and efficiency improvements that resulted from that newer process technology. Therefore, the choice of process technology was often determined by performance/efficiency requirements rather than fabrication cost or total cost of ownership (TCO) constraints.

However, with the end of Dennard Scaling and Moore's law, the marginal gains of more advanced process technologies are decreasing compared to the marginal cost increases. As a result, composing a system out of multiple chips in an older fabrication process technology (and perhaps even different process technologies) may turn out to be more cost-effective than integrating the entire system on a single chip in an advanced scaled process technology. Some models have already identified the benefits of using older fabrication process technologies for a category of custom cloud-class ASICs [156].

As generational performance improvement is no longer a proxy metric for the cost of the process technology, decisions regarding the choice of process technology may change at much later stages of the SoC development process as the overall costs of development crystallize.

As such, it is important for generator implementations to not be tied to a specific process technology, but rather provide the ability to map the generated digital designs to different process technologies through generalized transformations.

Higher levels of abstraction at the generator and RTL levels are required for this type of flexible mapping. For example, the Chisel embedded hardware construction language treats memories as a first-class primitive [23]. As such, mapping on-chip memories to various SRAM compositions or even flip-flops can become a portable mapping-time decision based on the process technology, rather than a decision that is encoded in the design RTL. Similarly, I/O handling often requires technology-specific cells, which can be mapped with a sufficient level of portability if the generator has a higher-level notion of chip-level I/Os.

Sufficient decoupling between physical design transformations and digital generator design can also enable more detailed design-space exploration by designers with a lower-level of VLSI expertise. This can be done by providing automated mapping transformation between the generated RTL and select process technologies to output area, power and timing reports with a level of accuracy that is not as high as the one that could be obtained through manual physical design, but sufficiently detailed compared to estimates from high-level modeling.

Integrated Tooling

By definition, generator-based SoC development methodologies require a greater focus and time investment in design automation tooling at the RTL abstraction level compared to traditional development flows. This was also noted in the *agile hardware manifesto* by Lee et al. [170]. The "meta-programming" aspects of generator-based design make temporary fixes or manipulations of a single instance difficult to sustain across multiple development stages and design iterations. At the same time, the requirement of generator/instance hybridization at the RTL-level as well as compatibility requirements with legacy EDA tools for synthesis and physical design require continuous interchange at the Verilog abstraction and language level.

While the commercial EDA industry consists of many tools used across the VLSI design process (albeit, from a small number of vendors), an important aspect therein is the integration of these tools into "flows". Previous analysis of the evolution of the electronic design automation industry [164, 225] has identified the fundamental role that complete design flows play in the evolution of EDA, and specifically that the latest "age" of EDA is characterized by increased integration between tools. In the EDA industry, this integration started with the integration of VLSI physical design tools for timing and power analysis together with routing and placement optimization. It required common data models and databases, as well message passing protocols between tools. Additional steps of the design flow such as synthesis, simulation, and verification are integrated, captured and automated using commercial EDA "flow tools" such as the Lynx Design System from Synopsys [250]. Recent open-source projects in the EDA domain also put an emphasis on providing complete flows, such as the OpenLane flow [235] using tools from the OpenRoad [4] project.

Since digital design flows are composed of a large number of tools, successful integration of generators and their associated methodologies into digital design flows must be captured through automation of generator-based flows and tighter integration between generators and their associated tools. Automation and integration can be performed directly between generators and generator-based tools, or between generators and traditional tools. The former allows for transfer of information between the generator and the tool (this information can be captured in the form of "annotations"), while the latter may require some transformation of the generator output in order to adhere to requirements of the traditional tool.

Information transfer between generators and tools has many uses in the mapping, analysis and simulation of generator-based designs, therefore, tight integration between generators and generator-based tools may have multiple benefits for the usage flow of these tools. Tight integration of automated generator-output transformation between generators and traditional EDA tools can further enable a broader application of generator-based developments to a wider spectrum of digital designs. Hence, an integrated generator-based methodology relies heavily on an integrated tooling environment for both generator-based and traditional tools.

4.6 Generator-based System-on-Chip Design Summary

Generator-based hardware development is a paradigm which relies on re-use of libraries and meta-programming for rapid implementation of a wide range of hardware design configurations. It enables extensive pre-silicon architectural exploration, validation and optimization based on the complete digital RTL designs and test chips. In this chapter, we discussed the applicability of generator-based hardware development to SoC design, and the challenges exposed when applying this approach for the design of a series of generator-based test-chips. These include the challenges of generator versioning and compatibility across generators, difficulties in verifying generators vs. verifying instances, complications in the distribution of work between team members in generator-based design due to the varied lengths of different stages of the development process, and the overheads of rapid process technology transitions. We therefore propose methodological steps to address these challenges, which highlight common standard libraries to support generator composition, the ability to integrate together generators and instances, integrated tooling to support more efficient work distribution, utilizing detailed generator-based implementations for design space exploration to reduce late-stage re-design events, end-to-end verification through complete system support, and portable VLSI mappings through automation and high-level abstractions. Chapter 5 will discuss the framework and tools to support the implementation of these methodological steps.

Chapter 5

The Chipyard Integrated SoC Development Framework

Chipyard is a framework designed to provide SoC designers and architects with a unified framework and workflow for agile generator-based SoC development. We developed Chipyard to capture the principles introduced in Chapter 4 by composing a variety of open-source tools and generator-based IP into a unified framework through automation and integration. Multiple separately developed and highly parameterized IP blocks can be configured and interconnected to form a complete SoC design. The SoC design can be verified and validated through multiple simulation, emulation and prototyping flows, and then pushed through portable VLSI design flows to obtain tapeout-ready GDSII data for various target technologies. Chipyard also provides a workload-management system to generate software workloads to exercise the design.

Chipyard Front-End RTL Generators

The front-end of the Chipyard framework is based on the Rocket Chip SoC generator [170, 22]. Chipyard inherits Rocket Chip's Chisel-based parameterized hardware-generator methodology [22], including a Scala-based parameter-negotiation framework, Diplomacy [62], that negotiates mutually compatible parameterizations and interconnections across all IP blocks in a design. A unified top-level SoC generator enables the generation of heterogeneous systems based on parameterized configurations. Chipyard allows IP blocks written in other hardware languages, e.g., Verilog, to be included via a Chisel wrapper.

Chipyard adds a large corpus of open-source IP generators to the existing Rocket Chip base library, allowing for the construction of modern digital SoCs. These include the Berkeley Out-of-Order Machine (BOOM) generator [47, 288], the CVA6 core (formerly known as Ariane) [285], the Hwacha vector-unit generator [173, 228], digital signal processing (DSP) modules, domain-specific accelerators such as the Gemmini deep-learning accelerator [96], memory systems, and peripherals. The majority of these generators have silicon-proven instances in a variety of process technologies. While some commercial IP vendors have large

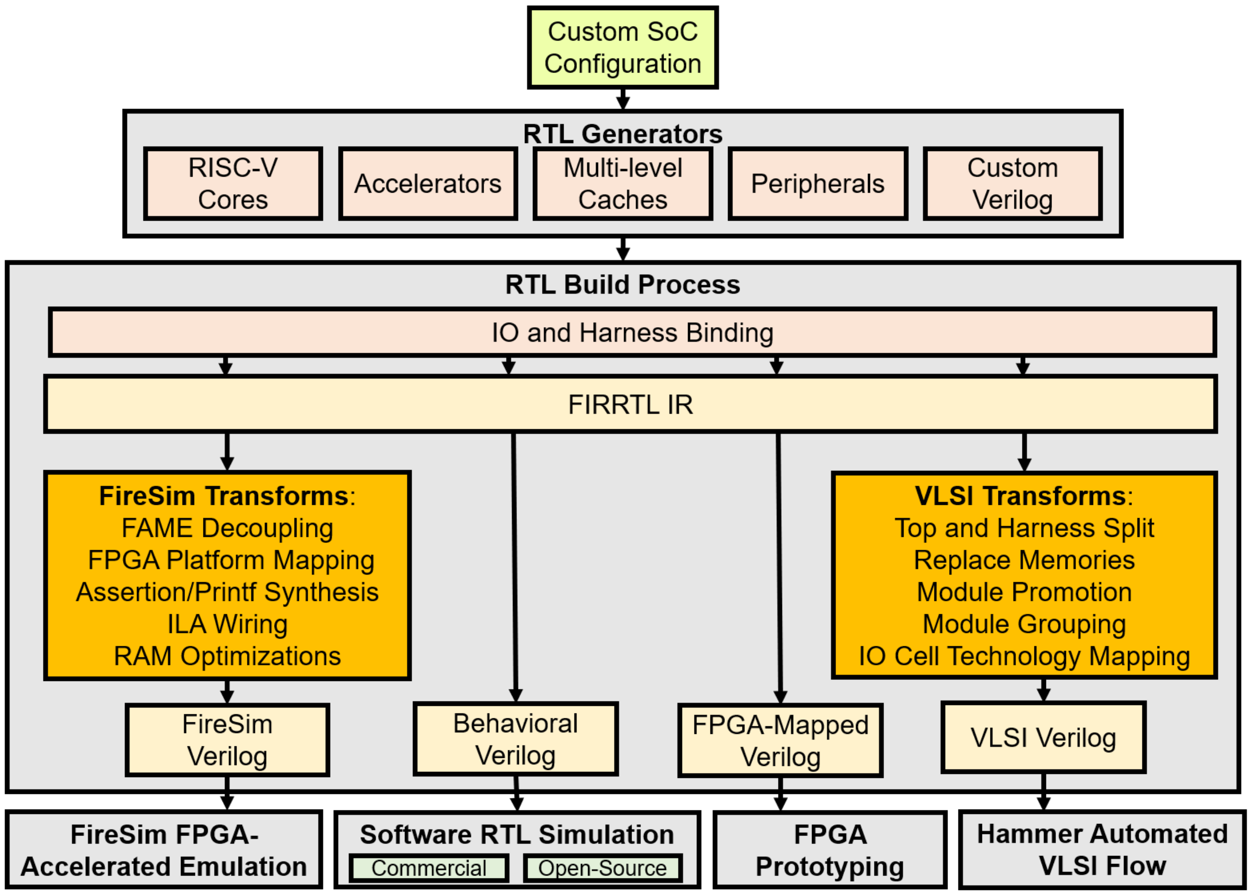


Figure 5.1: Multiple disparate design flows supported by the Chipyard framework through generators and transformations. Starting from the same generators and common custom configuration, a series of FIRRTL transformations outputs appropriate Verilog and associated collateral for different design-stage platforms.

collections of proprietary configurable IP for certain portions of an SoC, Chipyard provides a publicly extensible open-source alternative for complete SoCs to support continuing research and development of specialized state-of-the-art SoCs.

Other open-source SoC design frameworks focus on tile-granularity customization in many-core architectures [26, 186], or rely on rigid subsystems and proprietary IP [275]. The generator approach used in Chipyard does not rely solely on static interfaces for integration of IP blocks, but allows for dynamic customization of encodings, memory maps, and buses during the hardware generation stage, enabling custom components to be created and integrated at various levels, including in the MMIO periphery, as tightly integrated accelerators, and as heterogeneous cores and controllers. For example, through its fine-grained intra-core parameter system, the Rocket core can be used for different purposes in an SoC. Specifically, in several of our recent test chips, the application cores consist of fully Linux-capable

Rocket cores supporting RV64GC with floating-point units and virtual-memory support, while the controller cores (e.g. a power-management unit) are a Rocket core supporting only RV64IMAC, with significantly smaller branch-prediction and cache resources and no virtual memory. In essence, a significant portion of the microcontrollers running firmware on the SoC can be implemented using variants of Rocket or BOOM cores. This common core configuration interface for all software-managed controllers within the system improves designer productivity, compared to alternative SoC development frameworks that enable drop-in replacement core options or only coarse-grained sizing. Similarly, machine-learning accelerators have been integrated with Rocket and other core generators as tightly integrated accelerators [95], as well as in the form of MMIO periphery accelerators [86], demonstrating the various levels of possible customization in the Chipyard framework.

FIRRTL Intermediate Representation

The Chipyard framework currently integrates tools to address the three main activities within the custom SoC design cycle: front-end RTL design, system validation/verification, and back-end chip physical design. These different activities require different levels of design description. For example, while front-end RTL descriptions usually use abstract notions of memory and I/O, back-end RTL requires more precise descriptions mapped to the underlying process technology. Similarly, FPGA emulation or prototyping requires the digital design to interact with FPGA-specific interfaces, periphery, and internal components. Co-simulation also requires additional hardware clock gating to control simulation progress.

Chipyard elaborates the front-end RTL design into a FIRRTL [143] intermediate representation. Custom FIRRTL transformations convert the generated FIRRTL design to drive the different flows used at different stages of the design cycle. Using FIRRTL transformations to enable multiple disparate design flows from the same shared code repository and source RTL helps to reduce and amortize the environment setup costs incurred with frequent iterations between development stages, as is needed for an agile methodology. This approach is demonstrated in Figure 5.1.

While Chisel is the primary language for design entry in Chipyard using the FIRRTL compiler, a FIRRTL-based flow can integrate Verilog IP through either “Blackbox” IP integration or Verilog-to-FIRRTL support by certain Verilog elaboration tools [278]. Furthermore, while the Verilog outputs of various stages of a FIRRTL-based flow can be integrated into standard dynamic verification environments or compared using logical equivalence checking, tools for both simulation and temporal property checking of “FIRRTL-native” circuits are openly available [183].

Verilog or SystemVerilog-based design frameworks [275, 26] must rely on design-specific custom scripts or interface adjustments when transitioning between emulation, simulation and physical design. In contrast to alternative hardware package-management systems [158] or integration standards like IP-XACT, which focus on metadata associated with particular IP components to target different EDA flows, FIRRTL transformations can perform wholescale manipulation of complete RTL designs in Chipyard.

Software RTL Simulation

Software-based RTL simulators are a critical tool in most phases of the design process. Compiling a software simulator of a top-level design, including various IP components, peripheral and memory models, and an external test harness can be a time-consuming engineering task. Chipyard provides build flows for both the open-source Verilator simulator and proprietary commercial simulators. Open-source RTL simulators such as Verilator are also used in industry [70] to provide efficient and cost-effective digital-design verification. Chipyard provides Makefile wrappers for direct generation of a simulation executable which simulates tethered designs with emulated peripherals. The Makefile wrappers generate the top-level design and matching test harnesses based on the SoC configuration. Tethered designs use a host to send transactions that bring up the simulated SoC and load programs. These software RTL simulation wrappers enable quick design cycles and execution of RISC-V binaries in simulation. While tethered designs are the default form to generate software RTL simulations in Chipyard, Chipyard also supports un-tethered SoC configurations in which the SoC can boot standalone using a specialized boot ROM.

FPGA-Prototyping

FPGA prototyping is an important tool in software development for SoCs. FPGAs can be used to prototype the functionality of the digital design and the interaction of software with a variety of I/Os and system peripherals.

While the Verilog RTL generated by Chipyard could be manually mapped to an FPGA for prototyping purposes, Chipyard also provides integrated support for FPGA prototyping through FPGA shells. *FPGA-shells* is an open-source project by SiFive which provides a Chisel-based layer of abstraction between specific FPGA devices and Rocket-Chip-based generators [136]. The FPGA shell defines the top-level interfaces that are common in FPGA devices. Users can create *Overlays* within a shell, which connect top-level SoC interfaces generated by the Rocket Chip generator to FPGA-device-specific pins through FPGA IP blocks. The FPGA-shells project includes several default overlays for common interfaces such as UART or SPI. Chipyard adds additional example overlays for FPGA interfaces that are typically project-customized, such as an FPGA Mezzanine Card (FMC) interface or GPIO interfaces. These Overlays are placed with a new Chipyard test harness, which allows seamless switching between different FPGA platforms using the same SoC configuration.

FPGA prototyping can be used to prototype the integration of a Chipyard SoC with additional system-level components such as network and connectivity interfaces or multimedia inputs and outputs, or as flexible testing platforms for experimental test-chips and boards. For example, the Chipyard FPGA-prototyping support is frequently used as a host processor for bring-up of tethered RISC-V test-chips. The host system is implemented as an FPGA prototype of a BOOM processor on the FPGA, and drives the system under test using the FPGA board's FMC interface which connects to the test-board and test-chip I/O. The FPGA prototype loads the program into the test chip's memory through a serial or

JTAG interface, which can be controlled through software executing on the prototype's processor running on the FPGA. While FPGA-prototyping has many uses, we note that FPGA prototyping is not an accurate method for evaluating the performance of a simulated custom SoC. FPGA-accelerated emulation or simulation would be a better choice for such tasks.

FPGA-Accelerated Simulation with FireSim

For full-system validation and evaluation, the Chipyard framework harnesses the FireSim [152] open-source FPGA-accelerated simulation platform using the AWS EC2 public cloud. In contrast with FPGA *prototyping*, FPGA-accelerated *simulation* correctly models timing behavior of not only the design under test, but also the I/Os and peripherals of the SoC. Furthermore, FPGA-accelerated simulation in FireSim enables deterministic and reproducible evaluation within a realistic system environment, as opposed to FPGA prototyping where each execution is sensitive to the FPGA environment and timing depends on the performance of peripherals attached to the FPGA (e.g. DRAM performance). FireSim also provides FirePerf [151], a set of powerful on-FPGA out-of-band performance profiling tools that enable high-fidelity cycle-by-cycle introspection into software running on the simulated system, without perturbing the target system.

Originally developed as a platform to enable scale-out simulation for datacenter architecture research on hundreds of cloud FPGAs, FireSim automates the infrastructure management and simulation mapping necessary to automatically run high-performance simulations. As part of the agile chip-design stack, this automation and integration reduces the level of expertise required to harness cloud FPGAs for emulation purposes and thus increases the accessibility of high performance full-system simulation to a broad spectrum of designers. FireSim has been useful in pre-silicon verification, validation, and software development. From the perspective of small agile teams with limited resources, FireSim provides many of the features available in costly commercial emulation platforms. In contrast with prior FPGA-accelerated simulation tools, the accessibility of FireSim through FPGA instances on the AWS public cloud, as well as the automation of host-target interfaces with the FPGA, have made FireSim a popular tool within Berkeley and other academic hardware-development users, as well as emerging startup companies.

FireSim enables co-development of software and hardware simultaneously, allowing for quick software adjustment turnarounds based on hardware modifications. Furthermore, FireSim plays a major role in the performance and functional validation of processors, since it enables the identification of bugs deep into simulation execution time thanks to FPGA-acceleration with appropriate peripheral modeling. Unlike many other open-source hardware development platforms with FPGA support, FireSim's focus on simulation and emulation as opposed to prototyping enables true pre-silicon performance evaluation and validation in a full-system context within the Chipyard framework. While maintaining its stand-alone operation as an architectural research platform, FireSim was transformed into a library which is integrated into the broader Chipyard framework. As such, FireSim can now consume design configurations composed within the Chipyard framework and transform them into

FPGA-accelerated simulations. Furthermore, the FireSim Golden Gate compiler has been integrated into the Chipyard framework, so it can now consume arbitrary FIRRTL as its input, as well as external Verilog components necessary for broader system integration.

Back-End Physical Design with Hammer

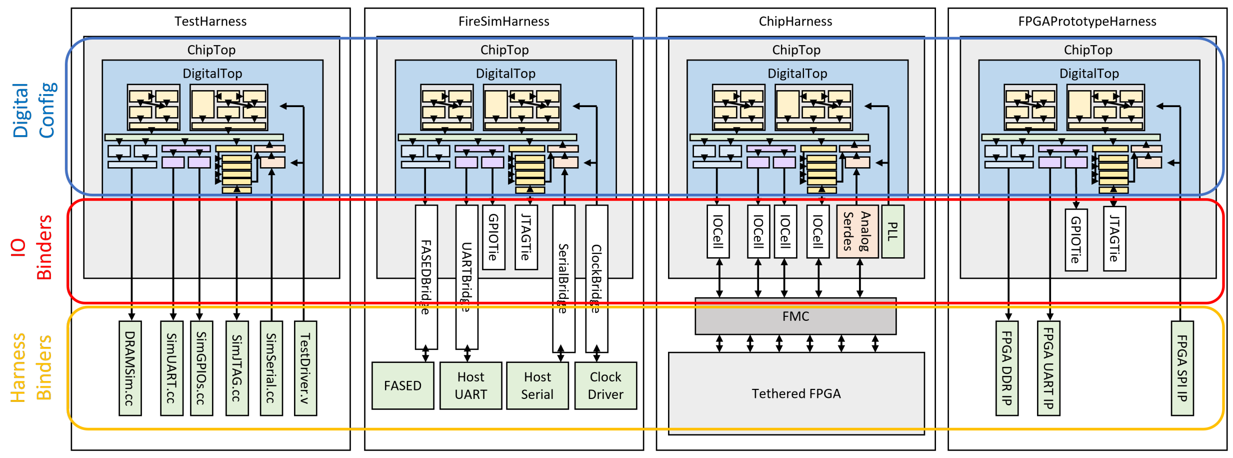
For back-end physical design, Chipyard includes a modular VLSI flow named Hammer [269]. The Hammer VLSI flow provides an abstraction layer above process-technology- and EDA-tool-specific concerns, with the goal of increasing re-use and modularity of vendor-specific components of the physical design flow. To this end, the Hammer VLSI flow utilizes separate vendor-specific process technology plug-ins and EDA-tool-specific plug-ins, which implement abstracted software APIs to generate design-flow collateral like Tcl scripts, clock constraints, and power specifications based on higher-level design inputs. For example, Hammer will emit process- and vendor-specific macro placement, obstruction, and power-strap placement commands from a high-level process- and vendor-agnostic description of the design. This separation of abstraction layers between design, process technology, and EDA tool vendor enables faster adoption of open-source components.

The Hammer flow aspires to support open-source tools in conjunction with commercial and proprietary tools by using common levels of abstraction. As such, while the first Hammer-based designs were implemented using proprietary process technologies, a plug-in for the ASAP7 [57] open-source predictive PDK was created in only a few weeks and is now included in the core Hammer repository. Similarly, a plug-in for the Skywater-130 (SKY130) process technology has been development in order to support open-source SoC development, in conjunction with driving designs through bleeding edge process technologies. With this, small teams and academic users can prototype design flows and experiment with RTL designs using predictive or simple physical design kits, while being able to reuse similar Hammer descriptions for chip fabrication using advanced process nodes.

Hammer was designed to support hierarchical physical design flows. Hierarchical physical design flows are of particular importance in highly complex custom SoCs, composed of multiple specialized blocks with a variety of physical design constraints. Decomposing a design into these smaller hierarchical components not only improves the quality of results emitted by EDA tools, but it also allows the distribution of physical design tasks among multiple hardware developers, which is important for agile design. FIRRTL-based grouping and flattening transformations in Chipyard further assist the hierarchical physical design flow in Hammer by enabling users to specify one logical hierarchy in the source RTL, while choosing a different hierarchy for physical boundaries through automated transformations.

Input/Output Management

The various implementation and simulation flows in the SoC design process will typically treat the digital I/Os on a system in different ways. For example, a software RTL simulation would typically connect digital I/Os directly to software models in the *TestHarness*. However,



```
class CustomConfig1 extends
 Config(
  new WithDefaultGemmini ++
  new WithNRocketCores(1) ++
  new WithNBoomCores(1) ++
  new WithBootROM ++
  new WithUART ++
  new WithJtagDTM ++
  new WithGPIOs ++
  new WithInclusiveCache(512) ++

  new WithPassThroughIOs ++

  new WithDRAMSim ++
  new WithSimUART ++
  new WithSimJTAG ++
  new WithSimSerial
 )
```

```
class CustomConfig2 extends
 Config(
  new WithDefaultGemmini ++
  new WithNRocketCores(1) ++
  new WithNBoomCores(1) ++
  new WithBootROM ++
  new WithUART ++
  new WithJtagDTM ++
  new WithGPIOs ++
  new WithInclusiveCache(512) ++

  new WithSerialBridge ++
  new WithUARTBridge ++
  new WithBlockDeviceBridge ++
  new WithFASEDBridge ++
  new WithFireSimIOCellModels ++

  new WithDefaultMemModel ++
  new WithUART ++
  new WithDefaultSerialTL ++
  new WithFireSimSimpleClocks
)
```

```
class CustomConfig3 extends
 Config(
  new WithDefaultGemmini ++
  new WithNRocketCores(1) ++
  new WithNBoomCores(1) ++
  new WithBootROM ++
  new WithUART ++
  new WithJtagDTM ++
  new WithGPIOs ++
  new WithInclusiveCache(512) ++

  new WithTSMCIOCells ++

  new WithBringupFMC ++
  new WithBringupBOOMTether
 )
```

```
class CustomConfig4 extends
 Config(
  new WithDefaultGemmini ++
  new WithNRocketCores(1) ++
  new WithNBoomCores(1) ++
  new WithBootROM ++
  new WithUART ++
  new WithJtagDTM ++
  new WithGPIOs ++
  new WithInclusiveCache(512) ++

  new WithPassThroughIOs ++

  new WithUART ++
  new WithSPISDCard ++
  new WithDDRMem
 )
```

Figure 5.2: Chipyard I/O and harness management using configurations with IOBinders and HarnessBinders. Different design flows require different I/Os and harnesses, with potential overlaps across platforms. Decoupling I/O and harnesses from the core digital design allows for portability across design flows, together with re-use of I/O and harness components across platforms.

on an FPGA prototype or FPGA-accelerated simulator, these I/Os would require some synthesizable “bridge” which can map the I/Os to the interfaces of the FPGA. For physical design, I/O cells must be inserted into the module hierarchy.

In order to handle the various methods of providing input and output to the SoC across different simulation and implementation flows, the Chipyard framework uses *IOBinders* and *HarnessBinders*. IOBinders define the attachment behavior of I/O ports to the digital system being developed. An IOBinder specifies a behavior for driving or interpreting an I/O port of the digital system. Thus, each simulation or implementation flow specifies its own set of IOBinders to control how the digital I/Os will be interpreted in that flow. This abstracts I/O management from the actual implementation of the digital system. HarnessBinders perform a similar task from the perspective of the test harness driving the generated SoC. HarnessBinders control the generation of hardware in the TestHarness. For example, software RTL simulation requires generated hardware to drive the I/Os of the SoC under test. In contrast, FPGA prototyping or test-chip bring-up drives these I/Os using different generated hardware which interfaces with the FPGA-shell, FPGA IP and FPGA I/Os. Therefore, HarnessBinders assist in the seemless transition between simulation, evaluation and implementation platforms. Figure 5.2 illustrates the use of different combinations of IOBinders and HarnessBinders within Chipyard configurations for use across different design flows and platforms while maintaining the same digital SoC design.

Software Management

In order to enable complete SoC design and customization, software testing is treated as a first-class component within the Chipyard framework. As such, Chipyard provides a compatible set of software tools for development and testing. Chipyard provides a versioned set of standard RISC-V software development tools (e.g. GNU toolchain, QEMU, Spike ISA Simulator), as well as a set of equivalent non-standard RISC-V development tools for non-standard extensions of custom IP blocks. The two software development tool sets can be used interchangeably in the framework. Chipyard provides additional support for bare-metal software testing by using a minimalistic port of `libgloss` [178] for RISC-V Machine-mode. This port enables testing of bare-metal systems by implementing system calls through a Chipyard-compatible Host-Target interface for tethered systems.

Chipyard enables shared software development and management of complex software workloads through the FireMarshal software workload-generation tool [211]. FireMarshal provides a standard version-controlled format for software workload descriptions and automates the generation of these workloads for various simulation targets (e.g. Spike, QEMU, FireSim). FireMarshal is especially beneficial for Linux-based software workloads, where it makes the complex task of software development and porting easily reproducible and reusable by anyone on the design team without requiring special expertise.

Using FireMarshal, software developers can begin work as soon as a functional model is available (e.g. in the Spike RISC-V ISA simulator or the QEMU emulator). Those workloads can then be used without modification in RTL simulations and FireSim simulations. In this way, the complex task of software development and porting (particularly for Linux-based workloads) can be managed through re-use and portability by anyone on the design team without requiring special expertise. FireMarshal includes several examples and templates for Linux-based workloads, enabling fast ramp-up of software development across the various simulation and emulation targets using preset Linux kernel configurations and base distribution images with matching drivers. The software development structure in Chipyard is illustrated in Figure 5.3.

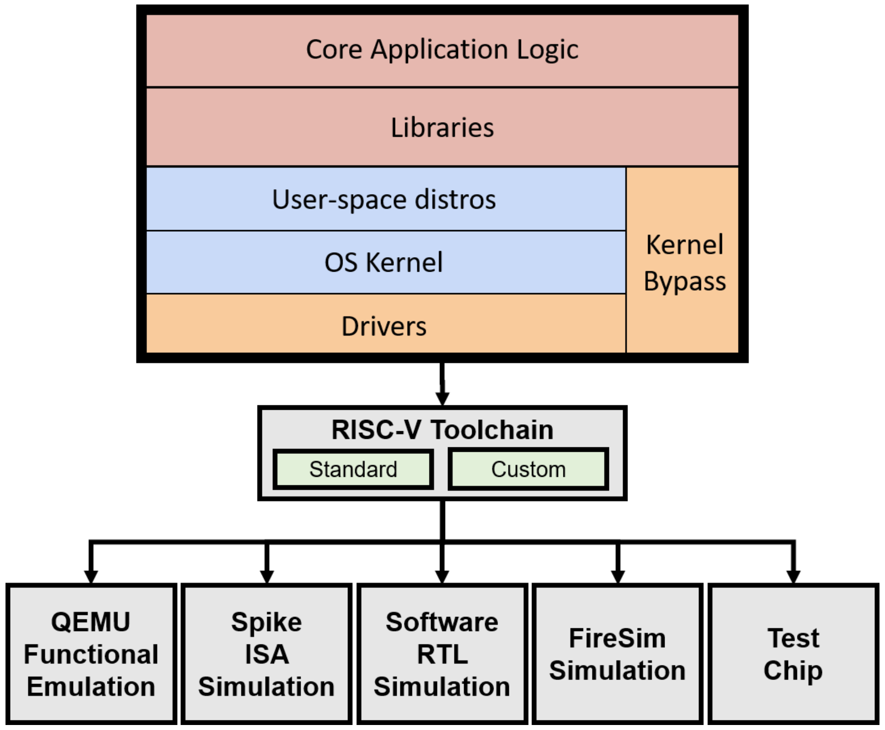


Figure 5.3: Shared software development in Chipyard using FireMarshal. Designers can use the standard RISC-V software toolchains, or custom software toolchains. While the core application logic and libraries are consistent with the SoC design, kernel and driver configuration may change based on the target platforms or tethered systems. FireMarshal automates workload generation to enable targeting multiple platforms using a single description.

5.1 SoC Customization in Chipyard

Inter-core Configurability and Composition

Chipyard provides extensive inter-core configurability and composition capabilities. Through the Rocket Chip generator processor tile interface, Chipyard can enable composition of multiple processor core options. These include various configurations of the Rocket in-order core, various configurations of the BOOM out-of-order core, the CVA6 in-order core, the Sodor educational cores, and any other core implementation that can integrate with the Rocket Chip processor tile interface. As such, a Chipyard SoC can include multiple cores for different purposes, for example containing a powerful application core together with an efficient specialized core or a collection of small low-power control cores.

All processor cores communicate with the memory system, platform, and peripherals through TileLink buses. The default SoC configuration uses a crossbar for communication between the processor tiles and the coherent memory system. Chipyard includes additional interconnect and bus structure options such as a bi-directional ring and other custom topologies. Chipyard SoCs are designed to support coherent Tilelink-based memory systems. A typical Chipyard system will use the shared L2 cache as the coherence management level. However, systems which do not wish to have such a cache can use a Tilelink broadcast hub as a coherence agent between the system bus and the memory bus. The broadcast hub will generate coherence probes for every memory request.

The diversity of Chipyard inter-core configuration has been demonstrated through highly heterogeneous test-chips [100], large homogeneous multi-core architectures [34], and as embedded controller systems for large accelerators [98].

Intra-core Configuration

Chipyard inherits the intra-core customizability of both the Rocket in-order core and the BOOM out-of-order core. As such, it can be configured to various subsets of the RISC-V ISA. These include both RV32 and RV64, as well as the M, A, C, F, D extensions individually. Further, both the Rocket and BOOM cores enable tight control over the parameters of micro-architectural structures such as the branch target buffer and branch predictor components, the organization of the L1 data and instruction caches, the depth of various queues and scheduling trackers, and the inclusion of various functional units.

For example, the BOOM core defines several default intra-core configurations named “SmallBoom”, “MediumBoom”, “LargeBoom”, “MegaBoom”, “GigaBoom”, etc. for 1-, 2-, 3-, 4-, and 5-wide configurations respectively. While these configurations are generally defined by their superscalar instruction issue width, they also differ in the types of branch predictors they include, their ROB sizes, number of physical registers, the number of entries in various issue queues and load/store queues, their L1 caches organization, system bus widths. and many other parameters. These parameters can all be used to generate additional configurations of out-of-order cores based on application requirements (very wide issue width, small branch predictors, large speculation windows, etc.).

Similarly, the Rocket Chip generator also defines several default intra-core configurations (with a similar “tiny”, “small”, “medium”, “big” naming scheme). While these configurations primarily differ in the organization and size of the L1 data and instructions caches, the largest configuration spatially unrolls the multiplication functional unit, while the smaller configurations omit the floating-point unit and branch target buffer. As the configurations get smaller, support for virtual memory is also removed, and the tiniest configuration supports only 32-bit ISA rather than 64-bit. While these default configurations provide coarse grained examples of the range of intra-core configurations within the Rocket-core, each can be further tuned based on the role of the specific core within the SoC as an application core, control core, support core, efficiency core, or any other role.

Previous Rocket-chip-based test-chips [229, 230], have included both application-core configurations of the Rocket core as well as control-core (no floating-point support, not virtual memory) configurations of the Rocket core.

Accelerators

Specialized hardware accelerators are a common building block of custom SoCs. Hardware accelerators are designed to execute a particular function or set of functions more efficiently than a general-purpose processor through specialized datapaths, specialized memory system, and reduced control-flow.

Specialized hardware accelerators can communicate with host processors in various methods, which can be analogized to "distance" - i.e. "far" peripheral accelerators and "close" co-processors. The term accelerator is often used as an umbrella term for a broad set of custom processors and digital blocks. These can be integrated with a micro-processor system as peripheral subsystems or as co-processors. While the terms "accelerator" and "co-processor" may often be used interchangeably (including in this dissertation), they generally differ in their degree of independent operation and their methods of communication with the host processor.

A co-processor is typically dispatched instructions by the host processor and may be connected to and use some of the internal resources of the host processor (for example, caches, virtual memory system, floating-point unit, etc.). In contrast, an accelerator will typically be viewed as a more independent subsystem that communicates with the host processor as a memory mapped I/O device and that has its own control logic.

Accelerators will typically be designed only for long latency operations due to the overheads of independent control logic and memory mapped I/O. The host processor will typically transfer data to the accelerator, issue a single execution command through memory mapped I/O, and then collect the results after an interrupt or polling loop indicates the accelerator is done. Co-processors can interact with the host processor in a variety of ways: they can be designed to act as an additional functional unit which communicates through registers, completes in a small number of cycles and stalls the host processor pipeline, or they can be designed to run in parallel with the host processor, taking a longer number of cycles and accessing data independently from the host processor.

Historically, the most well-known example of a co-processor was the floating-point unit (also known as "math co-processor"). A floating-point unit is typically dispatched instructions by the host-processor, can take several cycles to execute, and may operate in parallel to the host processor or stall the host processor pipeline. The Intel 8087 math co-processor was directly connected to the address and data buses, acting on instructions with specific prefix encoding. More recent examples of accelerators and co-processors which appear to include similar functional units include the Apple Neural Engine (ANE) and the Apple Matrix co-processor (AMX). Both are reported to perform matrix operations, but the Apple Neural Engine is reported to be an independent subsystem within the SoC with its own control cores and multiple execution cores capable of executing complete DNN matrix operations. At the same time, the Apple Matrix co-processor is reported to be programmed by custom instructions on the application processor, executing smaller matrix operations [82, 200].

The majority of accelerators included within Chipyard SoCs are actually co-processors rather than accelerators. The extensibility of RISC-V, through reserved major opcode en-

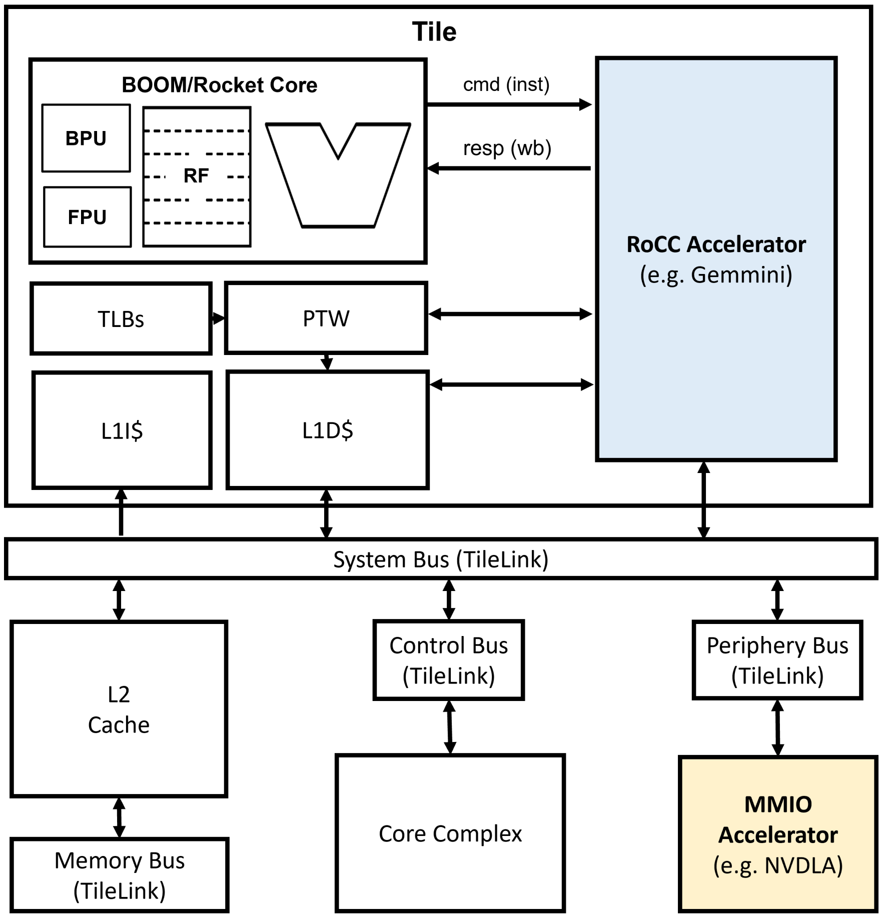


Figure 5.4: Memory-mapped peripheral accelerators and co-processor accelerators in a Chipyard SoC. Co-processors are integrated "close" to the core, using the Rocket custom co-processor interface (RoCC). Memory-mapped peripheral accelerators are integrated through the system bus and can be shared across multiple tiles in the SoC.

coding space for custom instructions, lends itself nicely to co-processor-based acceleration in custom SoCs. Co-processors are easy to program and can perform a wide range of acceleration operations due to their low latency and integration with the resources of the host CPU. Nevertheless, this integration with the host CPU may not fit all types of acceleration applications. Therefore, Chipyard supports both methods of hardware acceleration integration. Figure 5.4 illustrates the integration of co-processors and memory-mapped peripheral accelerators in a Chipyard SoC. Co-processors use the Rocket custom co-processor interface (RoCC), while larger accelerators are integrated farther from the core, on the periphery bus, using memory-mapped I/O.

Memory-Mapped Peripheral Accelerators

Memory-mapped peripherals are a common method for custom accelerator integration in SoCs. In a memory-mapped peripheral, the processor communicates with the accelerator through memory-mapped registers using the TileLink bus protocol. In Chipyard, peripheral accelerators are connected to the SoC through a periphery bus and programmed using memory mapped I/O (MMIO).

Memory-mapped accelerators are relatively generally accessible from an SoC perspective in the sense they do not require particular support or interface implementation from the application processor cores. This level of decoupling allows for the flexibility of integrating and sharing these accelerators under a variety of multi-core SoC configurations, but incurs software access costs of interrupts and memory operations when using them. Since memory mapped peripherals require fewer assumptions about the CPU than RoCC accelerators, MMIO is the preferred method of integration for third-party accelerators and larger independent subsystems in Chipyard. For example, the NVIDIA Deep Learning Accelerator (NVDLA) [241] is a third-party accelerator (written in Verilog) that is integrated within the Chipyard framework using a wrapper connecting it to the configuration system, memory map, and device tree generation mechanism. The NVDLA has been used with the Rocket Chip generator in several projects [86] and is also integrated as part of the Chipyard framework.

The integration of MMIO accelerators requires a modification of the SoC memory map. Different combinations of peripheral accelerators can result in different SoC memory maps, and therefore the generator system is very helpful in managing these varying memory maps. The Rocket Chip generator outputs a memory map based on the specific generated configuration in the form of a `json` file. Each entry in the memory map includes the name, base address, size of the segment, and permissions. This `json` file can then be processed into relevant header files for various device drivers. The Rocket Chip generator also outputs a device tree file which can be consumed by firmware and OS kernels to explore the devices generated in the SoC. The generated header files, memory maps and device trees help ease the software burden through automation of software header files for drivers and other low-level software elements which implement the communication between the processor and the accelerator.

Memory-mapped peripheral accelerators are a shared resource which is shared by all the processor cores in the system. As such, they are often managed by an operating system or similar resource-management runtime. They require software components in the form of device drivers and are located further down the memory hierarchy from the processor core. For efficient operation, a single command to the accelerator needs to amortize the cost of the device drivers and system calls required in order to access the accelerator.

Rocket Co-Processor (RoCC) Interface

The Rocket Co-Processor interface is a protocol definition and implementation that was created together with the Rocket chip generator and the Rocket in-order core [22]. It has since also been implemented with the BOOM out-of-order core. We describe the RoCC interface as background to help explain its role as part of the customization process of accelerators within Chipyard-based SoCs, and specifically the Hwacha and Gemmini accelerators that will be used for numerical data analysis SoCs in Chapters 6, 7, and 8. The RoCC protocol enables tighter integration of accelerator with the processor core compared to peripheral MMIO accelerators. Cores that support the RoCC interface communicate with the co-processor through a custom protocol and custom non-standard RISC-V instructions reserved in the RISC-V ISA encoding space. The RoCC protocol enables RoCC accelerators to access the L1 data caches, stall the processor pipeline, and pass values through registers. Each core can have up to four RoCC accelerators controlled by custom instructions and sharing resources with the CPU. RoCC has been used to implement research accelerators such as a Java garbage collection accelerator [182] and a memcpy accelerator [187], as well as accelerators which are included within the Chipyard framework such as the Hwacha vector accelerator and the Gemmini deep-learning accelerator.

RoCC defines a simple interface between a CPU core and a co-processor to support memory system access, pipeline stalls and communication of register values, as illustrated in Figure 5.5. The interface from the CPU core to the co-processors includes:

- command (`cmd`) - includes the instruction value, two operand-register content values from the CPU register file, and `mstatus` control register value from the CPU.
- exception signal (boolean)

The interface from the co-processor to the CPU includes:

- response (`resp`) - includes destination register identifier, and destination register content value to be written back to the CPU register file.
- busy signal (boolean)
- interrupt signal (boolean)

The RoCC implementation includes several generator ports to the CPU memory system. In particular, a port to the CPU L1 cache, a port to the CPU page table walker (PTW), and a port to CPU floating-point unit (FPU). RoCC accelerator implementations can also be wired to other levels of the memory hierarchy through TileLink buses. For example, the Hwacha vector accelerator and the Gemmini accelerator are connected to the L2 caches through the Tilelink system bus, as opposed to the L1 cache. This is since both these accelerators have large vector register files or scratchpad memories which are bigger than the L1 cache. Direct interfaces to the SoC's L2 cache enable shared communication with the memory system and other tiles in the system without polluting the core's L1 cache.

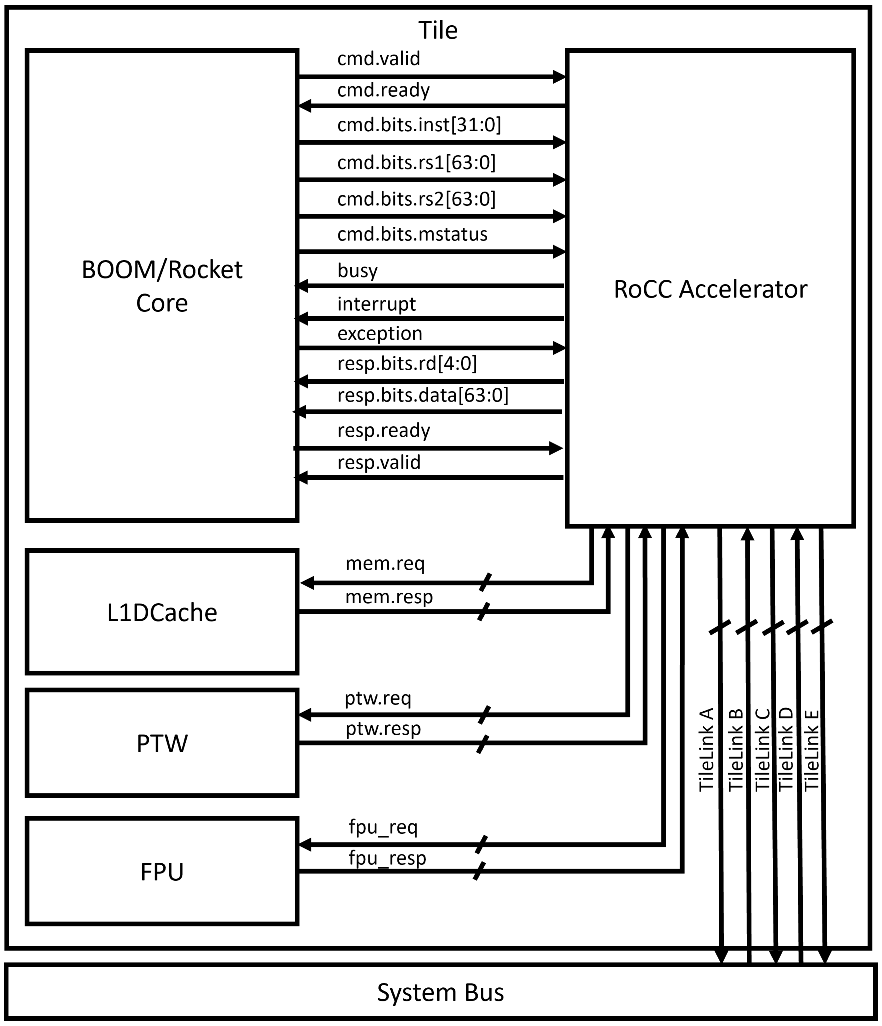


Figure 5.5: The RoCC interface. Signals communicating custom instructions and results between the core and accelerator, as well as additional interface signals to the core's L1 data cache, page table walker, and floating-point unit. Alternatively, the accelerator can communicate with the rest of the SoC memory system through the SoC TileLink system bus.

There are several considerations when choosing RoCC as an accelerator interface as opposed to a peripheral alternative. In some sense, a RoCC accelerator can be thought of almost like an additional functional unit within the CPU, albeit with several significant limitations. RoCC accelerators are programmed using custom instructions which are part of the user-space instruction stream. This makes RoCC accelerators very easy to program for simple single-threaded programs, as there is no requirement for device drivers or interaction with different protection rings. RoCC accelerators are associated with a single core (or single hardware thread), and therefore resource sharing consideration must be handled by the accelerator itself rather than the operating system. These include isolation in multiprocessing systems, state save/restore upon context switches, and adherence with memory system coherence. Due to its "close" integration with the CPU, RoCC can provide a higher level of support for virtual memory than can peripheral accelerators. Nevertheless, this support for virtual memory has its limitations due to the simplistic definition of the RoCC interface. The single exception signal means that RoCC does not provide an exception vector with additional information about the type of exceptions generated by the co-processor. For example, while both the NVDLA and Gemmini are designed to accelerate deep learning workloads, Gemmini, unlike the NVDLA, has DMA access to the SoC's shared L2 cache, can stall the host processor pipeline, and can be programmed directly from user-space with custom assembly instructions.

The RoCC interface can be considered a specific example of intra-core customizability in the Chipyard framework. In fact, each processor tile can be attached to different RoCC units.

Peripherals

Chipyard SoCs can includes a variety of standard peripheral blocks integrated through implementations from open-source repositories. These include complete peripheral devices such as JTAG, UART, GPIO SPI, I^2C, PWM and so on which have been used in multiple fabricated test-chips, as well as simulated stubs for more complex peripheral devices such as network interface controllers (NIC) device and block devices which enable system simulation. As is the case with peripheral accelerators, these traditional peripheral blocks also contribute to generated header files, memory maps and device trees.

For example, IceNIC is a peripheral network interface subsystem which implements the digital components of an Ethernet NIC. The automatically generated device tree allows simple integration with the Linux device driver across a variety of SoC configurations with different memory maps.

5.2 SoC Design Frameworks

SoC development has been a cornerstone of the mobile computing industry for more than a decade. With the demise of Dennard scaling and the rise of dark silicon, integration at the SoC level is one of the primary vehicles for continuing to improve system performance while maintaining low power and energy budgets. Custom SoCs are increasingly adopted in a wider variety of computing platforms from general purpose personal computing [132] to custom data-centers [262]. However, their design comes with challenges across all stages of the chip design process.

In recent years, several SoC design frameworks for agile hardware development have been developed in the academic community and the open-source hardware community. These include the OpenPiton platform [26, 25], the ESP framework [186, 43], the ChipKit platform [275], the BlackParrot [212] and HammerBlade projects, and the Efabless open-source fabrication shuttles [63]. While at first glance these SoC frameworks may seem as simple repositories of digital IP blocks, their primary value lays in their development and implementation flows with respect to their IP, as opposed to the simply providing the source code for the IP. SoC design requires development of complete verification, validation, implementation and software flows in order to be useful for custom SoC applications.

Due to the immense scope of expertise and design flows involved in SoC development, the different SoC design frameworks tend to focus on different approaches to achieving SoC customizability. Some frameworks, such as OpenPiton and ESP, focus on the scalability of the SoC into many-core architectures using tile-based SoC templates and mesh NoCs. Other frameworks such as ChipKit and Efabless assume integration of custom IP through standard bus interfaces and therefore focus on the VLSI implementation aspects of the SoC, under the assumption that the core digital system IP for the SoC will be provided by commercial vendors (ARM, in the case of ChipKit) or other projects in the open-source community (in the case of Efabless).

The frameworks also differ in their approaches to software development and verification. FPGA prototyping is integrated as a native method for software development in frameworks such as ESP, BlackParrot, OpenPiton. While Chipyard provides support for FPGA prototyping, its design flow directs software development to use FPGA-based emulation rather than prototyping in order to provide debuggable deterministic designs and timing accurate memory behavior. Support also varies in terms of simulation vs. VLSI implementation flows. While the Efabless Caravel SoC harness provides a relatively detailed VLSI flow based on the OpenRoad open-source toolset, other frameworks may integrate with commercial reference flows through patches or automated script generation. Some frameworks which primarily target simulation and prototyping may not have VLSI flow integration at all.

In comparison to alternative academic and open-source SoC design frameworks, Chipyard provides a comprehensive solution in terms of support for IP extensibility and customization, software development, and VLSI implementation. Through the generator-based IP and automated multi-flow transformations, Chipyard can match the abilities of similar SoC design frameworks, and exceed them in certain cases. While there are cases in which other frameworks provide better solutions in terms of scalability, readability, documentation, and integration with standard industry tools, Chipyard's high degree of automation and configurability often provides a more productive and agile environment for a wider range of use-cases. The diversity of RISC-V cores integrated with the Chipyard framework in comparison to other SoC frameworks, together with its automation and documentation, make it a prime platform of choice for comparisons between RISC-V application cores [80]. A summary of the Chipyard framework in comparison to alternative SoC design platforms (as of the time of writing) is presented in Table 5.1

5.3 Agile Hardware Development Using Chipyard

To demonstrate the agile framework in action, we take an example baseline Chipyard SoC configuration and iteratively apply changes throughout the development and customization process. We will further examine and validate various properties relevant to those changes across the customization process. The example baseline Chipyard SoC includes a single BOOM out-of-order application-class processor, a shared L2 cache, and mix of UART, TSI and GPIO peripheral interfaces. This example baseline SoC and its associated Chipyard configuration is shown in Figure 5.6. The BOOM L1 Data and Instruction caches communicate with the rest of the SoC memory system and periphery through a TileLink crossbar (typically called the "system bus"). The system bus communicates with the L2 cache, which then communicates with the backing memory system (not drawn for simplification). Peripheral devices such as UART and GPIO communicate with the system bus through an additional bus level called the "periphery bus" (not drawn for simplification). This separation of buses allows the periphery bus to run at different frequency and in different power domains compared to the rest of the system. The boot ROM and other components of the core complex such as the platform local interrupt controller (PLIC) and core local interrupt controller (CLINT) communicate with the system bus through an additional bus level called the "control bus" (not drawn for simplification). The SoC configuration is a composition of several other configuration *fragments* of system sub-components. For example, a BOOM `WithMegaBooms` configuration fragment specifies a BOOM configuration with a decode pipeline width of 4 instructions, 3-wide integer issue, 2-wide floating-point issue, 2-wide memory issue, 128 ROB entries and 128 physical registers, and many other additional configuration parameters.

Adding an Accelerator

With the slowdown of Moore's law, many research endeavours begin with implementing a specialized compute accelerator for particular applications. Chipyard offers multiple methods of accelerator integration through the Rocket Chip generator with varying degrees of coupling to the host processor. As noted in Section 5.1, these methods include memory-mapped peripheral accelerators and Rocket Custom Co-processors (RoCC).

All accelerator integration methods within Chipyard fit within the Chipyard configuration system and allow for easy addition and removal from the SoC. As a result, the Chipyard generator repository includes multiple open-source accelerator IPs that can be added to the SoC using a single line of code within the SoC configuration. For example, Figure 5.7 demon-

	Property	Chipyard [13, 12]	OpenPiton [26, 25]	ESP [186, 43]	ChipKit [275]	Black-Parrot [212]	Efabless [63]
Design	Cores	Rocket [22], Boom [288] , CVA6 [285], Sodor	Open-SPARC T1, CVA6 [285], PicoRV32, ao486 (x86)	CVA6 [285], Ibex, Leon3 (SPARC)	ARM A53	Black-Parrot	PicoRV32
	Cache Coherence	TileLink	TRI	✗	AMBA	BedRock	✗
	Interconnect Topology	Xbar, Ring NoC, Custom NoC	Mesh NoC	Mesh NoC	Xbar	Mesh NoC	Shared Bus
	HDL	Chisel, System-Verilog	System-Verilog	System-Verilog	System-Verilog	System-Verilog	System-Verilog
	Open Source	✓	✓	✓	✗	✓	✓
Customization	Intra-Core	Generator	✗	✗	✗	Tile Modularity	✗
	Inter-Core	Generator	Tile Modularity	Tile Modularity	Monolithic	Tile Modularity	Monolithic
	Accelerator Integration	RoCC, MMIO	Tile Interface	Tile Interface	AHB Bus	Tile Interface	Wishbone Bus
Development	SW RTL Simulation	VCS, Verilator	VCS, ModelSim, Verilator	NCSim, ModelSim	✗	VCS, Verilator	✗
	FPGA Emulation	FireSim [34]	✗	✗	✗	✗	✗
	FPGA Prototyping	✓	✓	✓	✗	✓	✗
Artifact	VLSI Flow	Hammer [269]	✓	✗	✓	✗	OpenLane [235]
	OS Support	✓	✓	✓	✓	✓	✗

Table 5.1: Comparison of non-commercial SoC design frameworks.

```
class BaselineConfig extends Config(
  new chipyard.iobinders.WithUARTAdapter     ++
  new chipyard.iobinders.WithBlackBoxSimMem ++
  new chipyard.iobinders.WithSimSerial      ++
  new testchipip.WithTSI                    ++
  new chipyard.config.WithBootROM           ++
  new chipyard.config.WithUART              ++
  new boom.common.WithMegaBooms             ++
  new boom.common.WithNBoomCores(1)         ++
  new freechips.rocketchip.system.BaseConfig)
```

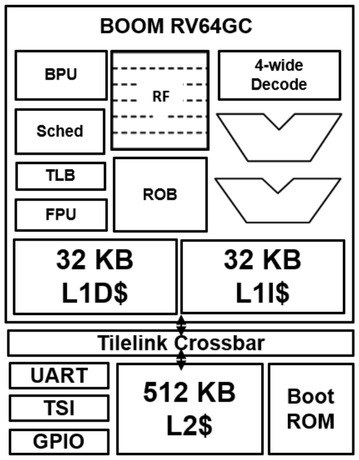


Figure 5.6: An example baseline SoC configuration consisting of a single 4-wide BOOM out-of-order application-class processor, a shared L2 cache, and mix of UART, TSI and GPIO peripheral interfaces. [12] (© 2020 IEEE).

```
class BaselineGemminiConfig extends Config(
  new chipyard.iobinders.WithUARTAdapter     ++
  new chipyard.iobinders.WithBlackBoxSimMem ++
  new chipyard.iobinders.WithSimSerial      ++
  new testchipip.WithTSI                    ++
  new chipyard.config.WithBootROM           ++
  new chipyard.config.WithUART              ++
  new gemmini.DefaultGemminiConfig          ++
  new boom.common.WithMegaBooms             ++
  new boom.common.WithNBoomCores(1)         ++
  new freechips.rocketchip.system.BaseConfig)
```

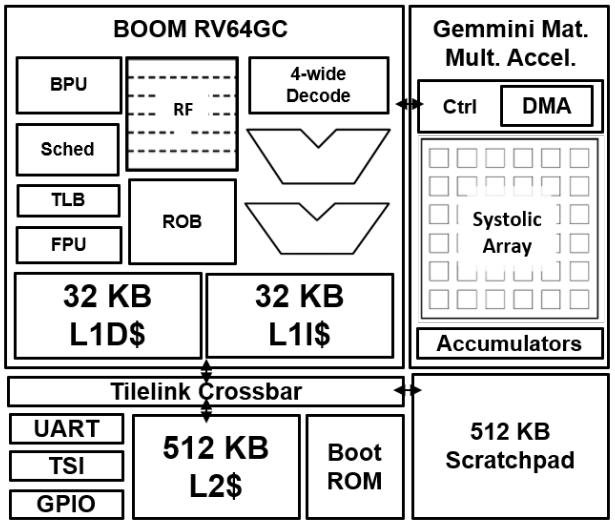


Figure 5.7: An SoC configuration adding a Gemmini machine learning accelerator to the baseline SoC config using a single configuration line (highlighted). [12] (© 2020 IEEE).

strates adding the Gemmini accelerator (which uses the RoCC interface) to the baseline SoC. This is done by adding the `DefaultGemminiConfig` fragment to the SoC configuration. the `DefaultGemminiConfig` fragment class is defined within the Gemmini project repository, and sets the various accelerator parameters such as the systolic array size (16x16), scratchpad size (512 KiB), accumulator sizes (64 KiB), datatypes (Int8), dataflows (WS), etc.

Accelerator Software Validation

Pre-silicon validation of software which uses the custom accelerator helps shorten the overall system development cycle and identify functional bugs and performance pathologies when changes can still be made.

For the SoC configuration from the previous section, we want to evaluate the execution of DNN inference using the accelerator within the SoC. Executing the inference of a batch of 4 images using the standard ResNet-50 DNN model takes 4 billion cycles. Running such a software RTL simulation would take several days. The relevant testing and validation flow within the Chipyard framework uses the Spike functional ISA simulator and the FireSim FPGA-accelerated simulation platform. A functional model of the Gemmini accelerator is integrated into a non-standard Spike functional simulator, which enables initial software development with the Gemmini custom instruction extensions. Once functional results are satisfactory, further performance tuning is performed using the FireSim FPGA-accelerated simulation platform. Executing the 4 billion cycles of ResNet-50 on Gemmini with FireSim takes a mere few seconds of wall-clock time. While initial FPGA simulation synthesis and build time is longer than standard RTL software simulation compilation time, the combined synthesis and build time is significantly shorter than software simulation time. The one-off build synthesis time overhead is further amortized over a large number of simulation runs when used for pre-silicon software performance optimization on the simulated custom SoC.

In fact, this flow of complete validation of software using custom accelerators on the Chipyard framework has become sufficiently easy-to-use that it can also be used for accelerator evaluation under educational settings. The aforementioned flow for hardware-software performance optimization of a Gemmini accelerator was used in a class-wide lab in a recent course offered at UC Berkeley [10].

Core Power-Performance-Area Tradeoffs

SoC designers can choose amongst a variety of application cores to anchor an SoC. A common power-performance-area (PPA) consideration when designing a custom SoC is whether to use smaller and more energy efficient parallel cores vs. larger and more power-consuming high-performance single-thread cores. While a particular application amenable to parallel processing might benefit from a multi-core configuration, Amdahl's law reminds us that single-threaded performance limits the potential overall performance gains from parallelism.

Thus, determining the right core configuration requires careful consideration of the expected software workload of the device. A SoC designer might choose to explore multiple core configurations before picking a design point. The Chipyard framework does not "lock-in" any particular core configuration or core count, as the framework can generate reasonable single-core and multi-core designs, with core configurations ranging from small embedded-class in-order cores to large application-class out-of-order cores.

For example, after careful analysis of the software workloads meant to run on the target SoC in Figure 5.7, the designer may want to explore the PPA tradeoff of a more parallel

```
class QuadRocketGemminiConfig extends Config(
  new chipyard.iobinders.WithUARTAdapter               ++
  new chipyard.iobinders.WithBlackBoxSimMem            ++
  new chipyard.iobinders.WithSimSerial                 ++
  new testchipip.WithTSI                               ++
  new chipyard.config.WithBootROM                      ++
  new chipyard.config.WithUART                         ++
  new gemmini.SmallGemminiConfig                       ++
  new freechips.rocketchip.subsystem.WithNBigCores(4)++
  new freechips.rocketchip.system.BaseConfig)
```

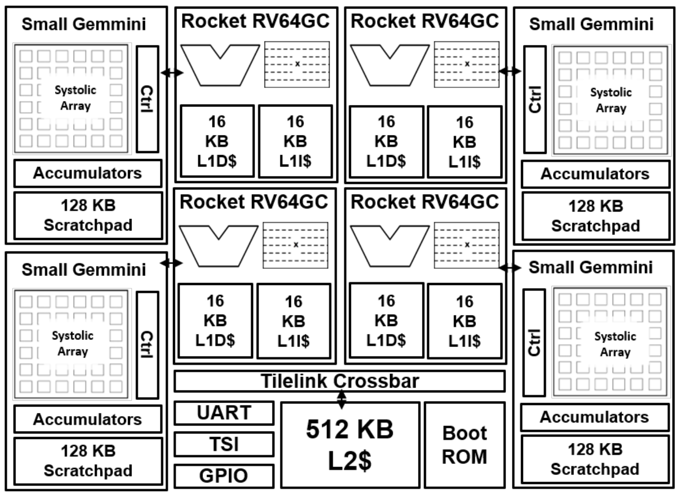


Figure 5.8: An SoC configuration replacing the single BOOM out-of-order core with 4 smaller Rocket in-order cores and smaller Gemmini accelerators to evaluate PPA tradeoffs. The highlighted lines represent the configuration lines that were replaced. [12] (© 2020 IEEE).

design-point which trades off the powerful single-thread performance of the BOOM out-of-order core for multiple smaller, but more efficient Rocket in-order cores with smaller Gemmini accelerators. With just a two-line change, the designer can reconfigure Chipyard to generate this vastly different SoC architecture (illustrated in Figure 5.8) with four smaller in-order cores each driving a smaller 8x8 Gemmini accelerator. The designer can then take advantage of Chipyard simulation and VLSI flows to quickly measure the power-performance tradeoffs of either design. Figure 5.9 demonstrates such an area tradeoff comparison using a commercial FinFET process and the Hammer flow as part of the SoC design process. Similar estimations can be performed using open-source PDKs.

Non-invasive Physical Design Optimization

Throughout the implementation process, constraints imposed by the physical and computational realities of VLSI processes many times result in divergences from the original architectural design. Chipyard attempts to address the design divergences that appear between digital/architectural simulation and silicon implementation, through a "source-of-truth" RTL generator with re-usable and customizable RTL passes that transform the RTL for VLSI flows. It is important that the source hardware description simulated during architectural exploration is as close as possible to the hardware description that goes through physical design and implementation.

Instead of performing a flat layout in which the entire SoC design is ran through the VLSI EDA tools at once, many SoCs instead use a hierarchical flow. In a hierarchical physical design flow, the SoC is split into smaller subcomponents which are later assembled together,

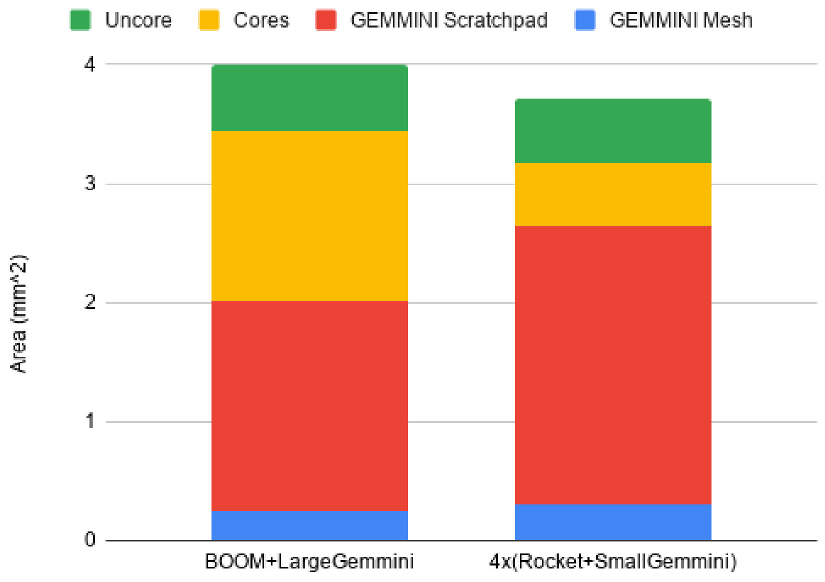


Figure 5.9: Post-synthesis area estimates using a commercial FinFET process comparing a powerful single-core system with an alternative equivalent parallel multi-core system. [12] (© 2020 IEEE).

so VLSI EDA tools can better optimize the smaller sub-blocks instead of attempting to find a global layout optimum across the entire SoC. When combined with floorplanning to get better timing and reduce wire congestion, the physical module hierarchy can differ significantly from the logical hierarchy described in the source RTL. However, directly modifying the source RTL to move and group modules to fit the new hierarchy is both time-intensive and error-prone. Leveraging the power of FIRRTL, a set of FIRRTL passes can be applied to the design to quickly make large-scale hierarchy changes while maintaining correct functionality. This is illustrated in Figure 5.10, in which the prior design in Figure 5.8 was transformed into four module regions: two clusters that both include two Rocket cores and two Gemmini accelerators, an I/O region that contains UART, TSI, and GPIO interfaces, and an uncore region that contains items like the L2 cache and bus subsystems. This is done with a native FIRRTL transformation called `GroupAndDedup` which groups modules together, followed by the `InlineInstances` FIRRTL transformation to split up a module into its submodules. This new hierarchy is used within a hierarchical physical design flow which synthesizes, places and routes each of the components individually and then assembles them together. As a result, the overall time of a placement and routing iteration is reduced from hundreds of hours to tens of hours.

By using these mainstream FIRRTL passes, the main Chisel RTL design is unchanged from the one that was originally tested and verified, but the design can now continue iterating through more efficient VLSI flow quality-of-results (QoR) optimizations. This enables shortening of design-cycle iterations through overlapping simulation, testing and physical

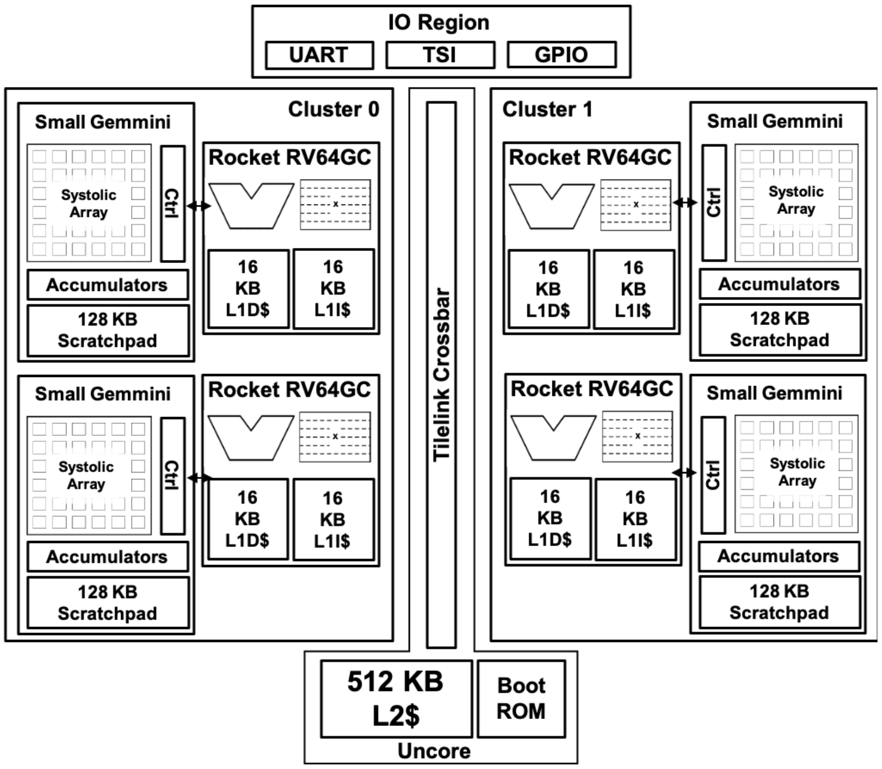


Figure 5.10: Physical-design friendly hierarchy generated after FIRRTL transformations, using the same SoC configuration from Figure 5.8. [12] (© 2020 IEEE).

design.

5.4 Accessibility and Education

Open Source SoC Development

The RISC-V ISA is a free and open ISA specification originally designed at UC Berkeley for research and education purposes. It has garnered industrial and academic momentum in helping lead the open-source hardware movement. The rapid adoption of RISC-V in the open-source community is evolving with a strong open-source software ecosystem, as well as accessible open-source processor implementations. The emergence of both open-source hardware and software ecosystems makes RISC-V an excellent tool for research [94, 93, 104], education [1, 268, 141], and commercial purposes [168, 139, 135].

The RISC-V ISA specification has enabled the emergence of many open-source processor implementations that conform to various subsets of the ISA specification - from simple microcontrollers [94, 226] to complete application-class processors with Linux support [22, 47, 285, 29]. These open-source processor implementations provide students, researchers, and developers unprecedented access and visibility to the nuances behind practical processor implementations. Chipyard is part of this open-source ecosystem which increases access to digital implementation and modeling of complex SoCs. The Chipyard mailing list [53] provides a window into an open-source community of users from across the world representing both academia and emerging start-ups.

Chipyard aims to be a fully open-source SoC development framework. However, while open-source tools and projects within the digital-logic abstraction layer of the hardware design stack have been gaining traction and trust in research and industrial communities, the manufacturing and circuit-level aspects of hardware development are still struggling to identify appropriate open-source development and integration models. This struggle encompasses both the digital and analog realms of circuit development.

In the analog domain, new generator-based approaches such as the Berkeley Analog Generator (BAG) [48] are envisioning a portable and process-technology agnostic generator-based approach to analog design. However, this approach cannot stand alone without proper integration with digital SoC components. The analog-digital divide is perpetuated by distinct tools and methodologies, despite the common goal of silicon development. Ongoing integration efforts of analog generators such as BAG into the Chipyard framework will help support common interfaces and integration between analog and digital components, through integrated tools and automation of design collateral such as appropriate generated behavioral models of analog blocks, as well as matching physical design constraints. We believe that integration of tools is the best way to break the rigid silo barriers that currently exist.

In the digital domain, the manufacturing and circuit-level aspects of hardware design are still tightly entangled with various legal agreements and property disputes. These issues are exacerbated by the tight integration of modern process technologies' process design kits (PDKs) with the standard commercial EDA tool stack due to the complexities of sub-micron process technologies. In our view, this problem, which has been identified in the past [106], is the elephant in the room of open-source and agile hardware development. Major academic open-source hardware projects choose to address this challenge in various ways: some publish "patches" to the common flow scripts provided by EDA vendors [26], or partially associate hardware design template implementations with particular process technologies [254], but the vast majority of projects simply choose to avoid publishing physical design collateral within the open-source ecosystem. The approach used in the Hammer VLSI flow within Chipyard is a "plug-in" model. These plug-ins provide a mapping between the Hammer vendor-agnostic level of abstraction, to the proprietary vendor specific APIs. Nevertheless, these mappings cannot be open-sourced due to some of the aforementioned agreements and property disputes. In-fact, these are the only components of the Chipyard framework which are not open-source.

Chiplet-based designs have been proposed as a potential solution to the challenges of open-source mixed-signal IP [255]. Open-source initiatives such as the OpenROAD project [4] are encouraging in their goal to address this problem in the EDA stack. Commercial support for fabrication-ready open source PDKs such as Skywater 130nm [101, 256] provide additional avenues for development. However, as process technologies continue to increase in complexity, we feel this is a challenge that requires collaboration between vendors and the open-source community to enable the use of new abstractions. These new abstractions should not impose on intellectual property rights, but enable the open source community to develop tools and methodologies which can co-exist aside the leading commercial solutions to provide viable pathways to modern silicon fabrication.

Methodology Automation

Methodologies provide important guidance in the development of SoCs. Methodologies include coding patterns and conventions, interfaces, best practices, tool flows, and much more. However, in absence of automation, methodologies may be left as high-level ideas and documents rather than the directions and conventions they were intended to implement. Automation can increase the adoption of methodologies by simplifying their practice and lowering the barrier of expertise to their implementation. Within the software development community, integrated development environments (IDEs), linters and text editor extensions have helped automate and enforce many of the methodologies, coding conventions and build flows prevalent in software engineering.

The integrated nature of the Chipyard framework provides a backbone for such automation within the context of hardware methodologies. The Chipyard framework enables automatic collateral sharing between IP generators, different tools, and different platforms across the development flow. While agile and generator-based approaches for design of SoCs based on the Rocket Chip generator have been presented in the past [170], multiple disparate tools and obfuscated development environments made it challenging to execute [3, 7]. Some proposed methodological ideas did not yet even have the tools required to support them. In other cases, ideas borrowed from software engineering were difficult to adopt in hardware engineering environments. And in some cases, the simple mismatches of tool versions or lack of documentation posed a hurdle in the implementation of these methodologies. Through automation and unification of RTL generation flows across multiple generators, and the integration of RTL generation flows with the system simulation and implementation flows, Chipyard not only builds upon previously proposed methodologies, but also automates them. The integration of the software workload-management system further helps maintain complete automation and compatibility which enables automated implementation of generator-based methodologies.

Automation enables the Chipyard framework to act as a gateway to multiple other tools and generators, that would otherwise be difficult to use as independent steps within a methodology. As such Chipyard provides a single introduction to a complete set of tools required for the execution of a generator-based SoC design methodology. This single gate helps streamline the execution of the methodology and lower the expertise barrier required to adopt it and execute it successfully.

Cloud-hosted Platforms

Hardware design has been traditionally performed by using on-premise compute resources and proprietary licensed EDA environments. As noted earlier, open-source software is helping in leading a shift in this traditional usage model. An additional aspect of usage that has emerged in recent years is the usage of public cloud resources as opposed to on-premise compute for compute-intensive hardware design workloads [232]. Major EDA vendors already offer cloud-based solutions for their major software products [133, 137], with some also offering cloud-based access to more complex emulation platforms [138, 134].

The traditional advantages of a public cloud over local compute resources include scalability, elasticity (flexibility), cost model and maintenance. The public cloud enables users to scale compute resource immediately, without the need to over-provision local resources or wait for a backlog of local compute acquisition. The public cloud provides users with the ability to pick compute resources that match a particular workload out of a wide variety of offerings (compute-optimized, memory-optimized, network-optimized, accelerators, etc.). The public cloud takes over the burden of local server farm and licensing maintenance. And finally, the public cloud's cost model trades off capital expenditures with operating costs, providing finer-grained control over spending on compute resources.

Within the context of hardware development, these properties are especially important for compute-intensive workloads such as verification and physical layout. Traditional ASIC verification flows require nightly regressions across a broad test suite. While this type of nightly regression often uses multiple compute resources across a server farm, individual developers may want to evaluate a single feature across a subset of tests. The elasticity of the public cloud can enable this by instantaneously allocating a large number of resources for a short amount of time to run a large set of tests in parallel immediately, without contention for local resources. Similarly, when a design decision needs to be made close to a tape-out deadline, multiple place-and-route jobs can be initiated concurrently using the elasticity of the public cloud in order to obtain the best result in a short amount of time, meeting the tapeout deadline, at the cost of a short burst of higher than usual spending.

From an academic and open-source hardware perspective, the public cloud provides even more advantages beyond scalability and flexibility. The use of local EDA tools and compute farms has often generated silo environments that hampered collaboration across academic and open-source hardware organization. The public cloud enables improved sharing and collaboration through common tools, compute resources (including accelerators), licenses, and environment setups. As an example of the benefits of sharing FPGA images in the public cloud, by sharing cloud FPGA images (AFIs) of a Rocket core with the RISC-V hypervisor extension on FireSim, open-source developers were able to bring-up multiple hypervisors in the RISC-V environment, including the Linux KVM [224, 208]. This common and publicly accessible infrastructure also enables better reproducability of academic work. Researchers can provide images or containers of their experimental setup in the public cloud to enable others to reproduce experiments without the typical experimental environment mismatches.

Within the Chipyard framework, the FireSim components take a "cloud-first" approach to

FPGA-accelerated simulation and emulation. In fact, while Chipyard supports prototyping and emulation using local FPGAs, it initially supported only cloud-based emulation. For other parts of the flow which require licensed tools, Chipyard attempts to provide both a commercial locally-licensed option as well as an open-source alternative that can be used in the public cloud (as is the case with software RTL simulation with Synposys VCS and open-source Verilator). While some components of Chipyard are still tied to local EDA licensing tools, it aims to eventually identify and provide complete open-source alternatives integrated within the framework (in additional to the traditional flows) in order to streamline adoption and execution using the public cloud.

Education

The study of computer architecture traditionally traverses the hardware-software interface: first, it informs programmers about the logical structure of the computing hardware executing their software, so as to explain its performance characteristics and guide the optimization of software; and second, it educates hardware engineers about the principles, techniques, and trade-offs that permeate the design and implementation of computer systems. However, the development of domain-specific computing systems in recent years requires a change in exploration of vertically integrated computing architectures and software/hardware co-design. This emphasis is reflected in explicit hardware and programming models for parallel computing through differentiated hardware architectures such as FPGAs, GPUs, and other accelerators in both edge and cloud computing platforms. While traditional computer architecture education has been able to develop analytical models and rules-of-thumb for processor performance based on the simple single-thread software model, modern computer architecture education requires a much more empirical approach and relies upon a higher degree of empirical experience and simulation.

Approaches to computer architecture education can be paralleled to the Iron Law of Processor Performance [81, 122], which factors processor performance into three components: the number of instructions executed, cycles per instruction, and cycle time. These succinctly capture the contributions from various layers of the computing stack, spanning from algorithms to circuits. While all computer architecture curricula broadly cover fundamentals of the Iron Law, individual courses tend to emphasize either a *software-centric* or *hardware-centric* perspective. A software-centric approach focuses on the interaction between the programming model and microarchitecture, associated with the first two components of the Iron Law: the number of instructions executed and the cycles per instruction. Experiments typically rely on abstract functional simulators or cycle-approximate simulators. Knowledge of RTL description languages is generally not required for most computer architecture courses. A hardware-centric approach focuses on the physical implications of computer architecture, represented by the latter two factors of the Iron Law: the cycles per instruction and the cycle time. Usually tightly entwined with the digital design curricula, this highlights the influence of technological constraints: for example, the relationship between pipelining and flip-flop setup and hold times; the considerations of SRAM organization in cache and register file design; and the limits on microarchitectural complexity due to wire congestion and power consumption. Lab assignments often involve the manipulation of RTL and associated methodologies [159, 196].

Hardware-centric approaches to computer architecture education primarily use HDL logic simulators, similar to the ones used in digital design courses. Through the use of such HDL-based simulators (e.g. ModelSim or VCS) and graphical-gate-based simulators (e.g. Logisim [41]) students will often design a basic processor pipeline with a minimal set of features. This approach does not enable students to observe phenomena that are the result of large hardware structures or long-running software workload behaviors such as branch predictions and caching policies. These hardware-centric simulators provide a high level of implementation detail and simulation granularity at the cost of system-wide view and execution time. Hardware-centric approaches may go beyond simulator usage, and choose to use FPGAs and FPGA-based soft cores to facilitate additional practical computer architecture experience [115, 166, 216]. FPGA usage may mitigate some of the execution time costs of hardware-centric simulators, but involve additional engineering effort by students, as well as resource accessibility and class scalability considerations.

Software-centric approaches often rely on performance simulators, which trade off implementation details for a simplified model of the problem domain. Simulators that are used both by the research community and for instructional purposes include complete processor models such as GEM5 [36] or component-specific trace-based models for cache behavior or branch prediction behavior such as ChampSim. However, in many cases instructors will choose to develop a home-grown simulation model designed for a particular assignment. High-level simulator usage in classes cannot be disentangled from a long-running debate in the computer architecture research community around over-reliance on quantitative evaluations from model-based performance simulators [202].

The Chipyard framework's generator-based integrated nature enables a multi-faceted approach that traverses the software-hardware interface, enabling a complete system view of processor performance and efficiency through practical exercises. In contrast to high-level synthesis (HLS) approaches, hardware generators do not raise the level of abstraction above the RTL level. The *design* of hardware generators still requires a thorough understanding of RTL-based digital design considerations. However, the *usage* of hardware generators is significantly simplified through modularity and parameter configuration schemes. Within the research and development community, these usage models enable additional re-use of building blocks towards specialized computing systems. Within the education community, these simpler usage models open up new educational opportunities for demonstrating complex system interactions through modular and parameter-based lab exercises. Similar approaches for computer architecture education have been proposed in the realm of embedded systems [245, 244] where, instead of RTL generators, Architecture Description Languages (ADLs) can be used to compose and configure processor structures at a higher level of abstraction. Unfortunately, unlike RTL, ADLs do not have tooling support to be synthesized into gate-level digital integrated circuits, and therefore their usage is still restricted to only modeling rather than implementation.

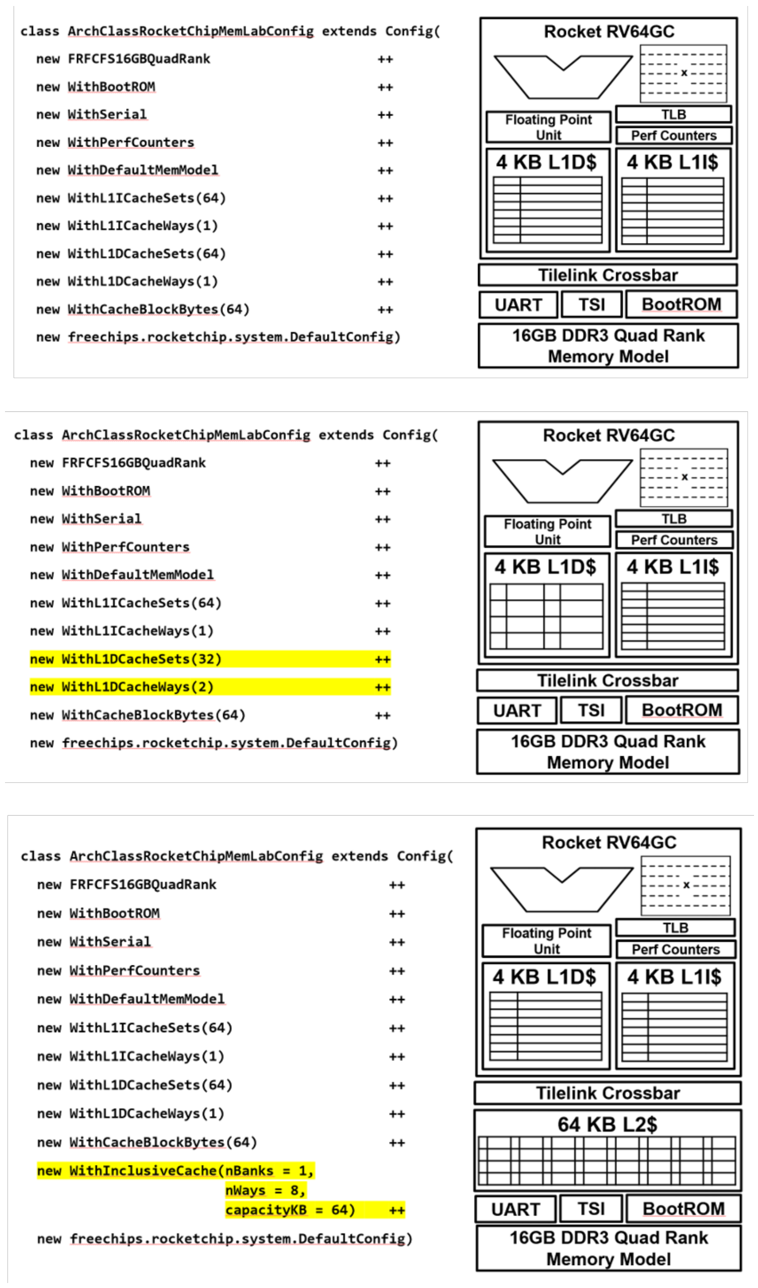
In a joint generator-based exercises approach using the Chipyard framework, students modify the configuration of an RTL hardware generator and execute RTL-level simulation or implementation flows through Chipyard's automated build process. Figure 5.11 demonstrates example configurations used in a lab exercise on the topics of memory hierarchies and caches. In this lab, students change cache parameters and memory hierarchy compositions and observe their interaction with cache-tiling of matrix operations in RTL-level simulation. The configuration interface is similar to dedicated caching simulators, but the generator emits complete SoC RTL which is compiled into a cycle-accurate simulation executing the software workload as it would be executed on actual silicon.

Prior work has surveyed many existing simulators [5, 6], with classifications based on traditional usage within the computer architecture research community as well as through an educational and pedagogical point of view [199, 116, 215], with some surveys proposing criteria for the evaluation of their suitability for teaching courses in computer architecture. Within the criteria proposed in [199], the proposed generator-based approach using Chipyard would be considered under the "Advanced Architecture (AA)" category with "Design Support." This joint model balances the trade-offs between simulation granularity and the level of implementation detail using a finer scale than those used in prior simulator surveys for education.

Vertical Curricula Span Using Chipyard

The use of the Chipyard environment has in fact allowed us to create an interconnecting thread between electrical engineering classes and computer science classes. Using this environment, students who choose to take classes across both fields are able to break through the levels of abstraction and see how different components of the hardware-software stack relate to each other by using the same environment for labs and assignments across various classes. Figure 5.12 illustrates the multi-class flow enabled by a unified generator-based system framework. Computer-science-oriented students do not need to interact with RTL, as they only change configuration files and associated software. Digital integrated circuits students get a baseline design to explore VLSI optimizations. In the overlap section between standard classes, special topics classes such as "hardware for machine learning" or "hardware for digital signal processing" can utilize the unified framework to integrate between the unique software algorithms mapping onto custom hardware architectures to demonstrate the interactions between the two with complete system design flows.

During the Spring semester of 2020, the Chipyard framework was used in three concurrent classes at UC Berkeley, as demonstrated in Figure 5.12: A computer architecture class, a digital integrated circuits design class, and a special topics class about hardware for machine learning. Approximately 15% of students in the special topics class were also enrolled in the computer architecture class in the same semester, while 10% of students in the special topics class were also enrolled in the digital integrated circuit design class in the same semester. The unified assignments framework reduced the ramp-up time for students. Those who took two classes concurrently were able to amortize their infrastructure setup learning curve



```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                     ++
  new WithBootROM                            ++
  new WithSerial                             ++
  new WithPerfCounters                       ++
  new WithDefaultMemModel                    ++
  new WithL1ICacheSets(64)                   ++
  new WithL1ICacheWays(1)                    ++
  new WithL1DCacheSets(64)                   ++
  new WithL1DCacheWays(1)                    ++
  new WithCacheBlockBytes(64)                ++
  new freechips.rocketchip.system.DefaultConfig)
```

```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                     ++
  new WithBootROM                            ++
  new WithSerial                             ++
  new WithPerfCounters                       ++
  new WithDefaultMemModel                    ++
  new WithL1ICacheSets(64)                   ++
  new WithL1ICacheWays(1)                    ++
  new WithL1DCacheSets(32)                   ++
  new WithL1DCacheWays(2)                    ++
  new WithCacheBlockBytes(64)                ++
  new freechips.rocketchip.system.DefaultConfig)
```

```
class ArchClassRocketChipMemLabConfig extends Config(
  new FRFCFS16GBQuadRank                     ++
  new WithBootROM                            ++
  new WithSerial                             ++
  new WithPerfCounters                       ++
  new WithDefaultMemModel                    ++
  new WithL1ICacheSets(64)                   ++
  new WithL1ICacheWays(1)                    ++
  new WithL1DCacheSets(64)                   ++
  new WithL1DCacheWays(1)                    ++
  new WithCacheBlockBytes(64)                ++
  new WithInclusiveCache(nBanks = 1,
                         nWays = 8,
                         capacityKB = 64)    ++
  new freechips.rocketchip.system.DefaultConfig)
```

Figure 5.11: Example Chipyard generator configurations used in a memory hierarchy lab exercise. Using a simple configuration-based user interface, the students are instructed to change the internal parameters of the caches such as associativity and cache size (middle example), as well as adding and removing components from the memory hierarchy (bottom example). The generator emits synthesizable Verilog which is then automatically compiled and simulated using a software RTL simulator or an FPGA-accelerated simulator for the students to characterize program execution behaviors across the different memory hierarchies. [16] (© 2021 IEEE).

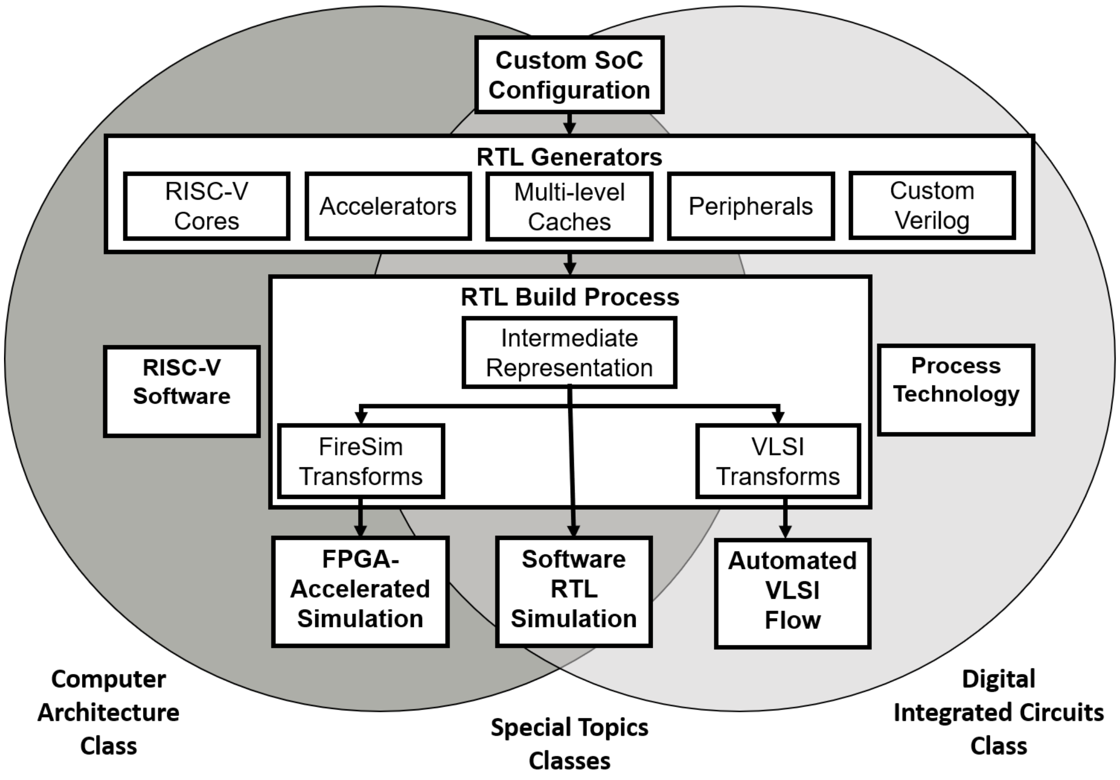
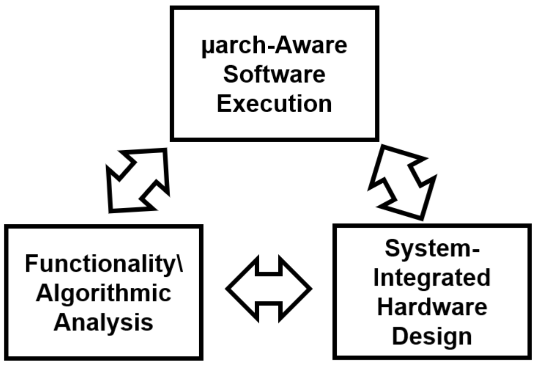


Figure 5.12: Multi-class flow using unified generator-based system framework. [16] (© 2021 IEEE).

across classes. As all three are advanced classes, they also include a project component. The unified infrastructure enabled students to complete more comprehensive class projects, focusing on different components of the framework based on the topic of the class. Students who utilized the framework for class projects in multiple classes demonstrated class project results that included both architectural innovations and characterizations with complete software evaluation, as well as integrated-circuits analysis with accurate VLSI power and area comparisons. These integrated projects effectively increase the capacity of a program to teach required system skills beyond the scope of a single class. In the subsequent year (Spring 2021), the Chipyard framework was again used in the same three classes, with its use expanding to an additional class dedicated to the fabrication of a complete SoC by students in a single semester [83], providing early evidence for the success of this approach and the Chipyard framework in training students for hardware/software co-design skills with full-system contexts. We believe that this approach has the potential to be further expanded to advanced systems-related classes in both electrical engineering and computer science - specifically in areas relating to embedded systems, networking, communications and operating systems.


µarch-Aware Software Execution
Functionality\ Algorithmic Analysis
System-Integrated Hardware Design

Class \ Lab Type	Functionality\ Algorithmic Analysis	System-Integrated Hardware Design	µarch-Aware Software Execution
Computer Architecture	Data-parallel Vectors	Branch prediction in out-of-order execution, prefetcher design	Cache tiling, Multithreading on multi-core, Speculation attacks
Hardware for Machine Learning	DNN quantization and precision	Systolic-array accelerator design	DNN tiling and scheduling on the accelerator

Figure 5.13: Lab flows across multiple courses teaching both hardware and software considerations

Single-class Hardware/Software Curricula

Even within a single class, the generator-based interfaces and FPGA-accelerated simulation within the Chipyard framework enable a lab exercises curriculum that spans both hardware and software. We follow three primary pillars of this broad curricula span for the design of lab exercises within a single class: functionality and algorithmic analysis, microarchitecture-aware software execution, and system-integrated hardware design. Figure 5.13 demonstrates how these pillars map the various lab exercises in two classes that have used this framework.

In the first pillar, *Functionality and Algorithmic Analysis*, students observe and analyze the impact of hardware considerations on the functionality of an algorithm. These labs demonstrate to students the preliminary stages of hardware design, as they do not consider performance evaluation yet, but rather just the functional behavior of certain hardware properties. These labs therefore use functional models based on the properties of the generators within the lab infrastructure, rather than the fully detailed implementations, in order to enable a wider space of exploration through faster development cycles. Example lab assignments that exercise these concepts include DNN inference accuracy analysis and vector-programming algorithm re-design. In the case of DNNs, the functionality of the algorithm to be analyzed is the inference accuracy, while example hardware properties such as quantization, pruning and accumulation precision could impact that functionality and require students to make adjustments in the algorithm. Similarly, translation of an algorithm to use lock-step SIMD and vector processing capabilities requires a functionality analysis of appropriate usage of vector-lengths, predication, and program ordering assumptions.

The second pillar, *System-Integrated Hardware Design*, attempts to help students experience the nuances of the interaction of different hardware components within a system. Often times in computer architecture education, micro-architectural features within a processor are designed in isolation and evaluated using closed-loop testing. While these approaches indeed help students learn about the micro-architectural properties of the specific hardware component under development, students often end up minimizing the impact of broader system considerations such as interaction with Inputs/Outputs, interaction with system buses and pipelines, and even simple rules-of-thumb such as Amdahl's law. As such, example assignments that exercise the concepts of this pillar include the design and implementation of an accelerator integrated within an existing SoC that includes control processors, a memory system, and support software. Co-simulation helps make hardware design exercises more accessible to students with a predominantly software development background. As an alternative to writing RTL, some lab assignments permit students to integrate C++ models into the Rocket Chip ecosystem. SystemVerilog DPI is leveraged to call C++ functions from a Verilog wrapper module, which in turn is instantiated as a Chisel blackbox within the SoC. This feature has proven particularly useful for implementing branch predictors and prefetchers, whose functional behavior is considered the main educational value, and where the design objective is purely a matter of performance, not correctness. C++ offers the familiarity of a procedural programming paradigm. Such an arrangement combines the "best of both worlds": A simplified C++ interface abstracts unnecessary details about pipeline signals and timing (to a certain extent), easing design space exploration, yet the surrounding out-of-order processor and system being simulated are still real artifacts, enabling realistic performance evaluation.

The third pillar, *Microarchitecture-Aware Software Execution*, addresses how the execution of software depends on its awareness of the underlying hardware microarchitecture. This includes topics such as the optimization space for high-performance computing, microarchitecture-dependent resource contention in parallel programming, and additional emerging areas of concern such as security. Software is traditionally written obliviously to hardware microarchitecture. However, traditional software often yields low utilization of existing computing hardware, and high performance computing work continuously demonstrates that correct and careful tuning of software scheduling to the properties of the microarchitecture is the primary method for obtaining performant software. This approach is further necessary with the presence of DSAs, which are designed to break the software-hardware abstraction barrier. As such, example assignments that exercise the concepts of this pillar include software tiling and scheduling to optimize software performance execution on an accelerator based on the properties of the accelerator (scratchpad memory size, accumulator size, array size, L2 cache size), as well as more traditional exercises such as multi-threaded programming in multi-core architectures with attention to synchronization primitives and resource contention within the microarchitecture. Recently, security topics have also entered this area of the curricula, with microarchitecture-aware side-channel attacks. As such, we were also able to provide hands-on experience for students with exercises to demonstrate such attacks using the out-of-order core implementation, with complete visibility into the speculation structures within the core microarchitecture.

Accessible FPGA Emulation for Education

The Chipyard framework also simplifies the usage of FPGAs for modeling and emulation in computer architecture classes. While the majority of FPGA usage in computer architecture classes focuses on FPGA *prototyping* in hardware-centric versions of the curricula, FireSim implements FPGA-accelerated *simulation*. In contrast with FPGA *prototyping*, FPGA-accelerated *simulation* correctly models timing behavior of not only the design under test but also the I/Os and peripherals of the SoC. FPGA-accelerated simulation enables deterministic and reproducible evaluation with a realistic system environment, as opposed to FPGA prototyping where each execution is sensitive to the FPGA environment, and timing depends on FPGA peripheral device performance (e.g. DRAM performance). This determinism is important in order to obtain reproducible class results and automated grading mechanisms. Furthermore, the construction of deterministic simulation also requires FireSim to automate the interaction between the host machine and the FPGA, which means that FireSim users do not need to directly interact with the FPGA toolchain and the FPGA-specific configurations.

Since FireSim is principally designed to work with cloud-hosted FPGAs, students do not interact with the complexities of any physical FPGAs. Furthermore, class size is not limited by the availability of physical FPGAs for hands-on lab exercises. AWS F1 instances enable scaling of FPGA-based lab exercises to large class sizes, since FPGA access is not limited by the number of physical FPGAs in the lab. FireSim further automates the interaction with the FPGA, so that users (and in particular, students) do not interact with any of the FPGA infrastructure at all.

As an example, in the architectural component of a Hardware for Machine Learning class, students were instructed to complete an RTL implementation of a machine learning accelerator with a software-managed scratchpad memory, and then study the effects of tiling and scheduling of DNN models using that implementation. The study of end-to-end performance of domain specific accelerators requires application-level evaluation using non-standard hardware implementations. The implementation stages were performed using the Chipyard framework and the base SoC and accelerator components it provides. Since DNN models would take many hours to run in software RTL simulation, FPGA-accelerated simulation was crucial for studying the tiling and scheduling components of the system. FireSim allowed students to perform fast experimental iterations with accurate performance evaluations.

During the first usage of FireSim within an advanced computer architecture class in Spring 2019 [261], a memory hierarchy analysis lab used similar configurations to the ones in Figure 5.11 to run tests out of two benchmark suites used for evaluation of commercial processors: SPEC CPU2017 intspeed [65] and GAPBS [31]. Although FPGA-accelerated simulation is orders of magnitude faster than software RTL simulation, the programs in the SPEC CPU2017 intspeed and GAPBS still take considerable time to run (50-80 minutes each on FPGA-accelerated simulation), while the relevant memory hierarchy effects could be demonstrated using shorter and more succinct programs. A retrospective comparison between these two example usages of FireSim in classes illustrates the trade-off in educational usage of this type of tool: simulation time must be appropriate to the level of the phenomena to be demonstrated in the exercise.

Remote Instruction

The Chipyard framework, in particular when using FireSim FPGA-accelerated simulation, enables a high level of accessibility and scalability of classes, as there is no physical infrastructure requirement. Class size is essentially limited by the cost of AWS credits (for cloud-hosted FPGA-accelerated simulation) and commercial EDA license usage for the VLSI flow. The lack of physical infrastructure requirements, and in particular the usage of cloud-hosted FPGAs, also presented an additional instructional benefit that was previously not as consequential. While the COVID-19 pandemic disrupted the Spring 2020 semester, the lab assignments in all three classes which used the joint framework were not disrupted despite the transition to remote instruction. However, while the theoretical possibility of large remote class exercises using these tools is within reach and attractive, hardware generator-based tools and cloud-hosted FPGA tools are still under research and development, and the usage experience of these tools is not yet entirely smooth. Large-scale class deployment of these tools would require more robust testing and infrastructure enhancements.

Usage of cloud-hosted FPGA instances presents additional educational considerations that are fundamental to the general usage of public cloud resources in classes. Using cloud-hosted FPGA instances teaches students to interact with the public cloud, which is an increasingly desirable skill in the current professional computing world. Just as how in many cases university assignments are the first time students interact with Linux command line interfaces which they later use for a significant part of their professional career, we believe that teaching students to interact with the public cloud has general benefits that extend beyond the basic assignment objectives of learning computer architecture concepts. Nevertheless, cloud FPGAs also come with challenges of managing variable spending among students in the class due to the dynamic nature of billing and usage. In traditional physical lab-based FPGAs and university instructional machines, students are only limited by their own time or availability of resources that have already been purchased. In contrast, when using cloud-based resources, there is virtually no availability constraints, rather only cost constraints, where cost is directly tied to usage. This raises interesting questions of responsibility and incentive models: Should resources be managed centrally for the entire class, or should students be allocated fixed amounts to "fund" their lab assignment? Should grades be associated with resource usage? What is the trade-off between students spending more time to further improve and perfect their assignment, while at the same time spending more resources in order to do so? While these questions are not unique to cloud-hosted FPGAs, we believe they take on new meaning due to the traditional usage model of FPGAs in classes.

Chapter 6

Generator-based SoC Hardware/Software Co-Design

Hardware/Software co-design methodologies have long been used for embedded systems design, in which hardware and software are closely co-designed for specialized applications [24]. Embedded systems have historically been domain-specialized, emphasizing safety, predictability and latency guarantees. These characteristics enable the software aspects in such systems to be confined to light-weight software stacks which can be evaluated relatively quickly on the relevant hardware. However, due to differing application requirements, the software stack of general-purpose systems is necessarily more complex, incorporating multi-processing, resource-sharing, file systems, virtual memory, and interactions with additional system components. Many of these system components are provided by fully-fledged operating systems such as Linux.

As an example, data-science-oriented applications often contain abundant data-level parallelism and are likely to benefit from multiple types of parallelism within the SoC, including multi-threaded or multi-core parallelism. A GNU/Linux-based environment helps provide simple programming interfaces for multi-threading such a POSIX threads [42] or OpenMP [68] libraries. At the application-level, more complex applications such as modern machine-learning and graph-analytics frameworks are built upon a large number of library components, many of which can be provided by common Linux distributions such as Fedora. These applications, which contain a high degree of parallelism, may require an additional degree of parallel processing, i.e. multi-node and distributed processing over a network. These settings commonly require full operating system support to manage the relevant devices and protocols associated with multi-node communication.

Providing a sufficient level of hardware/software co-design in such systems requires executing the complete software stack with sufficient visibility and performance to provide relevant insights. Often times, basic software samples are developed using architectural or functional models of a design, while the bulk of the relevant software is developed based on functioning silicon after the system (or a test system) has been fabricated. This process distorts a large part of the development flow, since hardware often times cannot be adjusted or optimized based on the final software version that was developed post-silicon. While functional properties can often be sufficiently verified pre-silicon, pre-silicon performance validation without sufficient software is a much more challenging task. FPGA-emulation as well as more complex commercial emulation platforms have been used for these purposes, and provide prime opportunities for the evolution of hardware/software co-design methodologies.

Since FireSim provides many of the features available in costly commercial emulation platforms, together with the accessibility of cloud-FPGA and integration with the generator-based development flow, it acts as a useful tool in advancing rapid pre-silicon software development. This in turns enables new degrees of generator-based hardware/software co-design and optimization.

6.1 Generator-Based SoC Design Space Analysis

We demonstrate two examples of generator-based pre-silicon design space exploration and analysis using two different accelerator generators for data-parallel computation integrated within multi-processor SoCs: the Hwacha vector accelerator generator, and the Gemmini deep learning accelerator generator. Generator-based design space analysis facilitates a holistic view of the system-level behavior of the accelerators under evaluation, and can be used to highlight the behaviors of parallel multi-processor systems with shared memory hierarchies. This is due to the intricate interactions of the system-level protocols and components that are implemented at an RTL-level rather than a modeling level.

In particular, FPGA-accelerated simulation using the FireSim platform enables application-level performance characterization for pre-silicon design space exploration. The FireSim environment constructs an FPGA-based simulator from the source RTL of the simulated design, making it a single-source-of-truth for both simulation and chip designs. FireSim enables integration of the simulated SoC with various peripheral and system-level models such as DDR3 memory [35], Ethernet NICs, and UART interfaces. FireSim has also been used to evaluate the performance of commercial SoCs [168]. Unlike conventional FPGA prototyping approaches, FireSim and its underlying compiler transform the target RTL description into a simulator rather than synthesize the target RTL onto the FPGA. This transformation enables decoupling between the simulated target design and the host FPGA platform, which allows for deterministic timing-accurate modeling of memory and I/O interfaces.

Hwacha Application-Level Design Space Exploration

The Hwacha vector accelerator was designed to execute data-parallel workloads in Linux-based data-intensive applications. As a general-purpose vector accelerator, it can be utilized by a large spectrum of software workloads - from high-level scientific computing matrix operations down to low-level operating-system kernel functions such as memory-to-memory copy [188], thanks to its design as a cache-coherent general-purpose accelerator with full virtual memory support. While preliminary evaluation of the Hwacha vector accelerator by using micro-benchmarks is informative [172], the desired level of evaluation for agile SoC design is application-level evaluation. Micro-benchmarks may demonstrate impressive performance in isolation, but when integrated into full systems and workloads, Amdahl's law and other effects have demonstrated the need for more comprehensive test suites in the past [218, 32]. Furthermore, since the Hwacha accelerator is a cache-coherent co-processor with virtual memory support, the memory system significantly impacts its application-level performance. Previous work using high-level models has demonstrated the impact of coherence and data-movement on accelerator integration in a system context [237].

Exploring the performance profile of different generator-based SoC configurations on their target applications can provide valuable pre-silicon design information, and well as complementary information to measurements from test chips which have already been fabricated with reduced feature sets [230]. Therefore, we perform application-level design space analysis for the Hwacha vector accelerator in a chip-multiprocessor setting for machine learning and data analytics applications.

Twelve SoC configurations (listed in Table 6.1) with vector accelerators were generated by using the Rocket Chip and Hwacha generators and evaluated on the FireSim platform. A scalar Rocket in-order application processor and a Hwacha vector accelerator are grouped together as a processor *tile*. A tile also consists of L1 data and instruction caches connected to the Rocket scalar core, as well as a floating-point unit. The SoC consists of an L2 cache which is shared between tiles using a cache-coherent TileLink [61] crossbar. The Hwacha vector accelerator memory interface is connected directly to the shared L2 cache through the TileLink crossbar. Additional peripheral interfaces such as the UART, block device interface and network interface controller (NIC) are added to the SoC for the purposes of full-system simulation. Each SoC instance was simulated with a DDR3 memory model with a 14-14-14 speed-grade. The varied design space exploration parameters include the number of processor tiles, the number of vector lanes per processor tile, and the sizes of L2 caches. Figure 6.1 illustrates the components of a FireSim FPGA-accelerated simulation and the evaluated SoC configurations. The largest configuration evaluated was a dual-tile chip multi-processor (CMP) with a dual-lane vector accelerator in each tile and a 2048 KiB L2 cache. The smallest configuration evaluated is a single-tile processor with a single-lane vector accelerator and a 512 KiB L2 cache. Note that this small configuration is also used as the baseline reference for speedup measurement (using only the scalar core). The SoCs are simulated to run at a frequency of 1033 MHz. This frequency is selected since test chips generated using these RTL generators have been previously signed-off to reach such frequencies. Furthermore, SoCs running at 1033 MHz demonstrate the distinction between FPGA prototyping to FPGA-accelerated simulation.

Of particular interest is a comparison between the architectures of two previously fabricated test chips: Hurricane-1 [279] and Hurricane-2 [230], both fabricated in the 28nm UTBB FD-SOI process technology. The Hurricane-1 test chip implements an architecture that contains dual Rocket application cores, each with a single-lane Hwacha vector accelerator. The Hurricane-2 test chip implements an architecture that contains a single Rocket application core with a dual-lane Hwacha vector accelerator. The Hurricane-1 configuration

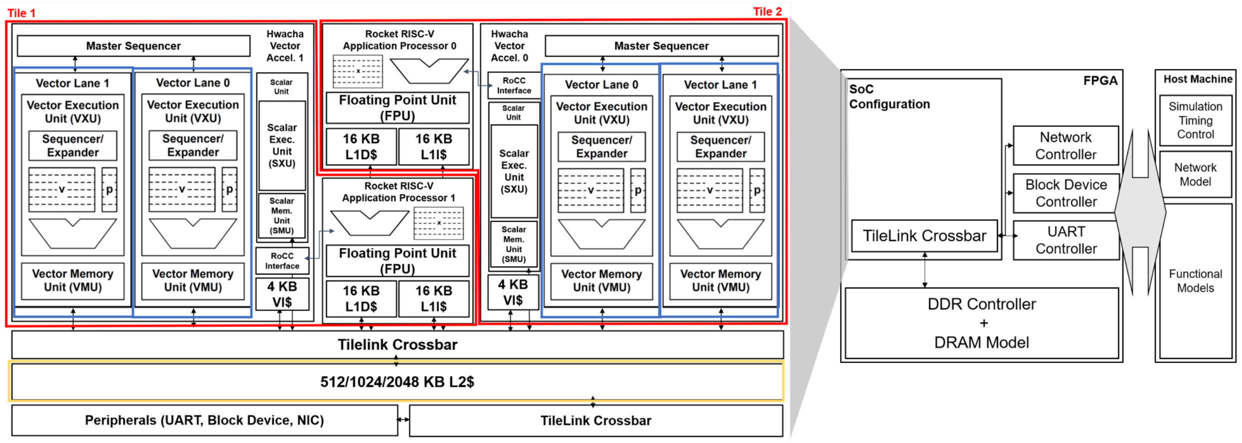


Figure 6.1: FPGA-accelerated simulation of Hwacha-accelerated SoCs with DDR3, block device, network, and other peripheral models, using various configurations of generated target SoC RTL. The generated configurations are varied across the numbers of tiles (marked in red), numbers of vector lanes per tile (marked in blue) and the L2 cache size (marked in yellow). [230] (© 2020 IEEE).

Config Name	Cores	Vector Lanes Per Core (Total Vector Lanes)	L2 Cache Size
1T 1L 512C	1	1 (1)	512 KiB
1T 1L 1024C	1	1 (1)	1024 KiB
1T 1L 2048C	1	1 (1)	2048 KiB
1T 2L 512C	1	2 (2)	512 KiB
1T 2L 1024C	1	2 (2)	1024 KiB
1T 2L 2048C	1	2 (2)	2048 KiB
2T 1L 512C	2	1 (2)	512 KiB
2T 1L 1024C	2	1 (2)	1024 KiB
2T 1L 2048C	2	1 (2)	2048 KiB
2T 2L 512C	2	2 (4)	512 KiB
2T 2L 1024C	2	2 (4)	1024 KiB
2T 2L 2048C	2	2 (4)	2048 KiB

Table 6.1: Hwacha-based SoC configurations

enables additional task-level parallelism, in contrast to Hurricane-2's single-core, dual-lane configuration which exploits more data-level parallelism.

Machine learning and deep learning in particular are arguably the fastest-evolving workloads in recent years. These workloads exhibit specific computation patterns, which are mostly centered around dense linear algebra kernels. Specifically, convolutional neural networks make up a significant portion of modern computer vision applications. SqueezeNext [97] is one such DNN that was specifically designed with hardware considerations in mind, with the goal of improving efficiency while maintaining high accuracy. In order to evaluate both the inference phase behavior and the training phase behavior of the deep neural networks at an application-level, we use the Caffe [146] framework, which provides useful abstractions for machine learning researchers. The Caffe framework also enables evaluation of arbitrary neural networks which were trained using this framework (unlike an evaluation of a specific instance of an efficient neural network software implementation mapped to a particular hardware instance).

A common problem in application-level evaluation regards the method of mapping a workload to a particular accelerator. The most comprehensive approach is compiler-based, in which a compiler is modified in order to account for the unique characteristic of the accelerator. With correct compiler optimizations applied, this code-generation approach enables the evaluation of a wide range of workloads on the accelerator. However, a compiler-based approach requires extensive engineering effort, and depends on many components of the software stack. Furthermore, compiler-based approaches for general-purpose vector accelerators are limited by the architecture-specific auto-vectorization capabilities of the respective compiler. The Hwacha vector accelerator ecosystem has an existing OpenCL-based compiler solution [172], however, this solution did not fit our deep learning workload programming model which heavily relies on popular machine learning frameworks with limited (and sometimes non-existent) OpenCL support. An alternative, library-based approach was used, where several computational kernels are hand-written and optimized. These kernels are then packaged in the form of a library with a common interface and linked to the target workloads as a replacement to an existing library implementation. We identified that the BLAS [165, 76] interface exposed the most significant dense computation operations that Hwacha was designed to support. Furthermore, we identified that the Caffe framework uses the BLAS interface for efficient mapping of neural networks to basic computational routines. We implemented multiple BLAS subprograms using custom written Hwacha kernels (nicknamed Hwacha-BLAS). Since these subprograms implemented the standard reference BLAS interface, they were able to be linked transparently to the Caffe framework without any modification to the larger framework code-base. Hwacha-BLAS provides a level of abstraction, allowing us to optimize basic dense linear algebra routines such as General Matrix Multiplication (GEMM) and vector dot products. This level of abstraction also allows us to attempt various parallelism techniques within the Hwacha-BLAS implementation. In particular, we use OpenMP [68] to parallelize GEMM implementations across multiple hardware threads. Figure 6.2 presents the software stack used for a full-system evaluation of SqueezeNext DNN inference and training using the Hwacha vector accelerator.

The results of the evaluation, illustrated in Figure 6.3, confirm that application parameters, such as the batch size in each iteration of deep learning inference or training passes, have an effect on the possible speedup obtained from an SoC configuration. Increased batch sizes group several independent input feature maps into a single iteration, hence allowing for more data-level parallelism as well as re-use of weights. For DNN inference, additional

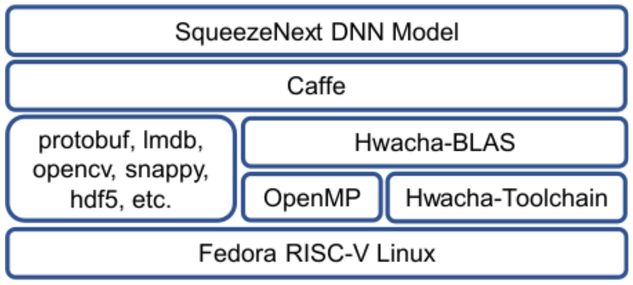


Figure 6.2: Dense DNN inference/training software stack

vector lanes do not provide additional speedup for batch size 16, as opposed to a batch size of 1. This is not an obvious result, as batched applications are known to expose more data-level parallelism than unbatched applications. We postulate that this result is due to non-vectorized processing that is required of the additional data in the batched workload. This non-vectorized processing is parallelized across the two scalar cores in the multi-tile architectures (such as Hurricane-1), but cannot be parallelized using the single scalar core in single-tile architectures (such as Hurricane-2), despite having multiple vector lanes.

We conclude that for infrequent inference applications, a dual-lane/single-tile SoC configuration such as Hurricane-2 is advantageous, while for batch-oriented applications, a dual-tile/single-lane configuration such as Hurricane-1 performs better. Furthermore, we observe that the speedup obtained for a Backward Pass (the training stage) is significantly less than the speedup obtained for the Forward Pass (the inference stage). This is since the inference stage is relatively homogeneous and consists mostly of a GEMM kernel. At the same time, the backward pass has a larger variety of computational kernels (some of which are more difficult to vectorize) and therefore, as expected from Amdahl's law, we see a smaller speedup for the backward pass. We also notice that additional vector lanes do not provide additional speedup (as opposed to additional tiles). This can be an indication of a possible limited level of data-level parallelism that can utilize multiple lanes. It can alternatively be an indication of poor performance scalability of the vector accelerator multi-lane design, requiring further micro-architectural optimization.

Another popular and quickly evolving workload in recent years involves data analytics. While data processing and databases are a wider topic that should be evaluated separately, graph analytics in particular has been explored as an emerging domain of data analytics since graphs provide a useful abstraction to describe relationships between objects. However, unlike deep learning, the computational primitives of graph analytics are not as clear. This is partially due to the fact that graph analytics are characterized mostly by the *data-representation* (a graph), rather than by a *computational* kernel. Nevertheless, some common graph processing kernels exhibit data-parallel properties, which are fertile ground for vector acceleration. In particular, we evaluate the performance of PageRank [205] - a graph algorithm designed to measure the importance of vertices in a graph.

Similarly to the DNN case, we use a domain-specific framework, in this case, Graph-

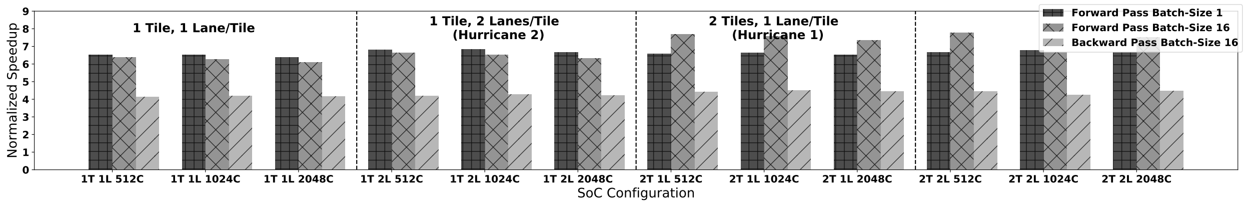


Figure 6.3: Design space evaluation of the speedup of inference and training of the SqueezeNext DNN model on the vector accelerator across tiles (T), lanes (L) and L2 cache sizes (C, KiB) compared to a minimal reference scalar implementation (1T 512C). Larger batch sizes expose more parallelism, which enables improved performance using multiple tiles and multiple vector lanes. Note that L2 cache size does not have a consistent effect. [230] (© 2020 IEEE).

Mat [249]. GraphMat is a parallel high-performance graph processing framework which uses a double compressed sparse column/row (DCSC/R) data structure [40] for the representation of the graphs in our analysis. In the case of the graph-processing domain, a framework is especially useful for the ingestion and pre-processing of a graph dataset, and the construction of an efficient graph representation data structure. Contrary to the DNN example, since graph computation functions are not as well-defined, we did not identify a basic set of primitives in order to provide a common interface equivalent to the BLAS library which was implemented for the dense linear-algebra case. While there are efforts in the research community to identify such basic sets of kernels for graph processing, for example GraphBLAS [154], those efforts often involve operator-overloading, which provide useful software abstractions but are challenging for direct and efficient mapping to hardware. Unlike the dense linear algebra kernels which were used in the deep learning case study, sparse linear algebra kernels are notoriously difficult to parallelize efficiently due to their irregular structure. This irregular structure exposes different implementation tradeoffs between data-locality and load-balancing based on the data structure representation. In the DCSR representation, we use custom implementations which utilize the two levels of indirection of the DCSR data structure to apply a coarse-grained level of parallelism using OpenMP pragmas, and a finer-grained level of data-level parallelism using Hwacha assembly vector instructions. Figure 6.4 presents the software stack used for a full-system evaluation of parallel PageRank graph processing kernels using the Hwacha vector accelerator.

The implementations were evaluated on 3 graph datasets from the Stanford Network Analysis Project [176]. The inner-most kernel implementation uses a "Loop Raking" vectorization technique, which was previously found to be the most performant technique on the Hwacha accelerator compared to alternative kernel implementations [15, 9]. Loop raking is a vectorization pattern which was originally proposed for sorting algorithms [283], and has since been commonly used for two-dimensional data structures [20].

The results of the evaluation, illustrated in Figure 6.5, confirm that application parameters such as the graph properties affect the possible speedup obtained from an SoC

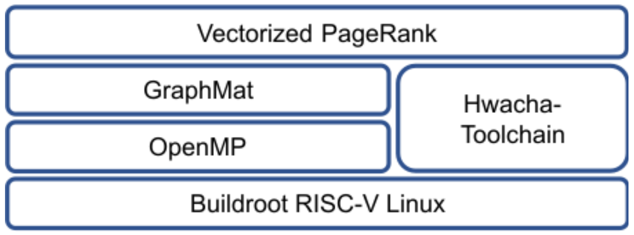


Figure 6.4: Sparse graph processing workload software stack.

configuration. The most significant speedup (25x) is obtained for a small graph (wikiVote, ∼400 KiB) that fits in L2 cache and utilizes the additional computational resources of the vector accelerator. Notably, variations in L2 cache size have minimal effects on all workload types, indicating little temporal locality within the applications. The only significant effect of cache size on speedup relates to the PageRank execution on the wikiVote graph which fits entirely within even the smallest evaluated cache size. If a graph does not fully fit in the L2 cache, then additional cache size does not provide a benefit. It is further evident that additional computational resources provide added benefits in all graphs. Hence, we can conclude that in a scenario with a fixed area budget for an SoC destined to run these types of graph applications, we would rather use the provided area for additional vector units or tiles rather than for additional cache.

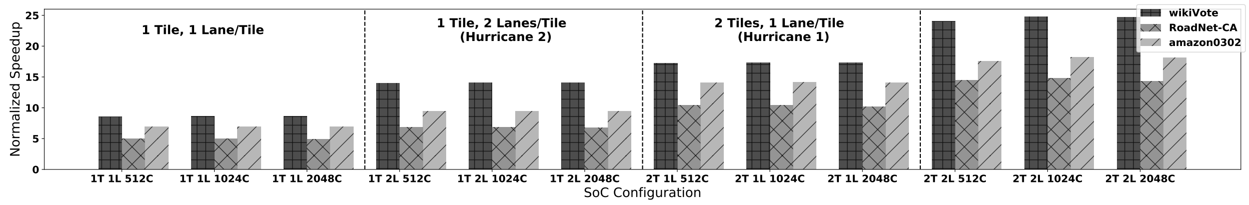


Figure 6.5: Design space evaluation of vectorized PageRank on different graphs using GraphMat infrastructure and the vector accelerator across tiles (T), lanes (L) and L2 cache sizes (C, KiB) compared to a minimal reference scalar implementation (1T 512C). The dual-tile/single-lane configuration provides greater speedup than the single-tile/dual-lane configuration (with similar area overheads). L2 cache size is not a factor for speedups on large graphs. [230] (© 2020 IEEE).

A common perception regarding graph workloads and sparse data structures regards the benefits of task-level parallelism vs. data-level parallelism. We evaluate the relative speedup of the vectorized PageRank implementations compared to the coarse-grained parallelism provided only by OpenMP pragmas and the scalar cores. We observe that for a single vector lane per tile, a dual-tile design will provide a higher relative speedup than the single-tile design, which demonstrates that the combination of data-parallel vector accelerators in conjunction with multi-processors provides a performance advantage compared to each of them individually. Between the Hurricane-1 and Hurricane-2 configurations, the dual-tile/single-lane design achieves the greatest PageRank speedup, because fine-grained load balancing across the decoupled cores targets the irregular structure of graph representations better. It is worth noting that since PageRank is in-fact implemented as an iterative process of sparse matrix-vector multiplication operations (SpMV), the conclusions regarding this parallel PageRank implementation can be applied to other applications which use SpMV primitives.

In summary, the application-level design space analysis of SoCs with the Hwacha vector accelerator demonstrates application-dependent performance of a range of SoC configurations. Specifically, it provides critical information about the performance benefits of multi-tile architectures compared to multi-lane architectures on full-system workloads with irregular structures or insufficient data-parallelism. This information complements additional power, energy and area data collected using generator-based test chips such as Hurricane-1 and Hurricane-2.

Gemmini System-level Analysis

A generator-based design space exploration was performed during the early development stage of the Gemmini accelerator, exploring the composition of a single Gemmini accelerator [95]. Building upon that preliminary design space exploration for the structure of a single accelerator, we demonstrate the additional benefits of generator-based design space analysis for system-level insights about accelerators at a multi-processor level. In particular, we demonstrate an analysis of the resource contention aspects of a shared-memory hierarchy using generator-based application-level analysis [96]. Recent work observed how limited host memory bandwidth contention affects cloud-based machine learning accelerators (specifically, the Google TPU) in which the CPU is shared by multiple applications [289]. In such cases, the host CPU memory bandwidth is the resource under contention, since multiple CPU applications require access to memory, as well as the single application running on the accelerator. We use Gemmini and FireSim FPGA-accelerated simulation to study a resource-contention scenario which is more representative of tightly-integrated edge devices with shared on-chip memory hierarchies. The SoC architecture under evaluation is composed of a dual-core application processor, in which each core has a dedicated Gemmini accelerator tightly integrated with it. The system has a total of two host processors and two Gemmini accelerators. In this SoC architecture, a shared last-level cache (L2) is shared by both the host CPU cores and the accelerators, as shown in Figure 6.6.

When both application cores are running workloads concurrently, we would expect contention over the shared L2 cache to impact the execution time of each workload, compared to running the same workload on an isolated single core. We would also expect this phenomenon to repeat itself when running applications which utilize the Gemmini accelerators, since the accelerators' memory interface is also connected to the shared L2 resource. We indeed confirm this expectation by measuring the performance of a ResNet-50 inference application [118] on a single core using Gemmini, and measuring the performance of two cores, each executing separate ResNet-50 inference applications using separate Gemmini accelerators.

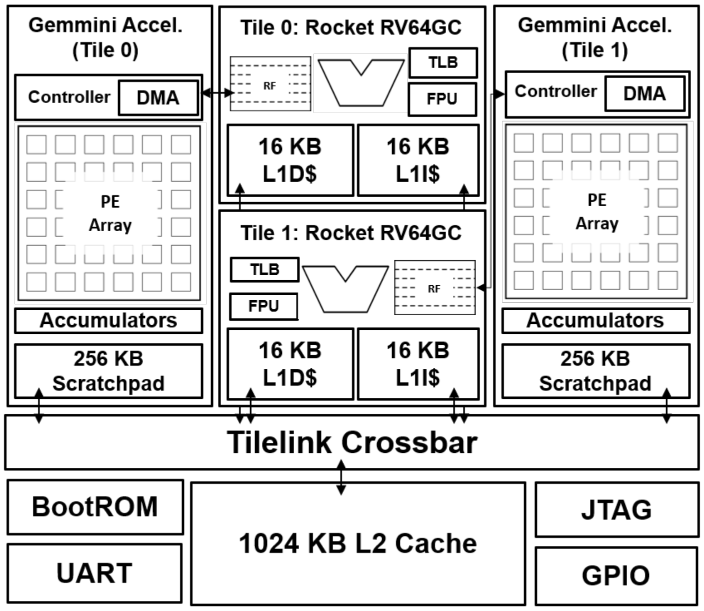


Figure 6.6: Dual-core system-on-chip with Rocket Linux-capable RISC-V application cores, a Gemmini accelerator attached to each Rocket core, a shared L2 cache, and standard peripherals. [96] (© 2021 IEEE).

The applications are run within an SMP Linux environment and pinned to their respective cores. The contention over the L2 cache results in a 13% slowdown in the execution cycles of each ResNet-50 inference application over running only a single core.

However, it turns out that the reasons for this performance profile are somewhat more nuanced, and present a design space tradeoff. Real-world DNN applications, such as CNN inference, have diverse layer types which have different computational requirements, and which contend for resources on an SoC in different ways. For example, ResNet-50 includes convolutions, matrix multiplications, and residual additions, which all exhibit quite different computational patterns. Convolutions have high arithmetic intensity; matrix multiplications have less; and residual additions have almost no data re-use at all. Additionally, unlike the other two types of layers, residual additions benefit most if layer outputs can be stored inside the cache hierarchy for a long time, rather than being evicted by intermediate layers, before finally being consumed several layers later. These different layer characteristics suggest different ideal SoC configurations.

Within the context of design space exploration, the design space of on-chip memory usage presents an interesting tradeoff. We are interested in the tradeoff between a larger L2 cache and a larger private scratchpad for two reasons: First, while a larger L2 should provide a benefit to all components of the target application (both those which utilize the accelerator, and those that run only on the host CPU and do not use the accelerator), greater re-use within a larger scratchpad may alleviate some of the load of requests to the L2 cache which generate the resource contention. Second, the benefits of a manually-managed

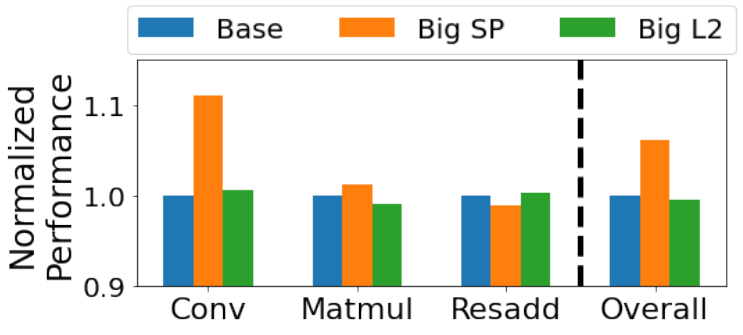
Config Name	Cores	Scratchpad Size (per core)	Accumulator Size (per core)	L2 Cache Size
Base-Single	1	256 KiB	256 KiB	1024 KiB
Base-Dual	2	256 KiB	256 KiB	1024 KiB
BigSP-Single	1	512 KiB	512 KiB	1024 KiB
BigSP-Dual	2	512 KiB	512 KiB	1024 KiB
BigL2-Single	1	256 KiB	256 KiB	2048 KiB
BigL2-Dual	2	256 KiB	256 KiB	2048 KiB

Table 6.2: Resource contention SoC configurations

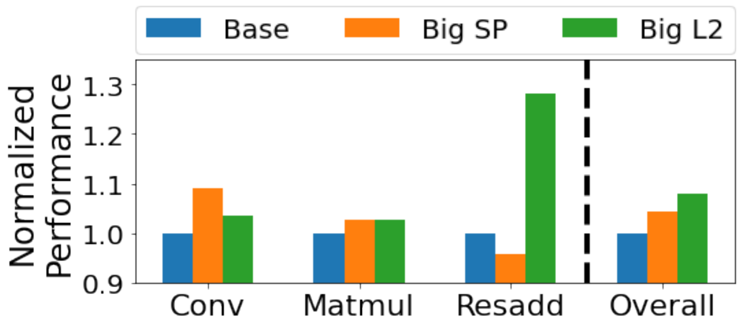
or controller-managed scratchpad are only as effective as the software which manages it, in contrast to a locality-based cache which may provide benefits for a wider range of software implementations. Since the software scheme generated by Gemmini uses only a single level of loop tiling (based on the size of the Gemmini scratchpad and accumulators), changing the size of the scratchpad inherently changes the L2 access pattern of an application which uses the Gemmini accelerator.

We explore methods to mitigate the performance impact of shared memory resource contention by adding additional memory resources to the system. The Generator-based approach enables us to study two additional SoC configurations with added memory. We explore the design space of an allocation of an additional on-chip SRAM memory budget of 1 MiB between the accelerator's private memories and shared L2 cache. In the first configuration (which we call *BigL2*), we add on-chip memory to the shared L2 cache (making it 2048 KiB rather than 1024 KiB) but leave the size of the Gemmini scratchpads identical to the base configurations (256 KiB each). In the second configuration, we leave the size of the L2 cache identical to the base configuration (1024 KiB) but increase the size of the Gemmini private memories (scratchpad and accumulators, 512 KiB each rather than 256 KiB each). Table 6.2 lists the different SoC configurations and their respective parameters.

As shown in Figures 6.7a and 6.7b, convolution layers benefit from a larger, explicitly managed scratchpad, due to their very high arithmetic intensity. Convolution kernels exhibit a 10% speedup with one core, and an 8% speedup in the dual-core case, when the scratchpad and accumulator memory sizes are doubled by the addition of a total of 1 MiB worth of SRAMs. The matrix multiplication layers, on the other hand, achieve only a 1% and 3% speedup when the scratchpad and accumulator memories are enlarged (in the single-core and dual-core cases respectively), due to their lower arithmetic intensity compared to convolution layers. Residual additions, which have virtually no data re-use and are memory-bound operations, exhibit no speedup when increasing the private scratchpad and accumulator memory sizes. Instead, they exhibit a minor 1%-4% slowdown, due to increased cache thrashing. In the single-core case, the increased convolution and matrix multiplication performance is enough to make the design point with increased scratchpad memory, rather than increased L2 memory, the most performant design point.



(a) Single core SoC configurations.



(b) Dual core SoC configurations.

Figure 6.7: ResNet-50 performance (by operation type), normalized to the performance of the Base configuration.

However, in the dual-core case, the convolution and matrix multiplication speedup cannot overcome the contention effects driven by the memory-bound residual addition operation. Figure 6.7b demonstrates that in the dual-core case in which applications compete for the same shared L2 cache, allocating the extra 1 MiB of memory to the shared L2 cache improves overall performance more than adding that memory to the accelerators' scratchpad and accumulator memories. Increasing the scratchpad size still improves convolution performance more than increasing the L2 cache size, but this improvement in performance is more than negated by the 22% speedup of residual additions that the dual-core BigL2 design point enjoys. This is because each core's residual addition evicts the input layer that the other one is expecting from the shared L2 cache, increasing the latency of memory-bound residual addition layers. The dual-core BigL2 configuration, which increases the shared cache sizes, alleviates this contention, reducing the L2 miss rate by 7.1% throughout the entire ResNet-50 run, and increasing overall performance by 8.0%. The BigSP configuration, on the other hand, improves overall performance by only 4.2% in the dual-core case.

To summarize, this design space exploration demonstrates that for SoCs which target DNNs dominated by convolution and matrix multiplication, contention can be mitigated by adding additional private scratchpad memory to the accelerators as opposed to adding shared memory resources. The low-latency data re-use within the scratchpad sufficiently compensates the additional L2 cache miss penalty which results from allocating memory resources to the scratchpads rather than to the shared resource. However, for SoCs which target DNNs that include memory-bound operations such as residual-additions, allocating additional memory to a shared resource such as the last-level cache would provide the best mitigation for L2 cache contention between the two cores, since these long-latency memory operations with no data re-use are extremely costly and can dominate execution time.

6.2 Generator-based Software Debugging and Optimization

After the initial performance measurements of a system are obtained using FPGA-accelerated simulation, the next step in the HW/SW co-design process is fine-grained characterization of this performance profile in order to optimize it. FPGA-accelerated simulation can provide valuable pre-silicon insights at this stage of the development process as well. However, a frequent challenge with FPGA emulation is the reduced level of visibility into the hardware aspects of the design when compared to the visibility of software RTL simulation. At the same time, traditional software profiling tools are often micro-architecture agnostic, which limits visibility from the software side as well. In order to enable HW/SW co-design in this pre-silicon performance optimization process, the pre-silicon simulation and emulation platform must enable debugging features which appeal to both hardware developers and software developers. The FireSim platform provides support for several FPGA-based debugging and profiling techniques, including co-simulation, synthesizable printf statements and synthesizable assertions [157]. In this section, we highlight several FireSim features which address hardware/software characterization of generated hardware designs executing system-level software. Specifically, we will demonstrate a hardware-centric debugging-assistance feature, a software-centric profiling feature, and an additional middle ground characterization feature to demonstrate the spectrum of hardware/software co-design tools in the profiling and debugging process.

Automatic ILA Generation

Hardware developers often use signal waveforms to debug and characterize digital hardware designs. As such, integrated logic analyzers (ILAs) are a common debugging method when using FPGAs. ILAs are digital FPGA IP blocks which probe signals on the FPGA and continuously sample the probes into an on-chip memory. When a trigger signal is observed on one of the probed signals, the content of the sampled memory is offloaded to the host FPGA management application, providing the designer a waveform of the probed signals of a time window around the event of interest.

The traditional flow for using an ILA typically requires manually generating the ILA hardware IP block in the FPGA vendor toolchains, instantiating the block in the top-level

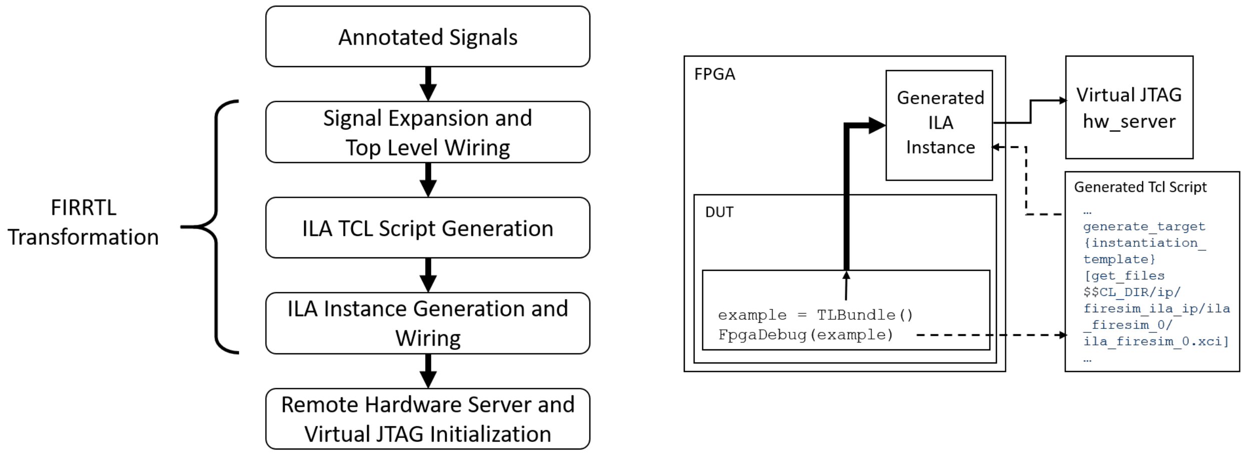

Annotated Signals
Signal Expansion and Top Level Wiring
ILA TCL Script Generation
ILA Instance Generation and Wiring
Remote Hardware Server and Virtual JTAG Initialization
FIRRTL Transformation
FPGA
Generated ILA Instance
Virtual JTAG hw_server
Generated Tcl Script
DUT
example = TLBundle()
FpgaDebug(example)
…
generate_target {instantiation_template} [get_files $$CL_DIR/ip/firesim_ila_ip/ila_firesim_0/ila_firesim_0.xci]
…

Figure 6.8: AutoILA generator-based integrated logic analyzer (ILA) integration flow.

RTL design, and wiring the relevant probe signals into the ILA IP block. Alternative vendor-specific flows enable annotating the probe signals directly in the Verilog source-code with vendor-specific annotations. Generator-based designs written using Chisel enable further automation of the ILA debugging flow through generated collateral during the elaboration process, annotation of signal groups, and conditional annotation using meta-programming schemes.

AutoILA is a FireSim feature designed to simplify debugging Chisel-based generators in FireSim using ILAs. This hardware-centric feature is designed to provide hardware designers with a familiar waveform-based debugging environment. Signals or bundles of interest are annotated in the Chisel generator source code with an `FpgaDebug` annotation. A custom FIRRTL transformation processes the annotated signals and bundles, and wires them out to the top level of the design. As is the case in any Chisel-based generator, annotations can be applied conditionally using meta-programming in Scala. The FIRRTL transform also outputs a custom Tcl script which executes on the FPGA vendor toolchain (Xilinx Vivado) to generate the appropriate ILA IP. The FIRRTL transform then modifies the generated Verilog to instantiate the ILA IP block and wire together the low-level signals and the ILA. The probed signals are automatically wired out to the top level of the DUT so they can be connected to the ILA at the top level of the FPGA design harness. This automatic wiring can help generalize the AutoILA flow to additional debugging IP blocks which require wiring of a custom digital block to probe signals deep in the design. In contrast to the traditional Verilog-based ILA annotation flows, AutoILA enables signals and bundles to be annotated at the generator level, before they are expanded to low-level signals based on the generator program. Figure 6.8 illustrates the implementation of the AutoILA feature using the FIRRTL transformation and wiring of the IP blocks and software drivers on the FPGA.

AutoILA has been used to uncover bugs and inefficiencies in Linux-based workloads that a bare-metal RTL software simulation environment was not able bring to the surface. For example, our experience of running Linux while offloading DNN kernels to a

Gemmini-generated accelerator uncovered a non-deterministic deadlock that would only occur if context switches happened at very particular, inopportune times. Deadlocks would occur because certain Gemmini instructions would assert the RoCC interface “busy” signal until they had finished receiving configuration instructions from the host CPU. However, if the host CPU were to context switch to another thread before sending *all* the configuration instructions, and that new thread were to fence, then the accelerator and the CPU would freeze forever waiting for each other to advance. Running on a full software stack with an OS also uncovered certain bugs where Gemmini had accessed regions of physical memory without the proper permissions. On a bare-metal simulation environment, these violations were simply silently ignored. These bugs were not exposed using simple bare-metal benchmarks on software RTL simulation, but were exposed using FireSim and identified using AutoILA. This demonstrates the importance of verifying hardware designs with complete system software and having the ability to debug the hardware using such full-system software stacks during pre-silicon development stages.

Segmented Cycle-level Execution Tracing

Software-centric profiling and debugging is often performed at the level of abstraction of instructions and functions, rather than individual micro-architectural components. As such, software developers are often only exposed to the semantics of the program and the programming language, rather than the digital signals and micro-architectural structures which implement the processor and the custom accelerators associated with it. Common software profiling and debugging tools typically utilize information sampled from the software-visible architectural state of the design. Specifically, instruction execution trace samples are useful for these software profiling tools, since they provide information about the progress of the program, which can then be associated with symbol tables and other ELF information to construct stack traces and program timing estimates, resulting in additional higher-level profiling insights.

FireSim provides a mechanism for extracting *cycle-accurate* instruction execution traces for analysis. This mechanism, called *TraceRV*, connects to the trace port of the simulated processor, and is able to backpressure the FPGA simulation when the instruction trace is not being copied to the host machine quickly enough. In essence, when the trace transport is backed-up, simulated time stops advancing, and resumes only when earlier trace entries have been drained from the FPGA, maintaining the cycle-accuracy of the simulation.

Directly capturing and logging any type of trace at a cycle-level granularity, and particularity committed instruction traces, has two significant drawbacks: Firstly, with high-speed FPGA-simulators like FireSim, it is easy to generate hundreds of gigabytes to terabytes of traces even for small simulations, which become expensive to store and bottleneck simulation rate due to the performance overhead of transferring trace data over PCIe and writing the trace data to disk. Second, both software and hardware developers are usually interested in *segmented* traces for a particular benchmark or application segment run, rather than profiling the entire simulation run. A full simulation run frequently involves booting an OS, running code to prepare data for a benchmark, running the benchmark, post-processing data, and powering off cleanly - all of which is information that is not relevant to the problem of interest. To address this problem, a *trigger* functionality was added to the FireSim debugging and profiling features, including the TraceRV feature, which allows trace logging to begin and end when certain user-defined external or internal conditions are satisfied. When trigger conditions are not satisfied, the various instrumentation mechanisms, including the TraceRV bridge, allow the simulation to advance freely without the performance overhead of copying data from the FPGA to the host over PCIe and avoiding writing to disk.

These trigger conditions can be set entirely at runtime (without re-building FPGA images) and include *cycle-count*-based triggers for time-based logging control, *program-counter*-based triggers, and *instruction-value*-based triggers. When instruction addresses are known, program-counter based triggers can be used to start and stop commit trace logging without any target-level software overhead. When instruction addresses are not known, instruction-value-based triggers are particularly useful, as some RISC-V instructions do not have side effects when the write destination is the `x0` register, and can essentially be used as hints to insert triggers at specific points in the target software with single-instruction overhead. In this particular example using the RISC-V ISA, the 12-bit immediate field in the `addi` instruction can be used to signal 4096 different events to TraceRV or to scripts that are processing the trace data. By compiling simple one-line programs which consist of these instructions, the user can even manually trigger trace recording interactively from within the console of the simulated system. For example, Chipyard and FireMarshal software workloads now provide by default two software commands, `firesim-start-trigger` and `firesim-end-trigger`, which issue an `addi x0, x1, 0` and `addi x0, x2, 0` instruction respectively, to enable such user-defined triggering from within the simulation console. These commands have been used to profile networking benchmarks [151], as well as for targeted performance counter profiling across specific regions of software executing DNN workloads [96].

Figure 6.9 illustrates the support of the FireSim trigger mechanism for both target-level triggers (program counter, instruction value) and environment-level triggers (cycle count). This support for a diverse spectrum of possible triggers enables focus and visibility into regions of interest for a range of different usage scenarios, from targeted software profiling to debugging of timing-related software misbehavior. The trigger conditions demonstrate a HW/SW co-design advantage of FireSim-based FPGA-accelerated simulation. The FireSim triggers enable segmentation both based on software-level information such as instruction value that can be set by a software developer, as well as hardware-level information such as cycle-count which is typically more relevant to the hardware developer.

Out-of-Band Performance Counters

Performance counters are a common profiling tool embedded in designs for post-silicon performance introspection [194]. However, since these counters are included as part of the final silicon design's area, power, and other budgets, they are generally limited in number and frequently shared amongst many events, complicating the process of extracting meaningful

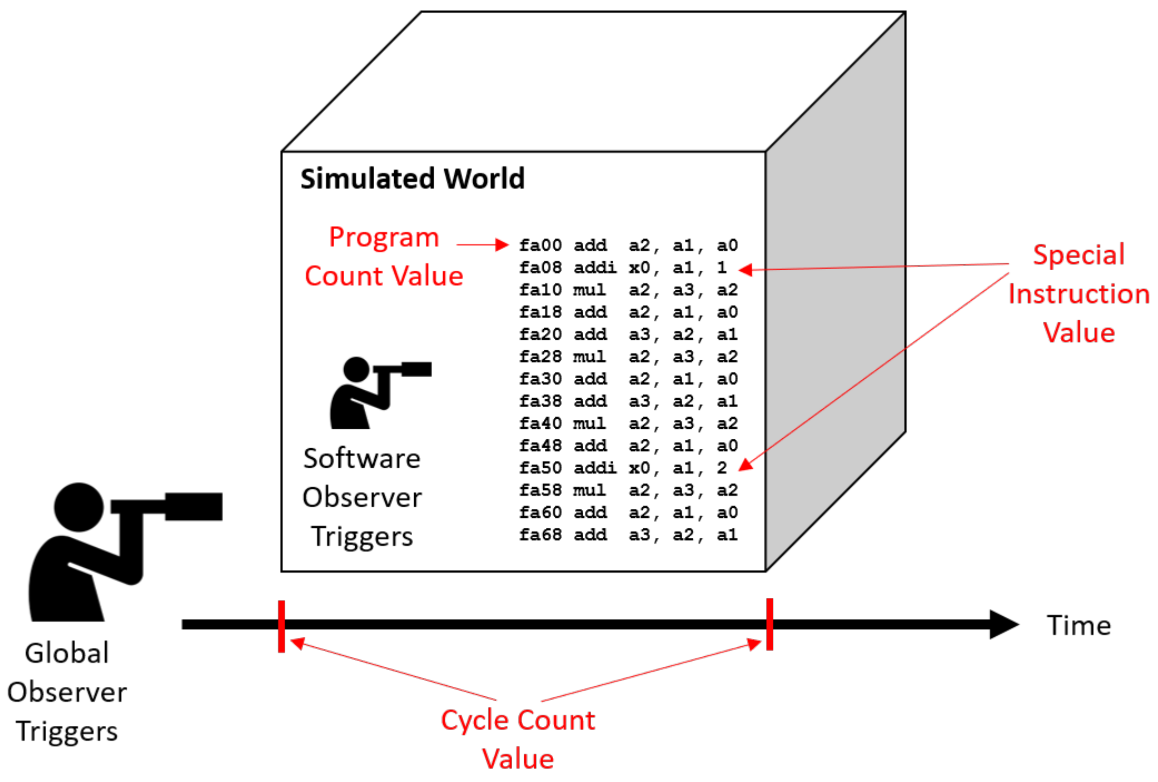


Figure 6.9: Debugging information segmentation triggering based on software-level information within the simulated world as well as hardware-level global information in FireSim.

information from them [181]. Pre-silicon use of performance counters in FPGA-simulation is not limited in this way. These counters do not need to be present in the final production silicon, and an unlimited number of counters can be read every cycle without perturbing the results of the simulated system (with the only trade-off being reduced simulation speed). *AutoCounter*, part of the FirePerf [151] collection of FireSim features, enables automatic hardware performance counter insertion for productive hardware-level performance profiling. These counters can be accessed *out-of-band*, meaning that reading the counters does not affect the state or timing of the simulated system—counters can be added easily and read as often as necessary.

The counters can be inserted ad-hoc by a designer using a simple FireSim `PerfCounter` API, or via more traditional verification cover points. Within the context of Chipyard and the Rocket Chip generator ecosystem, cover points are existing boolean signals found throughout the Rocket Chip SoC generator RTL that mark particular hardware conditions intended to be of interest for a verification flow. Unlike assertions, which only trigger when something has gone wrong in the design, cover points are used to mark signals that may be high under normal operation like cache hits/misses, coherency protocol events, and decoupled interface handshakes. By default, Rocket Chip does not mandate an implementation of cover

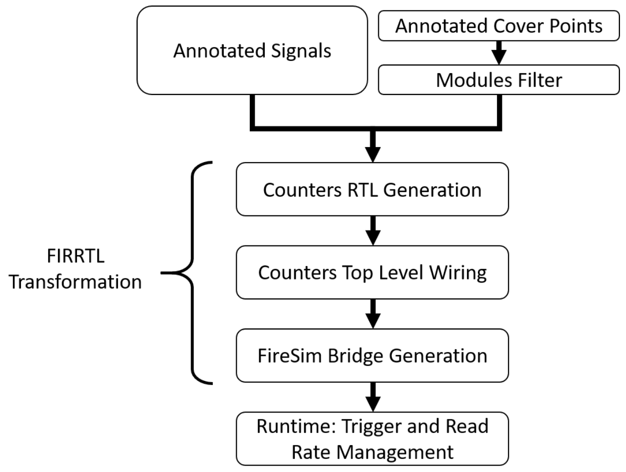


Figure 6.10: AutoCounter generator-based out-of-band counter insertion flow.

points; the particular flow being used on the RTL can decide what to "plug-in" behind a cover point. Unlike `printf`s, which print by default in most simulators, cover points can be inserted into designs without affecting other users of the same RTL codebase. This is especially important in open-source projects such as the Rocket Chip ecosystem. The cover API can also be expanded to allow the designer to provide more context for particular covers.

To enable adding out-of-band performance counters to a design in an agile manner, AutoCounter interprets signals fed to cover points as events of interest to which performance counters are automatically attached. AutoCounter also supports an extended cover point API that allows the user to supply multiple signals as well as a function that injects logic to decide when to increment the performance counter based on some combination of those signals. This allows for a clean separation between the design and instrumentation logic.

AutoCounter's automatic insertion of the performance counters is implemented by performing a transform over the FIRRTL [143] intermediate representation of the target SoC design. With a supplied configuration that indicates which cover points the user wishes to convert into performance counters, AutoCounter finds the desired covered signals in the intermediate representation of the generated design and generates 64-bit counters that are incremented by the covered signals. The counters are then automatically wired to simulation host memory mapped registers or annotated with synthesizable `printf` statements [157] that export the value of the counters, the simulation execution cycle, and the counter label to the simulation host. Figure 6.10 illustrates the AutoCounter out-of-band performance counter generator flow and interface with the simulation host.

By reducing the process of instrumenting signals to passing them to a function and automating the rest of the wiring necessary to pipe them off of the FPGA cycle-exactly, AutoCounter and FirePerf reduce the potential for time-consuming mistakes that can happen when manually wiring performance counters. Unlike cases where mistakes manifest as functional incorrectness, improperly wired performance counters can simply give confusingly erroneous results, hampering the profiling process and worsening design iteration time. This is compounded by the fact that marking new counters to profile requires re-generating an FPGA bitstream.

AutoCounter provides users with additional control over simulation performance and visibility. The rate at which counter values are read and exported by the simulation host can be configured during simulation runtime. As exporting counter values requires communication between the FPGA and the simulation host, this runtime configuration enables users to trade off frequency of counter readings for simulation performance.

Also at runtime, collection of the performance counter data can be enabled and disabled outright by the TraceRV-based trigger functionality. This enables designs to overcome the latency of re-building FPGA bitstreams to switch between different counters—many counters can be wired up at synthesis time, restricted only by FPGA routing resources, and can be enabled/disabled at runtime. Altogether, triggers eliminate extraneous data and enable higher simulation speeds during less relevant parts of the simulation, while enabling detailed collection during regions of interest in the simulation. The FirePerf AutoCounter flow enables a more holistic view of execution, as opposed to the limited capture window provided by other hardware-centric features such as ILAs. At the same time, the AutoCounter-injected counters still enable flexibility, determinism, and reproducibility (unlike post-silicon counters), while maintaining the fidelity of cycle-exact simulation (unlike software architectural simulators).

AutoCounter has been used in several SoC design-optimization studies. The first one, presented in a FirePerf case study, used AutoCounter to characterize and optimize the performance of the Linux networking stack on an SoC configuration supported by an integrated NIC [151]. In this study, out-of-band counters were added to several key points in the NIC micro-architecture to identify whether any of them were acting as hardware performance bottlenecks within the NIC. Specifically, counters were added to track send request and completion queue entry counts, receive request and completion queue entry counts, reader and writer in-flight memory transaction counts, hardware packet drops, availability of receive frames, and the fullness of the send buffer. Figure 6.11 illustrates how the performance counters are automatically generated and wired in the digital design of the NIC and connected to the simulation host manager. The request and completion queues in the controller are the principal way the device driver interacts with the NIC. To initiate the transmission or receipt of a packet, a driver writes a request to the send or receive request queues. When a packet has been sent or received, a completion entry is placed on the completion queue and an interrupt is sent to the CPU. The reader module reads packet data from memory in response to send requests. It stores the data into the send buffer, from which the data is then forwarded to the Ethernet network. Packets coming from the Ethernet network are first stored in the receive buffer. The writer module pulls data from the receive buffer in response to receive requests and writes the data to memory. If the receive buffer is full when a new packet arrives, the new packet will be dropped. By using AutoCounter, developers were able to constantly track whether the current level of performance was limited by a software bottleneck or hardware bottleneck, by identifying levels of queue and buffer occupancy through the automated counters. Using an iterative process and utilizing features such as

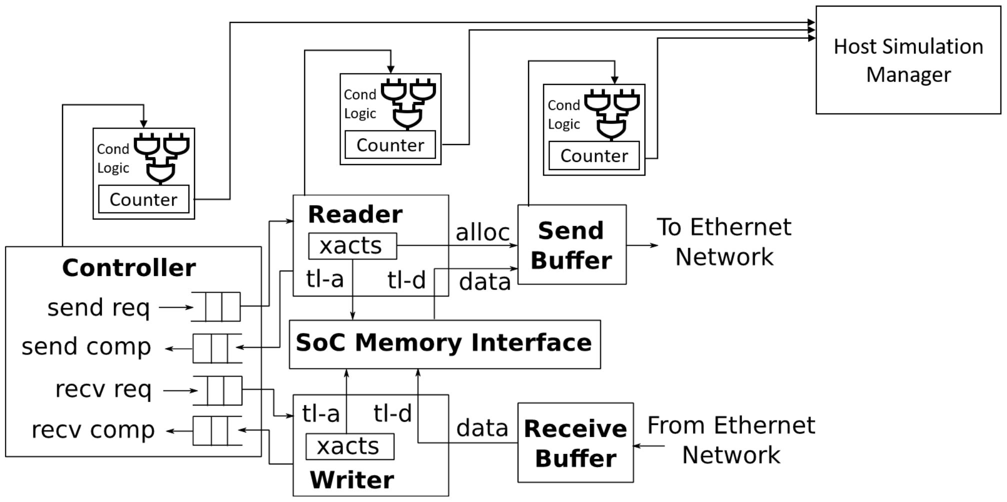


Figure 6.11: AutoCounter performance counters automatically inserted to an example NIC design and connected out-of-band to the simulation manager.

AutoCounter and TraceRV, developers are able to incrementally resolve both hardware and software bottlenecks.

AutoCounter has also been used in deeper analysis of the Gemmini resource contention case study presented in Section 6.1. By adding out-of-band performance counters throughout the memory system, researchers are able to confirm and validate the observations regarding the impact of each computational kernel on data re-use and the utilization of shared memory resources under contention. Similarly, AutoCounter has been used to identify performance bottlenecks throughout the development of the Gemmini accelerator by adding counters across buffers and queues within the Gemmini micro-architecture, using a similar methodology to the one used in the FirePerf NIC case study.

An additional usage example of AutoCounter is the characterization of program phases for power management algorithm analysis. An example presented in a paper by Schmidt et al. [279] demonstrates workload phases of a Linux boot process using cache miss counters and instruction retirement counters collected using AutoCounter (Figure 5 of Schmidt et al. [279]). This temporal characterization using out-of-band counters can then be used to set thresholds and analyze power management algorithms which utilize dynamic voltage and frequency scaling (DVFS).

6.3 Generator-based Performance Tuning

A major benefit of generator-based SoC design is the ability to generate additional collaterals together with RTL descriptions. Some of this collateral includes memory maps and device trees which can be used for boot code, operating system kernels and device drivers. An additional aspect of the generated collateral can include software header files (in particular, C header files) which can be included in the the SoC's software development kit (SDK) to help tune software based on the SoC's micro-architectural properties. As an added bonus, the same generated header files can be integrated into functional software simulators of the SoC (including accelerators) for correctness validation of target software. In contrast to functional simulation of general-purpose processors, the functional correctness of software targeting accelerator platforms often depends on the micro-architecture of the accelerator. For example, in accelerators with limited software-managed scratchpad memories or spatial computation arrays (e.g. Gemmini), software writing data to addresses exceeding the size of the scratchpad array would be functionally incorrect. By integrating the generated header file into a functional software simulator, the simulator can error on such events, providing faster debug mechanisms compared to full hardware RTL simulation.

Parameter-based performance tuning is a well-known technique in performance-sensitive software. Tuning-parameter values are typically set based on empirical experimentation during the design of the software library, or auto-tuning library components which evaluate the optimal tuning-parameter values by running tuning experiments directly on the target platform [274]. Template-based software design is a form of software meta-programming which enables the generation of a diversity of software implementations from a single-source software template. These implementations can be tuned during compilation time to a diversity of datatypes or tuning-parameters based on a centralized configuration. Generator-based hardware designs and their generated collateral can help simplify the tuning of performance-sensitive software for custom SoCs and integrate directly with template-based software design. By encoding first-hand knowledge of the custom microarchitecture in standard generated software header files, a template-based SoC SDK can be quickly tailored and tuned to the performance and capability profile of specialized hardware.

As an example, the Gemmini deep learning accelerator generator generates a header file which includes parameters describing the dimensions of the spatial computation array, the sizes of memory system components, the datatypes handled by the accelerator, and the existence of various optional functional units within the accelerator. An example of such a C header file is presented in Listing 6.1. Relevant segments from the equivalent hardware generator configuration which generated this header file are presented in Listing 6.2. The Gemmini parameters header file is then included in the software templates which form the basis for the low-level SDK used to run programs on the accelerator. These parameters help determine loop tiling factors, whether certain kernels or other code segments such as scaling can be executed on the accelerator or need to run on the host CPU, and whether datatype conversion or quantization are required in order to run on the accelerator.

The use of generated header files facilitates accurate pre-silicon design space exploration, as it enables comparison between well-tuned pairs of hardware microarchitectures and software benchmarks. This is in contrast to a common pitfall of design space exploration in which a variety of hardware configurations are explored with a set of software benchmarks that are tuned for a single microarchitecture configuration. This approach to hardware-software co-design enables data-driven analysis of the design space and opens up opportunities for automated optimization of hardware design points based on joint hardware-software design point evaluation. In fact, since hardware generators can be thought of as the hardware equivalent of software meta-programming, it is only natural that these two approaches integrate together to deliver an efficient hardware-software co-design cycle for a diversity of target applications.

Generated software header files further provide software flexibility in terms of accelerator integration in custom SoCs. RISC-V has several major opcodes reserved in its encoding space for user-defined custom instructions [272]. Composition of accelerators with custom instructions carries the risk of opcode collisions. While hardware generators can avoid this with a relatively simple configuration option which allows the SoC designer to choose a major opcode out of the available selection, such flexibility carries implications up the software stack, impacting compilers, assemblers and software development kits. The generated header files can provide solutions here as well, with templated implementations using the custom opcode indicated in the header file to provide correct instruction encodings across the software stack for a particular SoC configuration and accelerator composition. For example, the generated header file in Listing 6.1 includes the `XCUSTOM_ACC` parameter with the value 3, indicating to the software stack that out of four available major opcode encodings, the Gemmini accelerator in this SoC is using the major opcode encoded as `11` for the Gemmini custom instructions. This major opcode will be used as part of the assembler or custom instruction pre-processor macros to assemble the Gemmini custom instructions within the custom software development kit.

```
#ifndef GEMMINI_PARAMS_H
#define GEMMINI_PARAMS_H

#include <stdint.h>
#include <limits.h>

#define XCUSTOM_ACC 3
#define DIM 8
#define ADDR_LEN 32
#define BANK_NUM 4
#define BANK_ROWS 4096
#define ACC_ROWS 2048
#define MAX_BYTES 64
#define MAX_BLOCK_LEN (MAX_BYTES/(DIM*2))
#define MAX_BLOCK_LEN_ACC (MAX_BYTES/(DIM*4))

typedef uint16_t elem_t;
#define ELEM_T_IS_LOWPREC_FLOAT
static const float elem_t_max = 3.3895313892515355E38;
static const float elem_t_min = -3.3895313892515355E38;
typedef float acc_t;
typedef double full_t;

#define ELEM_T_IS_FLOAT
#define ELEM_T_EXP_BITS 8
#define ELEM_T_SIG_BITS 8
#define ACC_T_EXP_BITS 8
#define ACC_T_SIG_BITS 24
typedef uint16_t elem_t_bits;
typedef uint32_t acc_t_bits;
```

```
#define HAS_MVIN_SCALE
typedef float scale_t;
typedef uint32_t scale_t_bits;

#define HAS_MVIN_ACC_SCALE
typedef float scale_acc_t;
typedef uint32_t scale_acc_t_bits;

typedef float acc_scale_t;
typedef uint32_t acc_scale_t_bits;

#define row_align(blocks) __attribute__((aligned(blocks*DIM*sizeof(elem_t))))
#define row_align_acc(blocks) __attribute__((aligned(blocks*DIM*sizeof(acc_t))))

#define MVIN_SCALE_IDENTITY 1.0

#define ACC_SCALE_IDENTITY 1.0

#define ACC_SCALE_T_IS_FLOAT
#define ACC_SCALE_EXP_BITS 8
#define ACC_SCALE_SIG_BITS 24

#define ACC_READ_SMALL_WIDTH
#define ACC_READ_FULL_WIDTH


#endif // GEMMINI_PARAMS_H
```

Listing 6.1: Gemmini Generated Header File Example

```

val defaultFPConfig = GemminiArrayConfig[Float, Float, Float](
    opcodes = OpcodeSet.custom3,

    //datatypes
    inputType = Float(8, 8),
    outputType = Float(8, 8),
    accType = Float(8, 24),

    //spatial compute array
    tileRows = 1,
    tileColumns = 1,
    meshRows = 8,
    meshColumns = 8,
    dataflow = Dataflow.BOTH,

    //SRAMs
    sp_banks = 4,
    sp_singleported = true,
    acc_banks = 1,
    acc_singleported = false,
    num_acc_sub_banks = -1,
    sp_capacity = CapacityInKilobytes(256),
    acc_capacity = CapacityInKilobytes(64),

    //DMA
    dma_maxbytes = 64,
    dma_buswidth = 128,
    aligned_to = 1,
    tlb_size = 4,
    use_tlb_register_filter = true,
    max_in_flight_reqs = 64,
    use_dedicated_tl_port = false,
    acc_read_full_width = true,
    acc_read_small_width = true,
    mvin_scale_args = Some(ScaleArguments((t: Float, u: Float) => t * u, 4, Float
    (8, 24), -1, identity = "1.0", c_str="((x) * (scale))")),
    mvin_scale_acc_args = Some(ScaleArguments((t: Float, u: Float) => t * u, 4,
    Float(8, 24), -1, identity = "1.0", c_str="((x) * (scale))")),
    mvin_scale_shared = false,
    acc_scale_args = ScaleArguments((t: Float, u: Float) => t * u, 4, Float(8, 24)
    , -1, identity = "1.0",
      c_str = "((x) * (scale))"
    ),


    //additional hardware parameters such as pipeline depths, wiring paths, queue
    depths, etc., which do not affect software, ommited for brevity
)
```

Listing 6.2: Gemmini Generator Hardware Configuration Example

Integration with the Deep Learning Software Stack

Recent industry analysis has observed that despite the emergence of many AI accelerator startups, the depth and maturity of their software stacks is a major Achilles' heel on the path to market success [109]. Frequent changes in state-of-the-art models and the subsequent requirements from custom hardware often conflict with the tedious process of extracting high-performance from specialized hardware through hand-tuned software which can utilize it to the fullest extent. Using automatically-generated header files, Gemmini is easily integrated with the deep learning software stack. Specifically, Gemmini integrates with the deep learning stack using the ONNX (Open Neural Network Exchange) interchange format and the ONNX-runtime environment [214, 64]. Notably, custom execution providers in ONNX-runtime call into functions defined in the automatically-generated Gemmini header files, thus being able to immediately tailor the execution of the DNN to the micro-architectural parameters of Gemmini.

ONNX is an open interchange format that was created by a consortium of companies with the goal of providing interoperability across machine learning software frameworks. It encodes neural networks as directed acyclic graphs of *operators* operating on tensors. The operators have input and output tensors, as well as a set of attributes describing them. The ONNX specification consists of a set of standard operators which are required in order to maintain interoperability across frameworks. While ONNX-formatted files can be extended with custom operators, these can limit interoperability across frameworks and hardware platforms. Operators can then be processed by software frameworks and graph compilers in order to be mapped efficiently to hardware platforms.

The automated header files produced by the Gemmini generator integrate into templated ONNX operator implementations. We use the open-source ONNX-runtime software framework, which provides the notion of *execution providers*, to decouple between provider-dependent and provider-independent modules. While providers can generally be thought of as unique hardware architectures, they can also translate to different software library implementations. The execution providers register "kernels" for supported operators. These kernels are decoupled from the execution graph which can be processed and optimized independently based on the availability of kernels and operators. ONNX-runtime greedily assigns operator sub-graphs to execution providers based on their order of registration, with CPU-based execution providers having the lower priority as a fallback for all operators not supported by other execution providers.

The Gemmini execution provider uses templated implementations of matrix multiplications, convolutions, and fused operation including scaling, saturation, non-linear functions and pooling, based on indications found in the generated header files regarding their existence in the generated configuration of the accelerator. As an example, Listing 6.1 includes definitions such has `HAS_MVIN_SCALE` and `ELEM_T_IS_LOWPREC_FLOAT`. The former provides the software stack with an indication regarding the existence of a fused scaling operation as part of DMA transactions, while the latter indicates to the software stack that the computation datatype is a low-precision floating-point datatype that does not have native CPU support and would require conversion or quantization. The lack of fused operations would require falling back on CPU execution providers which requires data movement between the accelerator and the CPU.

Furthermore, operators such as matrix multiplication or convolutions, that require software management of accelerator resources for high performance (for example, efficient management of private scratchpad memories), can use the parameters in the generated header file to safely perform this management and compute tiling and loop unrolling factors. For example, the `BANK_ROWS` and `ACC_ROWS` parameters in Listing 6.1 are used to compute loop tiling factors, while the `DIM` parameter is used to compute a loop unrolling factor in the Gemmini execution provider templated implementations.

Using this templated approach, Gemmini is able to execute with high performance arbitrary ONNX models downloaded from online resources, including ResNet-50, BERT, and MASK-RCNN. Detailed analysis of the execution of such models using the ONNX-runtime framework and the Gemmini accelerator can be found in an associated technical report [214].

6.4 Generator-based SoC Hardware/Software Co-Design Summary

Hardware/software co-design of application-class SoCs requires executing complete and highly-tuned software stacks with sufficient hardware visibility and performance to provide relevant insights. In this chapter, we demonstrated several capabilities facilitating such co-design, enabled by generator-based SoC development, the Chipyard framework, and FireSim FPGA-accelerated emulation. Specifically, we discussed full-system design space analysis of accelerator generators for numerical data analysis – the Hwacha vector accelerator and the Gemmini deep learning accelerator. We highlight this design space exploration using complete system software stacks and SoC configurations which exercise tradeoffs in SoC resources such as private and shared memories, and SoC-level parallelism vs. accelerator-level parallelism. We discussed the tools and features enabled by generators and FPGA-accelerated emulation, which facilitate appropriate levels of visibility into digital signals while running full-system application-class software. These enable tighter loops of pre-silicon debugging and optimization of true target workloads, which are fundamental steps required for application-class hardware/software co-design. Finally, we discussed the ability to facilitate performance tuning of software for custom hardware through microarchitecture-aware template-based software design and generator-based software collateral generation. This integration and co-design of hardware and software parameterization enables rapid performance tuning of software for a variety of SoC and accelerator configurations based on the micro-architectural parameters of generated hardware blocks. We demonstrated this ability using the Gemmini deep learning accelerator generator, and the mapping of hardware generator parameters into software header files which get integrated into templated software libraries. Together, these represent new capabilities in application-class hardware/software co-design, all enabled by generator-based SoC development and the tools associated with it.

Chapter 7

Software Customization for Numerical Data Analysis

Recent analysis of modern processor architectures has identified that while the marginal performance and efficiency improvements of new chips is decreasing with the sunset of Moore's law, there are still significant potential gains through optimization of the software stack [175]. This idea, which was nicknamed *"there is plenty of room at the top"* (a contrast to the idea of *"there is plenty of room at the bottom"* from Richard Feynman's famed lecture [88], an idea which was popular during the era of Dennard scaling), observes that the modern software stack has relied on exponential improvement in single-thread performance over the era of Moore's law, and as a result developed many inefficiencies across the levels of software abstraction and reduction. At the same time, additional analysis observes that the increased specialization of custom silicon chips is likely to cause a "virtuous cycle" of fragmentation in computing, in which some applications would get massive investments and become orders of magnitude more efficient than they would on traditional general-purpose processors, while other applications will get much worst since they will no longer benefit from the performance improvement of general-purpose processors [257]. This analysis observes that collections of potential applications which lack sufficient coordination in terms of software development in order to justify specialized silicon are partially at fault for the "fragmentation cycle" caused by specialized processors. Hence, *efficient* mapping of software to custom SoCs is paramount in obtaining the customization targets of the SoC. This efficiency is measured both in terms of software performance and hardware utilization, as well as software developer time invested in optimizing the target software applications.

7.1 Software Mapping to Specialized Accelerators

The utility of specialized accelerators often depends on the quality of software mappings which enable applications to use these accelerators. The majority of programmers typically interface with hardware through high-level, general-purpose programming languages, as opposed to direct interfaces such as low-level assembly, machine language, and memory mapped I/O. A typical general and comprehensive approach to transforming a program written in a high-level general-purpose programming language and efficiently mapping it to a custom accelerator is through a compiler-based approach, in which a compiler is modified (or created from scratch) in order to account for the unique characteristics of the accelerator. With application of correct compiler optimizations, end-to-end code-generation by a compiler can theoretically enable high productivity of software developers as well as good code quality and performance on a wide range of workloads. However, writing a new compiler or modifying an existing compiler to account for accelerator mappings is a non-trivial endeavour due to nuanced trade-offs between accelerator semantics and programming language semantics, as well as the large number of dependencies of various programming languages on additional platform native components of the software stack.

Auto-vectorization in general-purpose compilers has been researched for many years as a solution for mapping general-purpose code to data-parallel hardware such as SIMD and vector processing units [203, 260, 195]. However, general-purpose auto-vectorization has seen limited adoption in mainstream compilers. Loop vectorizers in popular compilers such as LLVM and GCC primarily focus on analysis of an innermost loop body, but are challenged when faced with nested-loop structures. While advanced compilation techniques such as polyhedral analysis can assist in affine loops transformations and validation of more complex loop mappings, their computational complexity limits their use in mapping general-purpose code to accelerators. As a result, high-performance parallel software implementations typically rely on compiler intrinsics to explicitly call vectorized operations when optimizing complex multi-dimensional algorithms.

Programming languages with more explicit parallelism semantics such as OpenCL and NVIDIA CUDA assist with compilation to SIMD and vector acceleration hardware, but incur non-negligible integration costs with additional system components written in other general-purpose languages and software frameworks. Compatibility with additional languages and libraries may require transitions through scalar code in order to maintain program semantics, as well as matching of linker flags and function qualifiers for application binary interface (ABI) compatibility. For example, the Hwacha vector accelerator ecosystem has an OpenCL-based compiler solution [172]. However, deep learning and data science workloads rely heavily on popular machine learning and analytics frameworks, which are not well integrated with OpenCL.

The popularization of 2D spatial accelerators adds an additional facet to the problem of identifying and optimizing tensor operations in loop-structured code. Several recent approaches attempt to generalize compilation solutions to target 2D accelerators or dot product instructions. Some works propose hybrid compilation schemes in which the inner-loop tensor operations are encoded as hand-optimized minimal kernels, while the compiler provides additional compilation optimizations for outer-level loops and additional code [253]. Other works use domain specific languages (DSLs) for tensor operations to enable better-informed lowering transformations of code with high level abstractions down to tensorized instructions and accelerators [49, 273]. DSLs enable programming languages to further raise the level of programming abstraction, which provides compilers with additional information for mapping to complex accelerators and improved code generation. By encapsulating tensor operations using dedicated DSL constructs rather than general-purpose program semantics, a compiler can use higher-level program semantics to side-step the challenge of identifying and isolating tensor operations in general-purpose loop structures. Nevertheless, this approach does not resolve end-to-end optimization when integrating these objects with general-purpose programs.

Library-based approaches to accelerator software mappings are more targeted solutions as compared to compiler-based approaches. In contrast to compiler-based approaches, library-based approaches for high-performance execution rely on integration of hand-optimized computational kernels through common APIs and ABI compatibility rather than end-to-end optimization. In library-based approaches, common APIs and ABI compatibility enable capturing a sufficient level of abstraction required for accelerator mapping. Libraries with ABI compatibility can be interchanged at runtime without source-code re-compilation through dynamic linking, enabling flexibility when using accelerators and diverse system components.

Accelerator-based libraries are generally comprised of a select number of high-utility kernels that are hand-tuned by performance engineering (often by hardware vendors) to obtain maximal utilization of the underlying hardware. These kernels are then packaged in the form of a library with a common interface and linked to the target workloads as a replacement to an existing library implementation. These computational kernels can be additionally integrated into higher-level code generation frameworks. Nevertheless, libraries do not enable cross-kernel optimization across function boundaries. I.e, higher level programs which use a number of library routines will not be optimized end-to-end. Libraries can be optimized for performance only within their pre-determined API.

There is a strong relationship between libraries and DSLs. Both libraries and DSLs add new semantics and abstractions to a specific domain of interest by encapsulating and isolating lower-level concerns. However, DSLs often do so through the introduction of syntax and grammar, while libraries do so through the introduction of functions. The author is of the opinion that a proliferation of independents DSLs is not constructive from a user adoption and barrier-of-entry perspective, and that DSLs should be designed as extensions of existing core languages (sometimes referred to as "Internal/embedded" DSLs) similar to libraries, rather than new languages with their own syntax and semantics [11].

Notably, popular high-level languages such as Python, R, and Julia are heavily reliant on software library packages and package managers. These programming languages likely owe their popularity to their large selection of library and package repositories. In particular, the majority of data analysis code written in these high-level languages relies on a small number of software library packages such as NumPy, SciPy, Scikit-Learn and Pandas (Python), Caret, Flux and Tidyverse (R), and JuMP (Julia).

Library-based abstractions in software engineering thread a fine needle between enabling performance optimization and generating software bloat. Libraries are often optimized for a certain level of abstraction, and then later wrapped with additional packages and abstractions to generate higher level constructs. The previously mentioned "Plenty of Room at the

Top" article [175] identified this challenge of "reductionist software design", since software problems are reduced to related problems in order to eliminate software development time and decrease code replication. For example, BLAS libraries [165] are often extremely optimized for matrix multiplication. Deep learning frameworks utilize BLAS libraries in order to enable high performance DNN model execution. A software developer who needs to execute a matrix multiplication could potentially "reduce" their problem to a batched, single-layer, multi-level perceptron neural network which will execute the matrix multiplication within the deep learning framework, but that would be accompanied with additional software bloat generated by the deep learning framework. Hence, a more performant solution would be to use a library which was optimized for a more appropriate level of abstraction, which in this case would be a BLAS library.

With respect to domain specific accelerators, in the absence of general-purpose compilation frameworks with the ability to map code efficiently to an accelerator, identifying appropriate levels of library abstraction is paramount for maximizing the utility of an accelerator.

Supplemental Use of Domain Specific Accelerators

Supplemental-use of accelerators is a paradigm in which software developers can use the computational capabilities of domain-specific accelerators for applications which are not necessarily part of the original intended application domain of the accelerator. This paradigm of supplemental-use of domain-specific accelerator relies on several core principles and assumptions:

- Performance gains from supplemental-use on the accelerator will likely be limited and inferior compared to primary-use applications on the accelerator that the accelerator has been optimized for.
- Efficiency gains from supplemental-use on the accelerator may not be strictly positive. Efficiency gains depend on the utilization of the computation resources of the accelerator by the supplemental application in comparison to alternative resources on the SoC.
- Software development investment in supplemental-use applications must be cost-effective in relation to the potential performance gains.

The idea of supplemental-use of domain-specific accelerators is relatively prevalent in the high-performance computing (HPC) research community. Widely popularized with the adoption of graphics hardware for scientific computing in the early 2000s, including dense linear algebra [163], partial differential equations (PDEs) [222], fast Fourier transforms (FFTs) [192], physics simulations [114] and genome sequencing [185], researchers were able to identify the parallel nature of graphics hardware and map parallel computing primitives onto graphics-centric hardware and programming interfaces. Nevertheless, this effort by HPC researchers often requires non-trivial software development in identifying appropriate software hooks given preset domain-specific APIs and programming interfaces for accelerators. However, the performance benefits demonstrated by researchers generate a feedback loop between supplemental-use users and accelerator developers which feeds improvements into the accelerator programming environment, helping support supplemental uses while prioritizing the primary-use. This has been demonstrated with the development of a general-purpose GPU computing ecosystem through programming frameworks such as CUDA and OpenCL together with domain libraries such as cuBLAS and cuDNN, while not neglecting the primary graphics APIs: OpenGL, DirectX, Metal, and Vulkan.

Supplemental Use of DNN Accelerators

DNN accelerators (sometimes called Neural Engines, Neural Processing Units, Tensor Processing Units, etc.) are a recent class of domain-specific accelerators used for the acceleration of DNN applications. As noted in Chapter 3, DNN accelerators have experienced a proliferation throughout the spectrum of computing hardware, including full-system supercomputing solutions, discrete PCIe accelerators, integrated SoC accelerators, and processor ISA extensions.

DNN accelerators include all the necessary components required to implement deep learning pipelines, including non-linearity functional units such as ReLU and sigmoid, pooling engines for reduction operations, and dedicated local memory systems to increase data re-use. However, at their core, DNN accelerators are centered around matrix engines which perform matrix-matrix computations such as matrix multiplication and 2D convolution. Since DNNs are not the only application which utilizes matrix-matrix computations, DNN accelerators are an attractive compute platform for supplemental-use applications. In fact, the scientific computing research community has been able to harness many of these matrix engines, that were designed to accelerate DNN workloads in GPUs and TPUs, for use in more traditional scientific computing application settings [110, 282, 69, 286, 287]. These efforts often involve end-to-end optimization of a single target application with dedicated manual tuning and explicit calls to the accelerator within the target application source code.

Prior work in the literature has surveyed the potential of tensor accelerators and matrix engines to act as an effective platform for accelerating HPC and supercomputing applications [75], with the goal of calibrating the level of investment the HPC community should invest in this type of hardware architecture. An analysis of the execution logs of a selected supercomputers found that approximately 53% of execution time was spent on GEMM operations. Additional analysis of software library dependencies within the Spack package manager (a package manager which targets supercomputers) found that approximately 50% of Spack packages depend on BLAS libraries (directly or indirectly). Finally, the analysis consisted of measuring the execution time of GEMM, BLAS and LAPACK operations within the SPEC benchmark suite [65] for general-purpose computing, and the Riken, TOP500, and Exascale Computing Project benchmark suites for high-performance computing. This part of the analysis identified that only 9 out of 77 benchmarks that were evaluated utilized a

GEMM or library-based dense linear algebra computation The analysis concludes that there is little reason for the HPC community to invest resources in embracing matrix engines due to diminishing returns and insufficient utilization of GEMM, but at the same time acknowledges that market forces may drive their proliferation and that the small fraction of overall improvement enabled by matrix engines is better than nothing for the right cost. While this analysis focused on HPC workloads and only focused on GEMM rather than additional use-cases of tensor and matrix engines, this conclusion aligns with the supplemental-use philosophy which takes into account economic considerations - "if it's there, we might as well use it".

Supplemental Use of DNN Accelerators on Edge SoCs

As mobile SoC architectures are adopted and integrated in larger compute systems with a broader set of workloads (for example, the Apple M1 SoC for laptops and desktop computers), the various accelerators on the SoC have the potential to be used by a broader set of applications. Laptops may run Matlab, R, Python, and other applications which are usually not found on smartphones and other mobile devices. In addition, increasing attention to data privacy concerns provides an incentive for applications to process data directly on edge devices as opposed to sending data to the cloud for compute-intensive processing.

The properties of accelerators on integrated edge device SoCs are vastly different than accelerator characteristics and behaviors in HPC environments. For example, while the majority of server-class GPUs can have a theoretical peak performance of $3\times$-$4\times$ than their host server-class CPUs [149], only 11% of smartphones have a GPU that is 3 times more performant than its CPU (based on theoretical peak) [281]. On the other hand, application-specific embedded SoCs may have this same ratio skewed in the opposite direction, with embedded GPUs capable of $100\times$ the theoretical peak performance compared to the host CPU on the SoC [112]. This wide range of accelerator/CPU performance ratios possible within SoCs, and its impact on high-performance software implementations, demonstrates the need for hybrid software/hardware exploration of this design space in edge SoCs.

Furthermore, energy efficiency takes priority over performance in most edge SoCs. Therefore, while HPC applications may perform futile work (multiply by 0 or 1) on some of the internal compute elements within matrix engines as part of an implementation achieving greater total utilization of the entire machine (i.e., using both GPU streaming multiprocessors and tensor cores to achieve faster total performance), edge devices may not be as tolerant to a large number of compute elements performing wasteful work while consuming energy and memory bandwidth. Hence, while HPC application implementations of non-GEMM-based algorithms can be inefficiently executed using subsets of compute elements within spatial matrix multiplication units and achieve greater performance than if they were run only using smaller vector/SIMD units [69], these use-cases may not be an appropriate choice from an energy-efficiency perspective on similar spatial matrix multiplication units on edge devices.

As a result, DNN accelerators on edge devices exhibit several fundamental architectural and micro-architectural characteristics which differentiate them from HPC DNN accelerators. These aspects need to be properly addressed by custom supplemental-use software. DNN accelerators on edge SoCs differ from their counterparts in the HPC setting in the following ways:

- Shared cache and memory hierarchy with other subsystems in the SoC;
- Dedicated controllers for increased energy efficiency;
- Numeric precision which meets application requirements while maintaining power efficiency;
- The size of the accelerator subsystem (in terms of number of arithmetic units and dedicated memory resources);
- Software ecosystem

These architectural and micro-architectural characteristics, together with the proliferation trends of integrated SoCs across more diverse platform classes, add an interesting aspect to analyzing the utility of domain specific accelerators on SoCs. DNN accelerators on SoCs were originally integrated with the primary intention of camera and image-processing related applications. These image processing applications, which experience tight latency and power constraints benefit for a dedicated accelerator which can perform compute-intensive DNN operations. However, like many other accelerators on mobile SoCs (multimedia encoders, encryption, etc.), these accelerators remain idle for a significant portion of the SoC operating time. Therefore, we would like to explore supplemental-use DNN accelerators in edge device SoC for the broader class of numerical data analysis applications in order to increase their utility on the SoC and provide a more efficient execution path for this emerging class of applications on mobile SoCs. This requires the exploration and customization of the relevant software stack to to the aforementioned unique properties of DNN accelerators on edge SoCs.

7.2 Software for Supplemental Use of DNN Accelerators for Numerical Data Analysis

DNN accelerators are associated with a complex software stack that has been designed to accommodate machine learning researchers using machine learning frameworks such as Tensorflow and PyTorch embedded in high-level programming environments such as Python. Deep learning models are implemented within such frameworks, which are built on top of a deep ecosystem of software components including graph interchange formats, graph compilers, optimized libraries, language runtimes, operating system platforms, and accelerators drivers, as illustrated in Figure 7.1.

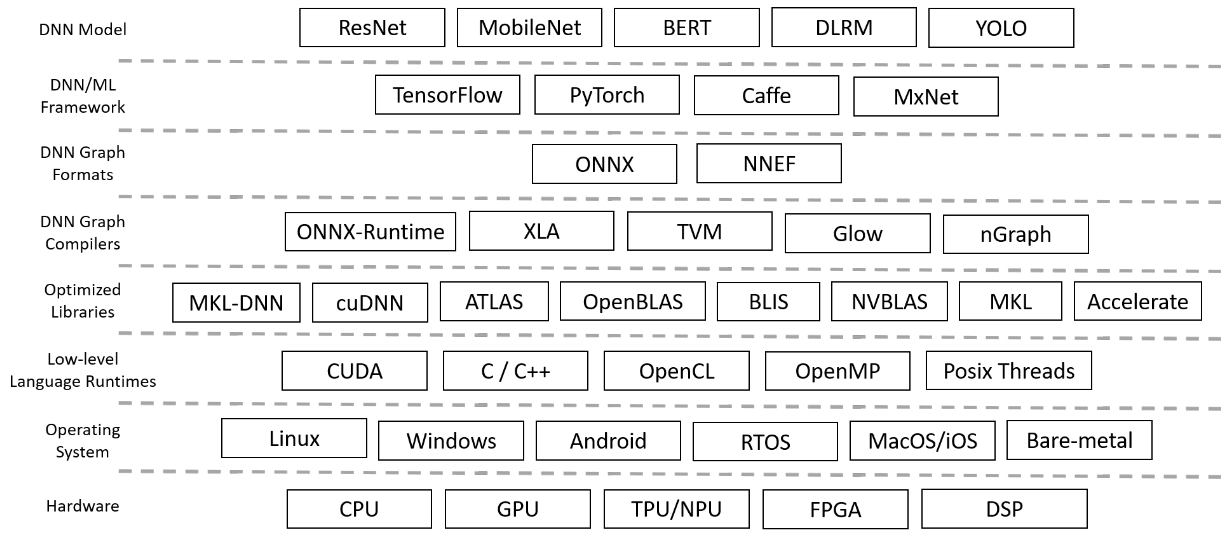


Figure 7.1: Deep learning software ecosystem.

This diverse software ecosystem provides many opportunities for hardware accelerators, but may also generate a swath of challenges. From an opportunity perspective, hardware accelerators have integration opportunities into different levels of the software stack. Some accelerators may choose to support only optimized libraries for subsets of kernels, other accelerators may prefer an end-to-end approach to code-optimization through integration with graph compilers, or a direct programming approach through intrinsic integration with the high-level machine learning framework.

Nevertheless, the challenges generated by this deep and diverse software ecosystem include mounting development resources required for the introduction of specialized components into the ecosystem both in hardware and software, as well as portability of applications and benchmarks across platforms. The creators of the MLPerf inference benchmark identify and analyze the challenges of benchmarking DNN accelerators through such a diverse software ecosystem [217]. Specifically, the challenge of providing a portable benchmark that can work across multiple accelerators and a cross-product of software ecosystem, while being able to attribute performance benefits to specific components through an "apples-to-apples" comparison. Similarly, recent work from Facebook observes that the high diversity of mobile SoCs results a large crossproduct of accelerator and processing units, which make it impossible for the software stack to be optimized for each particular SoC, often resulting in a preference for simple CPU-based application implementations [281].

An important building block in the development of the software industry has been the distinction between interfaces and implementations. While implementations of operating systems, networking, and hardware may change across vendors, interfaces such as system calls, networking protocols and instruction set architectures remain relatively stable. This enables decoupling of engineering efforts across organizations and projects, allowing for faster developments of individual implementations[1]. The deep learning ecosystem is relatively young, and as a result, there is no consensus yet regarding stable interfaces across layers of the deep learning software stack. At the same time, supplemental-use of accelerators must rely on standard interfaces, since the potential performance gains from supplemental-use acceleration are limited compared to primary-use acceleration, hence reducing the cost-effectiveness of extensive software development for supplemental-use.

In this chapter, we are interested in exploring the customization of the software stack for SoCs with DNN accelerators for numerical data analysis applications. As noted earlier, initial experiences in the HPC research community indicate that numerical computing and numerical data analysis are attractive supplemental-use applications for deep-learning accelerators. In contrast to the deep learning software stack, the numerical computing software stack is more mature, and relies on several decades of research and development in algorithms, programming languages, libraries and hardware. Numerical computing software can be used in a variety of computing environments, from supercomputers to client laptops and edge devices. Figure 7.2 illustrates important layers within the numerical computing software ecosystem. Notably, both the deep learning software stack and the numerical computing software stack have a diversity of optimized libraries from a variety of commercial and open-source software vendors. However, the deep learning stack does not present a unified interface to optimized deep learning libraries such as cuDNN and MKL-DNN (in addition to BLAS operations), while the numerical computing stack primarily converges to two interfaces derived from open-source projects: the BLAS interface and the LAPACK interface. As such, an implementation of these interfaces for accelerators can provide cost-effective supplemental-use, since a single implementation can be applied to a broad set of numerical computing applications which depend on this low-level interface implementation.

7.3 BLAS and LAPACK

While the matrix decompositions and numerical linear algebra algorithms which enable numerical data analysis can be implemented from scratch by high-level data analysis applications, numerical computing applications tend to rely on a set of core library implementations due to the nuances of floating-point computation and numerical analysis. Numerical stability, problem conditioning, and rounding behaviors are of vast importance in numerical computing, and often the algorithms that are taught in introductory linear algebra classes are not the algorithms that would be numerically stable for computation with finite floating-point precision. For example, while the normal equations are the most commonly taught solution for a least-squares problem, numerical computing libraries typically implement alternative solutions which rely on more stable matrix decomposition. Similarly, matrix inversions are often performed using LU decompositions rather than direct methods. Upon the selection of the appropriate numerical methods, additional algorithmic techniques such as blocking or

[1]This observation was recently repeated in a brief amici curiae by prominent computer scientist filed in the Supreme Court of the United States in the case of Google LLC. vs. Oracle America Inc. [184]

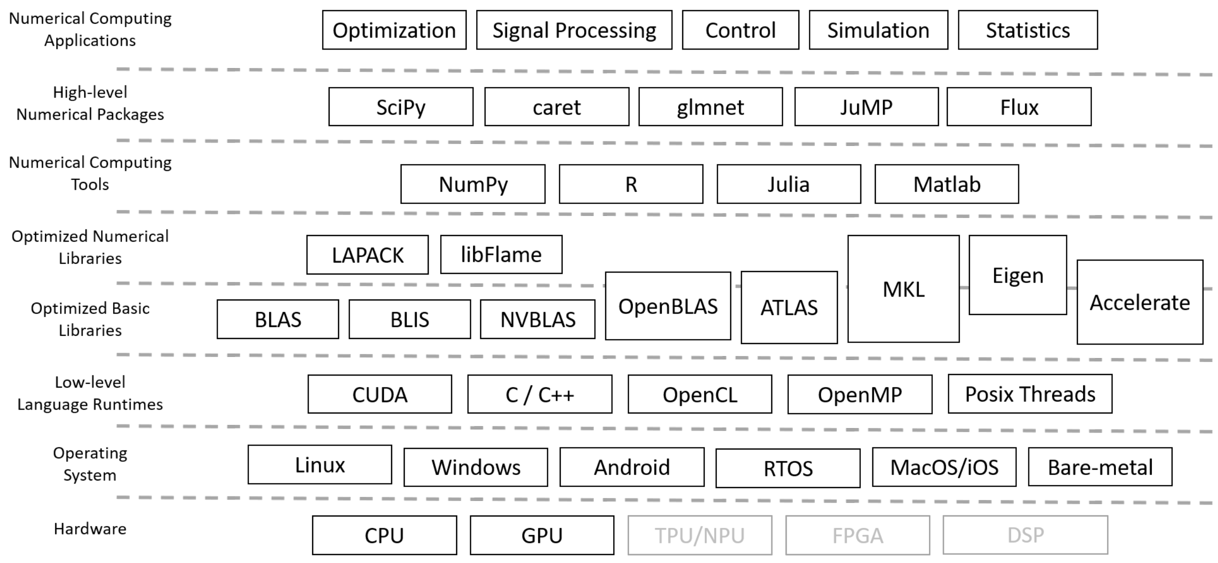


Figure 7.2: Numerical computing software ecosystem. (Note that numerical computing applications have traditionally not used FPGA or DSP hardware since those were often thought of as primarily fixed-point computing platforms.

recursive algorithms can be used to improve performance by reducing communication costs across the processor memory hierarchy.

The numerical computing software stack relies on standards and interface definitions such as BLAS and LAPACK which were set in the 1970s and 1980s [165, 17]. These interfaces are defined for single and double precision floating-point (IEEE-754) and real and complex datatypes. They have not been defined for low-precision and low-mixed-precision operations (there is a mixed-precision definition for double and quad precisions, which does not have many implementations [177]), which are now common in DNN accelerator matrix engines [148, 87]. While there are ongoing discussions and proposals for the extension of these standards and interfaces to low-precision and mixed precision, the current software stack will require deep modifications in order to support these behaviors.

The BLAS functions (**B**asic **L**inear **A**lgebra **S**ubroutines) were originally defined in the early 1970s with the goal of defining a set of basic functions which would enable "*portability* with *efficiency*" [165] when performing vector operations. The criteria for inclusion in the original BLAS package was that an operation should involve only one level of looping and occur in the usual numerical linear algebra algorithms such as Gaussian elimination or orthogonal transformations. They were later extended with BLAS-2 for matrix-vector operations [77] and BLAS-3 for matrix-matrix operations [76], with BLAS-3 holding the key property of a higher arithmetic intensity value than BLAS-1 and BLAS-2. As such, much focus has been dedicated over the years to the optimization of BLAS-3 operations through "blocking", and the improvement of algorithms to further use BLAS-3 operations rather than BLAS-2 or BLAS-1 operations in order to increase their arithmetic intensity. As noted, part of the original principles of the BLAS package definition were "*portability* with *efficiency*" [165]. It is not surprising that the original BLAS package was developed in the same years as the adoption of the "information hiding" concept in software engineering [207] and as the term Application Programming Interface (API) was coined in other computing fields such as graphics and databases, which also advocated for software library portability that can be used across applications [66, 71]

LAPACK is an open-source software library implementation for solving linear algebra problems such as exact and least square solutions for linear systems of equations, eigenvalue problems, and singular value problems [17]. LAPACK supersedes similar packages such as LINPACK and EISPACK which were used in supercomputers during the 1970s and 1980s. LAPACK improved performance and utilization of linear algebra operations by taking advantage of the hierarchical nature of memory systems on modern multiprocessors and reorganizing algorithms to use block matrix operations to better utilize those memory hierarchies. More importantly, the LAPACK routine nomenclature and ABI has been adopted as a de-facto standard for such optimized operations, and are being used by commercial vendors providing optimized numerical computing library implementations such as Intel MKL and Apple Accelerate. As a result, higher levels of the numerical computing software stack often rely on the open-source LAPACK ABI, hence enabling compatibility across multiple micro-architecture-specific optimized commercial vendor libraries. LAPACK has historically been designed with single-thread scalar CPUs in mind. Additional open-source implementations such as ScaLAPACK for distributed memory computers [54], Plasma for shared-memory multi-processors [78], and MAGMA for heterogeneous CPU/GPU systems [259], provide additional variants for more complex system architectures. Nevertheless, there variants often implement only a subset of the complete LAPACK collection of routines. LAPACK assumes the existence of an efficient BLAS implementation in the system in order to obtain high performance for blocked operations.

7.4 BLAS Implementations

As noted in Figures 7.1 and 7.2, there are multiple widely used implementations of the BLAS interface. In some cases, the BLAS implementation is packaged together with an implementation of the LAPACK interface (or a subset thereof). Implementations can generally be categorized as vendor-provided, closed-source, optimized implementations vs. open-source implementations targeting multiple platforms.

Prominent vendor-provided implementations include Intel MKL, Apple Accelerate and NVIDIA cuBLAS/NVBLAS. Commercial vendors invest significant software engineering resources in optimizing these implementations for each product generation and micro-architectural variant of their products. Since these optimizations often rely on intricate knowledge of product micro-architectures, these library implementations are closed-source, and cannot be modified to be used by alternative implementations.

Open-source implementations of the BLAS interface have been developed and maintained by the academic high-performance computing community since its origin. The Netlib BLAS reference implementation is an open-source reference implementation of the functions which compose the BLAS interface. This is a non-optimized Fortran implementation, in which all the operations are implemented in their trivial nested-loop form. Since it is written in Fortan, it can be compiled to any target architecture which has a supporting Fortran compiler. Using the Netlib reference BLAS implementation as a BLAS library will be functionally correct, but generally exhibit the lowest possible performance for target applications.

ATLAS (Automatically Tuned Linear Algebra Software) is an open-source implementation based on auto-tuning principles [274]. ATLAS relies on empirical experimentation during library build time in order to generate tuned implementations through a combination of code generation, parameter tuning, and composition of optimized kernels. ATLAS explores the search space of various tuning parameters and kernel routine compositions by measuring the empirical performance of a large number of potential implementations during library generation time. This provides ATLAS with a high degree of portability for a variety of CPU micro-architectures. In practice, ATLAS relies on architecture-specific kernel routines in order to make use of SIMD extensions and other architecture-specific properties.

OpenBLAS is an actively developed fork of the GotoBLAS project since GotoBLAS ceased development. It is based on the *Goto algorithm*, which uses a decomposition of operations into *"inner kernel"* routines in a block-panel (GEBP) structure, and emphasizes streaming data from the L2 cache rather than L1 caches [102, 103]. Kernel routines are manually developed and optimized for specific micro-architectures using software pipelining, loop unrolling, and SIMD extensions, and are often written using native assembly language.

BLIS (BLAS-like Library Instantiation Software) is another open-source implementation which is based on the principles of GotoBLAS. It is also based on the Goto algorithm, but uses a layered approach in order to increase code organization and re-usability. BLIS further re-factors the *"inner kernel"* of the Goto algorithm in terms of a smaller micro-kernel that requires less code to be optimized. We expand on BLIS later in this section.

In adherence with the principles of software customization for supplemental-use applications on accelerators, we would like to enable the broadest set of applications through a minimal but cost-effective software engineering investment. Table 7.1 compares BLAS library implementations with respect to properties of interest for our usage scenario of numerical data analysis workloads as supplemental-use applications. In particular, we focus on performance optimization, source-code organization and documentation, and the requirements for adding support for a new architecture and micro-architecture.

The Netlib reference library is the most flexible option, since it allows fine-grained replacement of individual functions and end-to-end optimization of each of those functions. However, this comes at a significant cost in terms of complete library performance or software development resources. Using the Netlib reference library as the base for library modification means that every function that will not be optimized and replaced will be as slow as the original non-optimized reference implementation. Hence, if we optimize only the `SGEMM` function, the `DGEMM` function will be as slow as the Netlib reference implementation. Similarly, the `STRSM` functions will also be as slow as the Netlib reference implementation. In

Property	BLIS	ATLAS	OpenBLAS	MKL	Netlib BLAS	Accelerate	NVBLAS
Open-Source	✓	✓	✓	✗	✓	✗	✗
Blocking-Optimized	✓	✓	✓	✓	✗	✓	✓
RISC-V Support	✗	✗	Partial	✗	✓	✗	✗
Source Code Documentation	✓	✓	✗	✗	✓	✗	✗
Organization	Arch-Isolated	Mono-lithic	Arch-Isolated	N/A	Mono-lithic	N/A	N/A
New Arch. Portability	✓	✓	✗	✗	✓	✗	✗
New Arch. Modification	Micro-kernels	kernels	kernels	N/A	None	N/A	N/A

Table 7.1: Comparison of BLAS library implementations.

fact, this was the approach that was used in the DNN case-study evaluation in Section 6.1, where we noted this same challenge when analyzing the results for batched DNN inference vs. non-batched DNN inference.

In contrast, modifying one of the open-source optimized libraries enables sufficiently good performance on kernels that are not specifically targeted by the accelerator. While open-source portable implementations such as ATLAS are able to generate a functional and relatively optimized CPU-based implementations for RISC-V architectures, these implementations will not be able to utilize custom instruction extensions or SoC accelerators without specialized kernel routines. Such routines are not trivial to add in ATLAS, since the framework is organized in a monolithic fashion and does not include source code documentation for adding a new architecture implementation. Furthermore, the ATLAS usage model is based on recursive algorithms for BLAS-3 kernels, which would require adding or modifying a variety of kernel routines in order to be able to efficiently utilize a throughput-oriented matrix-engine architecture.

Similarly, OpenBLAS would require the development of multiple BLAS-3 *"inner kernels"* for functions such as `GEMM`, `TRSM`, `TRMM`, `SYRK`, `SYR2K`, `HERK`, `HER2K` to use the accelerator. Optimizing each individual BLAS-3 kernel is a time-consuming task that may not be worth the cost for supplemental-use applications. However, a micro-kernel-based approach to BLAS-3 may be sufficient. Therefore, BLIS's use of the micro-kernel formulation was found to be a good fit with our target for customization for supplemental-use applications. BLIS additionally provides a high degree of source-code documentation and organization that is noticeably superior compared to alternative open-source BLAS implementations. These include detailed configuration files, templates, and guides on the topics of adding and optimizing new target architectures. Hence, we find that the BLIS library is the best apparent fit for generating an optimized BLAS library based on an SoC accelerator for numerical data analysis workloads as supplemental-use applications.

BLIS

BLIS is a portable template framework for generating high-performance BLAS-like dense linear algebra libraries [264, 265]. Originally developed at the University of Texas at Austin, it is based on the Goto algorithm [102] which is also the basis for additional popular BLAS implementations such as OpenBLAS and GotoBLAS.

A BLAS-3 implementation in BLIS relies on three microarchitecture-specific micro-kernel implementations: `GEMM`, `TRMS`, `GEMMTRSM`. Micro-kernels are designed to process the inner most loop of a blocked matrix multiplication using the Goto Algorithm, with micro-kernel sizes typically being determined by the size of the register file, factoring the number of registers and the size of the registers (especially when using wide SIMD registers). These micro-kernels should be optimized to obtain close to 100% utilization of the target micro-architecture on data resident in cache. Under this assumption, BLIS can generate the full set of BLAS-3 function implementations that are tiled based on the Goto algorithm, generally obtaining over 90% utilization on popular commercial CPUs.

Since supplemental-use applications target cost-effectiveness benefits, the ability to get a full API implementation which utilizes an accelerator is an attractive prospect. Together with the approach described in Section 6.3, we can use BLIS to generate highly tuned performance libraries for various accelerator configuration with a relatively small investment in software engineering resources. Nevertheless, we must also keep in mind that BLIS was designed for the CPU rather than SoC accelerators, and includes internal design choices which reflect that underlying assumption. We elaborate on these considerations in Section 7.5.

BLAS on SoCs

BLAS implementations have developed together with the evolution of computer processors, primarily targeting CPUs. However, as computer architectures are evolving with the end of Moore's law, we observe a progression of BLAS implementations for non-traditional computer architectures, albeit with various limitations and changes.

The cuBLAS library for NVIDIA GPUs supports the BLAS operations but assumes that data is already on the GPU device memory, which requires additional processing in the application code using additional custom CUDA code. Therefore, cuBLAS is does not in fact implement the BLAS ABI and cannot be used as a "drop-in" replacement in applications which use the BLAS interface. However, The NVBLAS library is such a library which implements the BLAS-3 ABI for NVIDIA GPUs. It performs tiled data transfers between the CPU and the GPU device, allowing it to be used transparently with applications using the legacy BLAS API. Similarly, the MAGMA project is an open-source project which targets BLAS implementations for heterogeneous CPU/GPU systems [259].

In edge devices, vendor-provided BLAS implementations have mostly focused on efficient utilization of SIMD and vector extensions within high-performance mobile CPUs. To our knowledge, mobile device BLAS implementations generally do not utilize the various accelerators found on the SoC, but rather focus on vector ISA extensions that are tightly integrated with the CPU [276, 131]. Similarly, BLAS implementations for embedded processors are also primarily CPU-centric [91, 252], with additional recent efforts towards CUDA-based implementations for embedded GPUs [112].

Edge device SoCs provide an interesting platform for BLAS implementations due to their heterogeneity of compute units, complex memory system composed of many subsystems with private and shared memories, the limited size of computational units compared to dedicated accelerators such as GPUs, and a software ecosystem that is typically distinct to each individual device platform.

In order to evaluate the software customization required to use BLAS for numerical data analysis applications using DNN accelerators in a general-purpose SoC environment, we use Gemmini and Hwacha within a Chipyard-based SoC. Unlike a large part of SoC-integrated accelerators, the co-processors studied in this work are tightly integrated with the core, and do not require system calls or other long-latency drivers in order to execute their custom instructions. In this sense, they exhibit similarities to the Intel AMX ISA extensions and the Apple AMX CPU extensions, as reported by the Linley Group and additional researchers [144, 82, 200].

7.5 BLAS/BLIS for Gemmini and Hwacha

We implement and customize a new micro-architecture target within the BLIS library[2] designed to use a mixed-precision floating-point configuration of the Gemmini DNN accelerator, together with the Hwacha vector unit. Hwacha, as a general-purpose data-parallel vector unit, is used to implement BLAS-1 and BLAS-2 operations which are primarily element-wise operations and do not exhibit data re-use. Gemmini, as a matrix accelerator, would be the key for the implementation of BLAS-3 operations. BLAS-3 operations in BLIS are based on the implementation of two micro-kernels: `GEMM` and either `TRSM` (triangular matrix solve) or `GEMMTRSM` (`GEMMTRSM` is a micro-kernel which fuses the `GEMM` and `TRSM` micro-kernels). These micro-kernels implement the inner-most loop of the Goto/BLIS algorithm and are used to compose higher-level tiled BLAS-3 operations. In addition, a custom `GEMM` kernel implementation that does not utilize the Goto/BLIS algorithm can be used in the form of a "sandbox". However, this "sandbox" cannot be used for the other BLAS-3 kernels such as `TRMM`, `TRSM`, etc. We use the reference SoC configuration illustrated in Figure 7.3, which uses a high-performance BOOM 3-wide out-of-order host CPU with both Hwacha and Gemmini as RoCC accelerators. The particular configurations of the accelerators will be varied

[2]This new micro-architecture target is maintained in a fork of the BLIS code repository, since it is not a commercial silicon product that can be used by additional BLIS users.

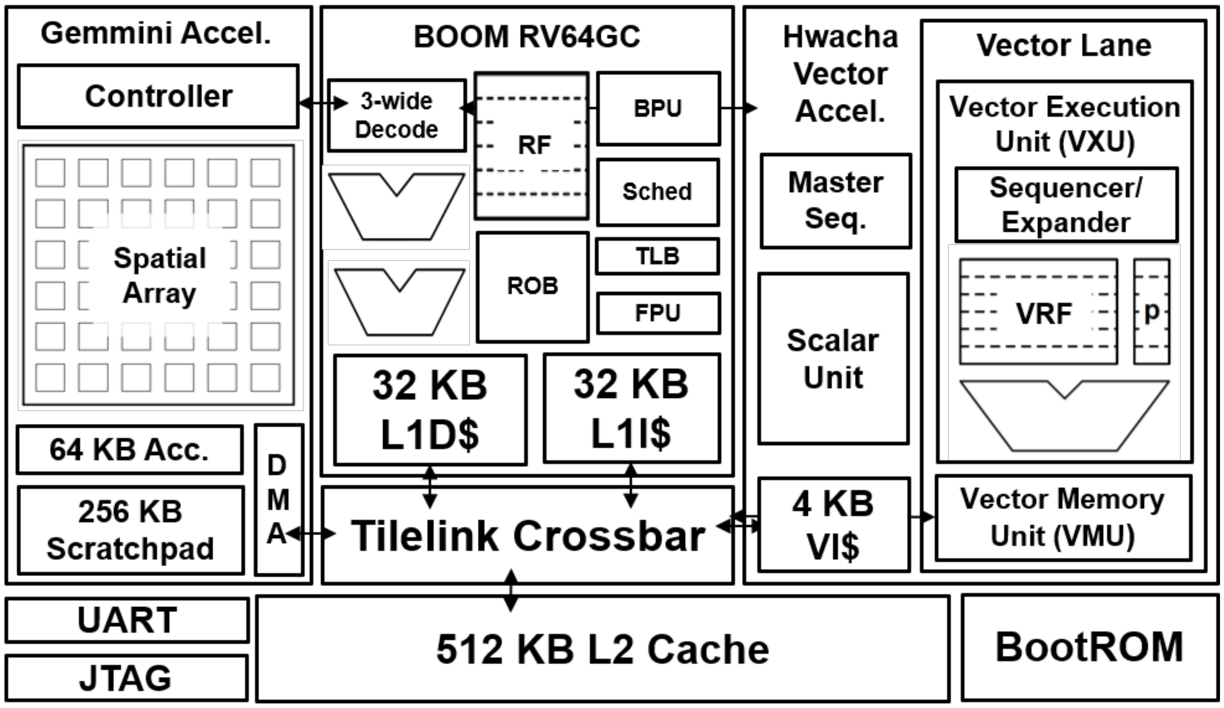


Figure 7.3: Reference SoC configuration for custom BLAS/BLIS implementation using Gemmini and Hwacha.

throughout this chapter for exploration of various hardware design points and software design choices.

There are several implementation considerations when implementing a BLIS micro-architecture to use a custom SoC with Hwacha and Gemmini, which are generally applicable to supplemental-use of other DNN accelerators. Figure 7.4 illustrates these primary areas within the illustration of the BLIS algorithm. The following sections elaborate on these considerations and trade-offs.

Numerical Precision

The BLAS and LAPACK interfaces are defined for single- and double-precision floating-point and real and complex datatypes. They have not been defined for low-precision and low-mixed-precision operations, which are now common in DNN accelerator matrix engines (as noted in Section 7.3, there is a mixed-precision BLAS definition for double and quad precisions, which does not have many implementations [177]). While there are ongoing discussions and proposals for the extensions of these standards and interfaces to low-precision and mixed precision, the current software stack will require deep modifications in order to support these behaviors.

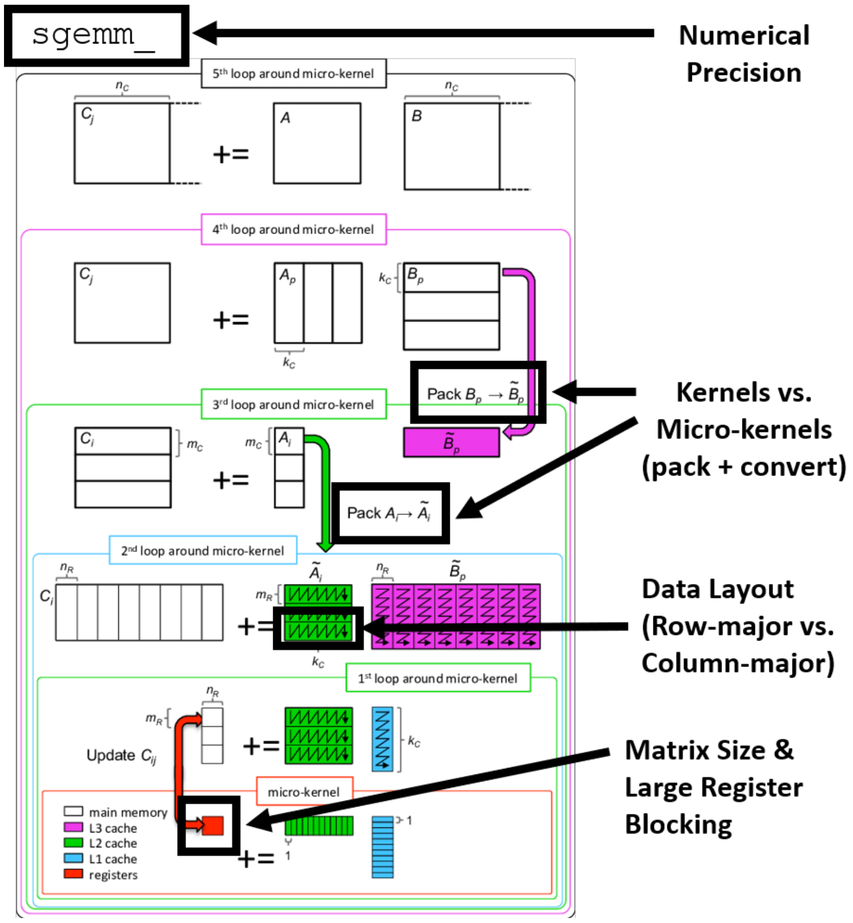

sgemm_
Numerical Precision
5th loop around micro-kernel
4th loop around micro-kernel
Pack $B_p \rightarrow \tilde{B}_p$
Kernels vs. Micro-kernels (pack + convert)
3rd loop around micro-kernel
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
Data Layout (Row-major vs. Column-major)
1st loop around micro-kernel
Update C_{ij}
Matrix Size & Large Register Blocking
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers

Figure 7.4: Areas of special attention within the BLIS algorithm for the target micro-architecture using Gemmini and Hwacha (Base BLIS algorithm illustration reproduced with permission from [263]).

Full-precision (IEEE-754 single-precision, double-precision) hardware floating-point operations are long-latency and energy-consuming in comparison to fixed-point integer or low-precision arithmetic operations. In-fact, in our experience with the implementation of the Hwacha vector unit, the critical paths of a design often pass through the logic of the floating-point unit [173]. Recent deep learning hardware accelerators have focused on reduced precision thanks to the high tolerance of deep neural networks to computation precision errors. This tolerance to low precision has been utilized in hardware accelerators in both fixed-point and floating-point forms. While the Gemmini generator can generate accelerators for any types of fixed-point or floating-point representation, a single-precision floating-point matrix unit consumes significant area and power due to the large number of wide multipliers and adders. This overhead was evaluated in a Gemmini accelerator with an 8×8 spatial array using an educational predictive process technology for 8-bit fixed point integer, 16-bit low precision floating-point, and 32-bit single-precision floating-point. It was observed that the area of the spatial compute array for 16-bit low-precision floating-point is $8\times -9.2\times$ (for IEEE-754 FP16 and bfloat16 respectively) larger than an 8-bit integer spatial array, and that the full single-precision spatial array is $2.6\times -3\times$ larger that the low-precision floating-point spatial array. These results are supplemented by data from Google which observes a $1.5\times$ energy advantage of bfloat16 over IEEE-754 FP16 [201].

Similar considerations in the choice of arithmetic numerical precision are evident in commercial deep learning accelerators, of which only a minority implement single-precision floating-point matrix multiplication in hardware. Table 7.2 lists the numerical precision support of a sample of commercial DNN accelerators. As can be seen in Table 7.2, many of the recent inference deep learning accelerators on the market for edge devices utilize 8-bit fixed-point integer precision. Deep learning accelerators which support training also support various floating-point precisions, with a particular focus on 16-bit floating-point formats such as bfloat16 (BF16) or IEEE 754 half-precision floating-point (FP16).

	Matrix Multiplication Accelerator Numerics						
	Int4	Int8	Int16	fp16	bf16	fp32	tf32[3]
NVIDIA Volta TensorCore	✓	✓		✓			
NVIDIA Ampere TensorCore	✓	✓	✓	✓	✓	✓	✓
Google TPUv1		✓					
Google TPUv2					✓		
Google TPUv3					✓		
Intel AMX		✓			✓		
AWS Inferentia		✓		✓	✓		
Qualcomm Hexagon [4]		✓					
Huawei Da Vinci [5]		✓		✓			
MediaTek APU 3.0		✓	✓	✓			
NVIDIA DLA [6]		✓	✓	✓			
Samsung NPU [7]		✓					
Tesla NPU		✓					

Table 7.2: Numerical precision support in commercial matrix multiplication accelerators. Information based on [129, 140, 149, 148]

As of the time of writing, only a subset of edge devices support floating-point precisions in their DNN accelerators. However, the onset of data privacy concerns and a desire to support a broader diversity of DNN models are driving additional accelerators on edge device SoCs to implement low precision floating-point representations which allow greater support for local tuning and model execution while maintaining manageable area and power budgets. In conjunction with the use of mobile SoCs in laptops, desktops, and other personal computing devices (for example, the Apple M1 SoC), an increase in adoption of low-precision floating-point matrix multiplication acceleration in edge devices can be expected.

Unlike 32-bit and 64-bit floating-point formats which are standardized across a wide range of applications through the IEEE-754 standard, 16-bit floating-point formats are often defined based on the needs of particular applications due to tradeoffs between precision and range within their limited encoding space. The origin of the IEEE-754 FP16 is in graphics rendering, where a ratio of 10 mantissa bits and 5 exponent bits was found to be sufficient for most shaders [210, 2] (while being insufficient for computing derivatives and difference-based algorithms). In contrast, the recent adoption of bfloat16 ("Brain float") for machine learning applications was due to the insufficient range of IEEE-754 FP16 causing frequent overflows [148]. In order to increase range, the bfloat16 format maintains the same number of exponent bits as the FP32 format. This means that the number of mantissa (precision) bits is reduced, but this also has the advantage of simplifying the conversion between FP32 and bfloat16 representations (since only a simple truncation or rounding of the mantissa bits is required) [150, 148, 33]. For these reasons, and based on the analysis in Table 7.2, the reference Gemmini evaluation platform that will be used in this chapter will use a mixed-precision Gemmini configuration with bfloat16 operands and single-precision accumulation, as is the case for many deep learning accelerators which support low-precision floating-point [148, 204, 87, 140].

From numerical analysis, we know that the numerical error E of a floating-point matrix multiplication $A \cdot B$ is bounded by $|E| \leq n \cdot \epsilon \cdot |A| \cdot |B|$, where ϵ is the machine error of the floating-point format, and n is the dimension of the matrix. For a block matrix multiplication with mixed precision accumulation, such as the one used in DNN accelerators with bfloat16 operands and FP32 accumulation, this error bound changes depending on the internal rounding implementation of the blocked matrix multiplication. Blanchard et al. [37] show that this can range from $|E| \leq (n+2)\epsilon_{bf16} \cdot |A||B|$ in cases where rounding is based on the lower precision after each fused multiply-add (FMA) operation, to $|E| \leq 2\epsilon_{bf16} + n \cdot \epsilon_{fp32} \cdot |A||B|$ in cases where rounding is based on the higher precision after each individual FMA. While many DNN accelerators use partial floating-point computation implementations which do not fully conform to any particular floating-point computation standard (with respect to rounding modes and overflow/underflow behavior), Gemmini uses the *Berkeley Hardfloat* generator for hardware floating-point computation, which is a parameterized implementation of floating-point computation based on the IEEE-754 standard [117]. This implementation includes IEEE-754 compliant rounding procedures and provides a selection of rounding modes supported by the IEEE-754 standard. As such, Gemmini demonstrates similar numerical properties to those observed in NVIDIA tensor-cores [87]. We note that the machine roundoff error of bfloat16 is $\epsilon_{bf16} = 3.91 \times 10^{-3}$, while for comparison purposes single-precision floating-point machine unit roundoff is $\epsilon_{fp32} = 5.96 \times 10^{-8}$ and IEEE-754 half precision is $\epsilon_{bf16} = 4.88 \times 10^{-4}$. Hence, for datasets in which normalized data precision can be differentiated by less than 2-3 decimal digits, simple-to-use low precision computation can be utilized to increase performance.

For the vast majority of numerical applications, which are not as precision-tolerant as deep neural networks, 16-bit floating-point computation in itself does not provide sufficient accuracy of results. Mixed precision computation, which combines 16-bit, 32-bit, and 64-

bit floating-point formats at different parts of an algorithm, have been shown to provide proper mitigation techniques to obtain a desired computation accuracy for a core number of application kernels, including linear system solutions and least squares solutions [110, 124, 111, 44, 45, 123]. These mitigation techniques have been mostly experimented with and utilized in the realm of high-performance scientific computing, since programmable reduced-precision floating-point hardware with explicit programming models is found mostly in cloud and high-performance computing processors such as TPUs, GPUs, and the A64FX ARM processor.

Numerical modeling and simulation applications, which are prevalent in scientific and high-performance computing, indeed have stringent precision and accuracy requirements for their numerical methods. At the same time, data analysis applications may often be limited by input data precision rather than numerical and computational precision. We observe that some data analysis applications and frameworks use high-precision floating-point representations due to default configurations or legacy libraries, rather than explicit numerical analysis reasons. For example, the R statistical analysis framework uses only double-precision floating-point to represent its `numeric` floating-point datatype. There is no single-precision floating-point datatype in the base R implementation. This design choice does not have much noticeable consequences on CPU-based execution which uses a single hardware FPU to process both single-precision and double-precision floating-point arithmetic. However, in parallel computing platforms with many floating-point execution units such as GPUs, vector units, or tensor engines, lower precision floating-point implementations can have significant throughput and energy implications.

In data analysis cases where accuracy is dominated by data precision rather than the numerical backwards error of the computation, it is clear that high absolute computation precision is an over-provisioning of resources. Further, backwards stable algorithms provide strong confidence that a small error in the input data generates a small error in the output results. Hence, if the backwards error is smaller than the uncertainty in the data (due to sensor precision, sampling noise, etc.), then relaxed computational precision can be accommodated by data scientists. Since the error bounds for traditional linear algebra operations are relatively well established, data scientists can use their domain knowledge about their datasets to make this determination. A recent analysis of LAPACK, the widely used numerical linear algebra library which forms the basis for a major portion of the numerical computing software stack, observed that the vast majority of LAPACK routines are numerically backwards stable with a bfloat16 BLAS implementation.

The traditional approach for using low-precision floating-point hardware within the high-performance computing community has primarily relied on explicitly replacing single-precision floating-point operations with low-precision floating-point operations only in appropriate places within the application codebase after a careful analysis, in order to maintain tight control over the numerical properties of the application [221, 39]. Similarly, use of mixed-precision floating-point computation has also been based on explicit function calls which precisely define the precision of each operand and the precision of the results. This approach can result in an explosion of function variants within the definition of mixed-precision BLAS (e.g. `sssdgemm`, `sdsdgemm`, `ssddgemm`, `dssdgemm`, `sdssgemm`, etc.) [177]. However, under an assumption of well-conditioned problems and numerical backwards stability of the core linear algebra algorithms using accelerators, we hypothesize that coarser-grained approaches may provide similar benefits of accurately using low-precision hardware, while providing a simpler programming model.

We postulate that a non-negligible number of applications which use single-precision floating-point do so because it is the default precision (or the minimal standard precision under IEEE-754), rather than because those applications have specific accuracy requirements that necessitate single-precision accuracy. The alternative approach we use in our implementation of the BLIS library is a coarser-grained solution which uses a system environment variable to enable relaxed numerics. By enabling this POSIX system-wide environment variable, we allow data scientists to indicate to the system that they are working with a low-precision and well-conditioned dataset that is tolerant of low precision computation. This information enables the underlying library implementations to transparently cast single-precision floating-point operations to low-precision floating-point operations when it may be advantageous for performance. By transparently casting single-precision floating-point level-3 BLAS operations to low-precision floating-point operations, it may be possible to improve the performance of a class of applications which use traditional numerical computing libraries primarily for their performance advantage properties rather than their numerical accuracy properties. In contrast to the mixed-precision algorithms which use single- and double- precision iterative refinement to compensate for a low-precision compute-intensive kernel, the approach presented here places a larger burden on the data scientist to be familiar enough with their dataset to allow for computation with relaxed numerical precision.

In practice, when the data scientist is confident their dataset is tolerant of low-precision computation, they will enable a POSIX environment variable named `RELAXED_NUMERICS`, which enables a codepath within our BLIS implementation which registers Gemmini-based functions as the implementation of BLAS-3 "single-precision" kernels. If the `RELAXED_NUMERICS` environment variable is not enabled, our BLIS implementation will register single-precision Hwacha-based implementations as the default BLAS-3 single-precision kernels. The relaxed numerics environment variable allows us to maintain the single-precision floating-point BLAS-3 API (i.e., `sgemm_`, `strsm_`, `strmm_`, `ssyrk_`, `ssymm_`, etc.) in order to match the large corpus of applications which expect ABI compatibility to this API. Figure 7.5 the usage flow of the custom BLIS implementation using the `RELAXED_NUMERICS` environment variable. Since the interface expects single-precision operand matrices A and B, this requires converting the operand to bfloat16 precision before performing a computation using Gemmini. The result matrix C does not require any conversion, since accumulation is performed in single-precision in Gemmini. This is a coarser-grained approach compared to traditional HPC low-precision methods, since it dictates that all single-precision BLAS-3 operations within the application will be performed in low-precision when the relaxed numerics flag is enabled. We hypothesize that this coarser-grained approach is advantageous in terms of usability, and in terms of the opportunity for data scientist to apply their domain knowledge of the dataset without needs to explicitly analyze each arithmetic operation in the under-

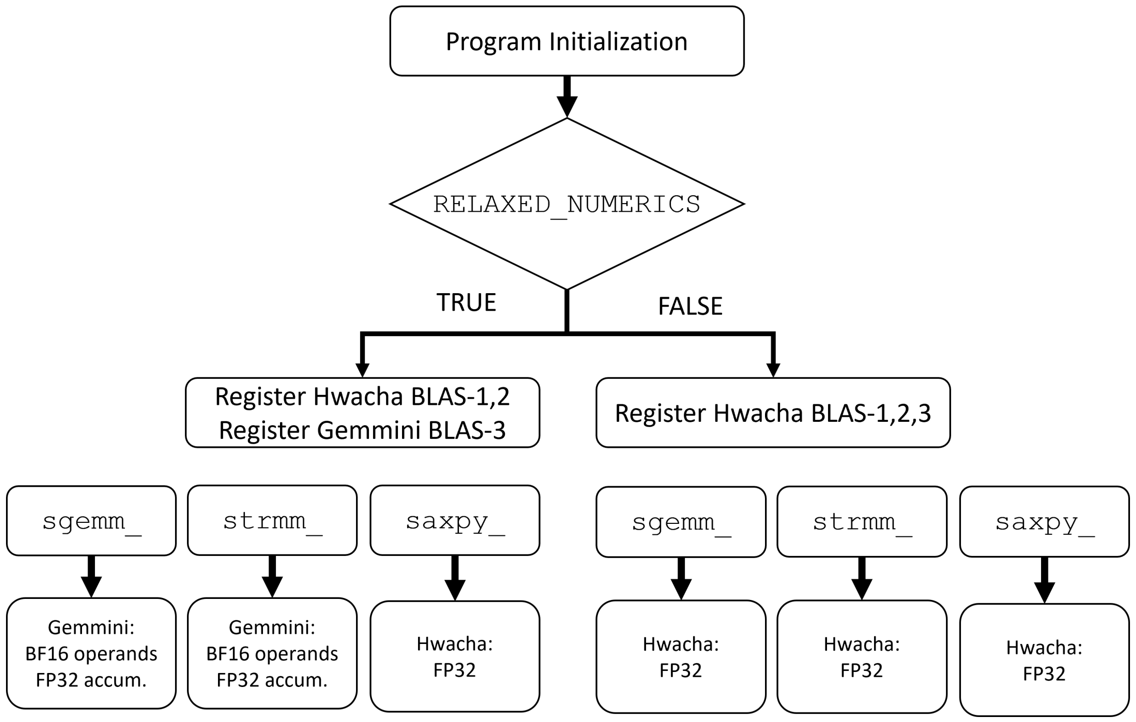


Figure 7.5: Usage flow diagram of the RELAXED_NUMERICS POSIX environment variable as coarse-grained control over low-mixed-precision computation within the legacy BLAS interface. When the RELAXED_NUMERICS environment variable is enabled by the data scientist, Gemmini-based BLAS-3 functions will be registered by the BLIS runtime, and legacy single-precision BLAS-3 API calls will be transparently converted to low-mixed-precision. BLAS-1 and BLAS-2 function will continue to run in full single-precision on the Hwacha vector unit. When the RELAXED_NUMERICS environment variable is disabled, all single-precision BLAS functions will run on the Hwacha vector unit.

lying codebase. As user interface studies have shown in the past, streamlining the usage of a feature can increase its usage, and we hypothesize that by streamlining low-precision capabilities into only a single switch required to enable this property, we are increasing the chance it will be used in practice.

Micro-kernels vs. Kernels

BLIS (and several other BLAS implementations) are designed around optimized kernels and micro-kernels. By optimizing small micro-kernels and scheduling those micro-kernels using empirical tuning or algorithms such as the BLIS/Goto algorithms, such BLAS implementations enable simple re-targeting of the implementation to different micro-architectures through template-based programming and software scheduling while maintaining high performance. Nevertheless, this software organization design choice is generally made under

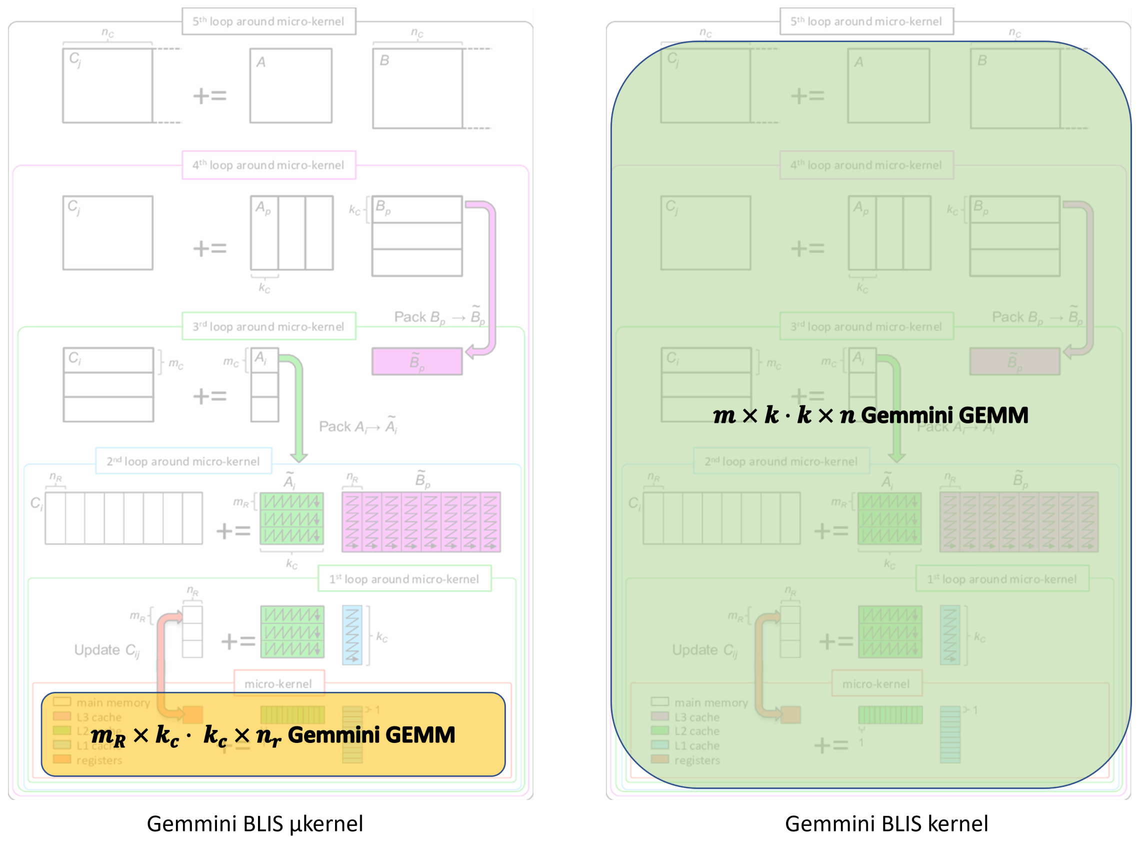

5th loop around micro-kernel
4th loop around micro-kernel
3rd loop around micro-kernel
2nd loop around micro-kernel
1st loop around micro-kernel
micro-kernel
Pack $B_p \rightarrow \tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
Update C_{ij}
main memory
L3 cache
L2 cache
L1 cache
registers
$m_R \times k_c \cdot k_c \times n_r$ Gemmini GEMM
$m \times k \cdot k \times n$ Gemmini GEMM
Gemmini BLIS µkernel
Gemmini BLIS kernel

Figure 7.6: BLIS kernels vs. microkernels and their projection on Gemmini. Kernels operate on complete matrices, enabling the underlying accelerator to utilize throughput-oriented optimization (the `GEMM` kernel can be exposed using a BLIS sandbox). Microkernels operate on packed register- and cache-blocked panels, and can be utilized for a wider variety of kernels. (Base BLIS algorithm illustration reproduced with permission from [263]).

the assumption that implementations would primarily target CPUs, which are based on scalar units or packed SIMD units optimized for low-latency operations. Processors which take advantage of data-parallelism to provide high throughput and hardware utilization, at the cost of latency, may exhibit only partial benefits from the micro-kernel-based software architecture.

Micro-kernels are used for BLAS-2 and BLAS-3 operations, and require a fixed size for at least one of the dimensions of each matrix operand (which we refer to as the "size" of the micro-kernel). Both Hwacha and Gemmini rely on latency-hiding through internal re-use and hardware sequencing in order to achieve high throughput and high utilization of their respective arithmetic and memory resources. By having multiple operations in-flight at the same time, the Hwacha and Gemmini sequencers use data parallel abstractions to implement latency-hiding techniques such as systolic execution, double buffering, and decoupled access-execute to maintain their execution pipelines highly utilized. Hwacha is a temporal vector machine, which achieves high utilization by hiding the latency of memory accesses using a deep pipeline and long vector lengths. As a temporal vector machine, in order to achieve high utilization it requires high re-use of data in vector registers, or vectors long enough to hide memory access latency from the SoC L2 cache or DRAM. This would typically require vectors of lengths greater than 16 or 32 elements. The Gemmini accelerator similarly relies on data re-use within the private scratchpad memory in order achieve high utilization of the compute array. The Gemmini hardware controller in-fact uses the private scratchpad as a register file when it schedules DMA transactions and compute operations using its hardware schedulers. However, if there is insufficient data-reuse due to the size or shape of the matrix operands, the Gemmini hardware scheduler will not be able to hide the latency of memory accesses.

The throughput-oriented latency-hiding approaches implemented in Hwacha and Gemmini do not perfectly align with the micro-kernel software principles used in BLIS and the Goto algorithm, since micro-kernels are intended to be limited in size, hence conflicting with the fundamental assumptions of throughout-oriented hardware which requires sufficient data-parallelism to enable latency-hiding. The micro-kernel approach relies on small matrix operations that can be composed into larger matrix operations using software algorithms. As such, they are a good fit for low-latency spatial packed-SIMD vector ISA implementations as opposed to temporal vector machine implementations. Similarly, while fine-grained Gemmini instructions which directly perform Gemmini micro-operations could be a better fit for programming a micro-kernel compared to Gemmini's coarse-grained instructions, the instruction issue bandwidth of the host CPU is often insufficient for the accelerator to achieve peak utilization using only fine-grained instructions. The micro-kernel's fixed size exposes a tradeoff with latency-hiding hardware, since large micro-kernel sizes which benefit Hwacha and Gemmini's latency-hiding requirements can create overheads for small matrices, while small micro-kernel sizes could result in lower utilization of Hwacha and Gemmini.

We demonstrate and evaluate this tradeoff (also illustrated in Figure 7.6) using the "sandbox" feature in BLIS. The "sandbox" bypasses the BLIS micro-kernels and allows developers to directly implement the `GEMM` kernel rather than implementing a micro-kernel and relying on the BLIS algorithm. This feature is intended to allow developer to explore optimal `GEMM` software algorithms for different target hardware architectures. The Gemmini matrix multiplication operation can be used both as part of a BLIS micro-kernel, as well as an independent BLIS `GEMM` kernel operation through the aforementioned "sandbox". We restrict the discussion regarding kernels vs. micro-kernels to the `GEMM` kernel, since the developer productivity benefits of micro-kernel-based templates for the remaining BLAS-3 kernels (`TRSM`, `TRMM`,`SYMM`, `SYRK`, etc.) outweigh the potential utilization penalty of the micro-kernel-based approach within the context of high-diversity SoC micro-architectures and supplemental-use applications.

Figure 7.7 illustrates the performance of a BLIS-based `GEMM` operation across various square matrix sizes using a 1 GHz SoC (single clock domain) with a DDR3 backing memory. It evaluates different SoC hardware configurations as well as software implemented using the BLIS `GEMM` micro-kernel vs. "sandbox". Notably, the "Sandbox loops" variants, which use

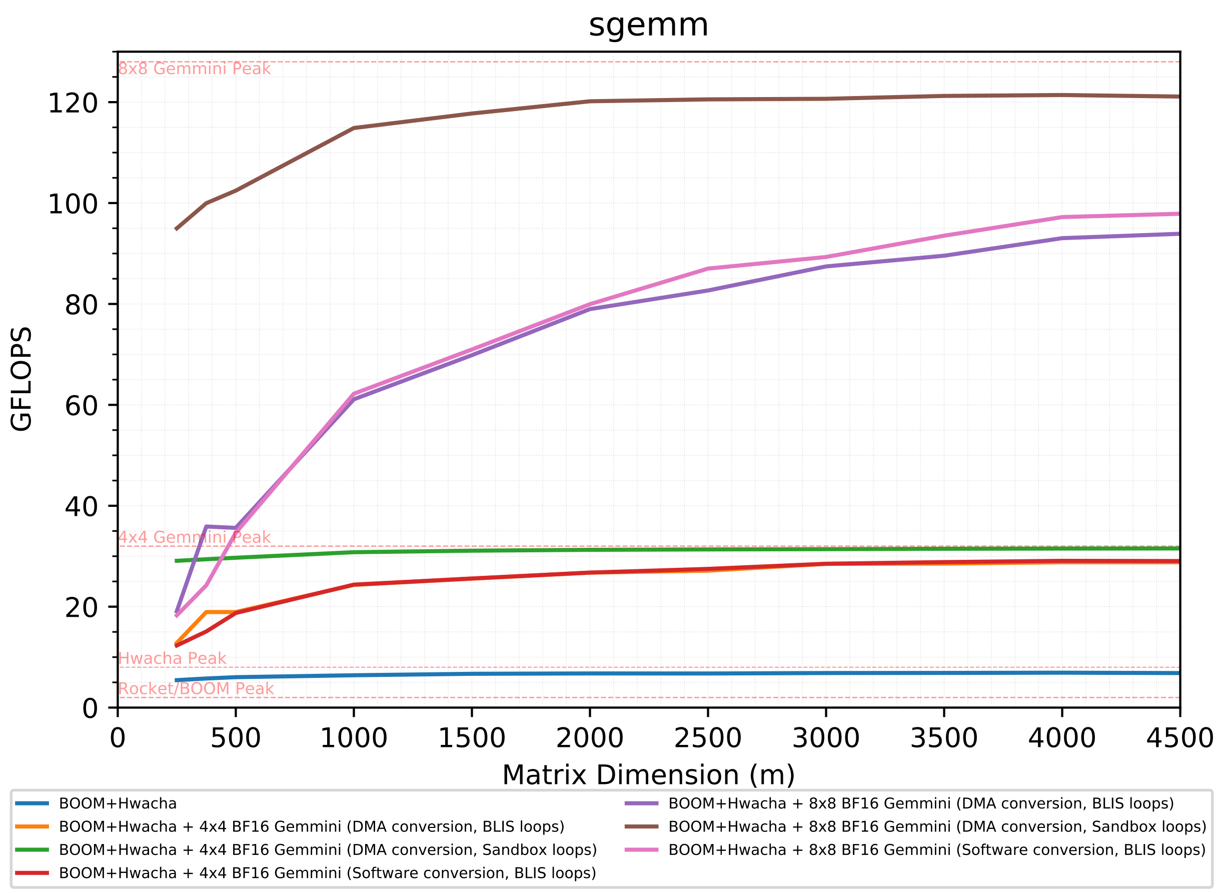


Figure 7.7: `SGEMM` kernel performance on square matrices using a 1 GHz SoC with DDR3 backing memory.

Gemmini's native SDK to implement the BLIS kernel, bypassing the BLIS micro-kernel-based implementation, outperform the micro-kernel-based implementations and attain significantly higher utilization of compute resources on the SoC. This holds true across both 4x4 and 8x8 configurations of Gemmini. Nevertheless, the "sandbox" kernel-centric approach comes with several caveats since Gemmini is primarily designed as a deep learning accelerator rather than a general-purpose BLAS accelerator. Gemmini's native SDK is able to support the `GEMM` operation natively since it is also used extensively used in DNNs, and therefore supported and optimized by the accelerator. However, other BLAS-3 kernels (`TRMM`, `SYRK`, `TRSM`, `SYMM`, etc.) are not used by DNNs, and therefore do not have such native support in Gemmini. Therefore, other BLAS-3 operations continue to be implemented using BLIS micro-kernels since they will not benefit as much from a direct kernel "sandbox" implementation as they are not able to utilize such native Gemmini features including hardware scheduling and hardware zero-padding. Additionally, the use-case evaluated in Figure 7.7 used a matrix layout which uses unit column stride. Use-cases with more complex data layouts (non-unit stride, mixes of column-major and row-major) could pose a challenge the Gemmini's native SDK. Furthermore, the usage of Gemmini's native SDK in the BLIS sandbox assumes that data is converted from single-precision to bfloat16 representations in hardware by the accelerator DMA. This assumption has implications on the micro-kernel-based implementation of the remaining BLAS-3 kernels. We therefore further elaborate on both these topics (with the latter one specifically discussed within the context of micro-kernels).

Data precision conversion

An important principle of our approach to numerical precision is maintaining the legacy single-precision BLAS interface in order to transparently enable the legacy software stack. As a result, this requires converting the operand matrices A and B from single-precision to bfloat16 precision before performing a computation using Gemmini, since we are using the single-precision BLAS interface which expects single-precision operand matrices A and B. The result matrix, C, does not require any conversion, since accumulation is performed in single-precision in our Gemmini configuration (similar to other low-precision deep learning accelerators). Conversion between single-precision floating-point operands to bfloat16 operands can be performed either in Gemmini hardware or in "software" (using Hwacha or BOOM), as also illustrated in Figure 7.8:

1. "On-the-fly" as part of the accelerator DMA transaction: Reading 32-bit single precision values from shared main memory, and writing bfloat16 values to the Gemmini private scratchpad

2. "Software conversion" using the CPU vector unit: The vector unit will read 32-bit values into the vector register file, perform a conversion operation, and then write bfloat16 values back to shared (cached) memory. The accelerator DMA will then read bfloat16 values from shared (cached) memory and write them to the private scratchpad.

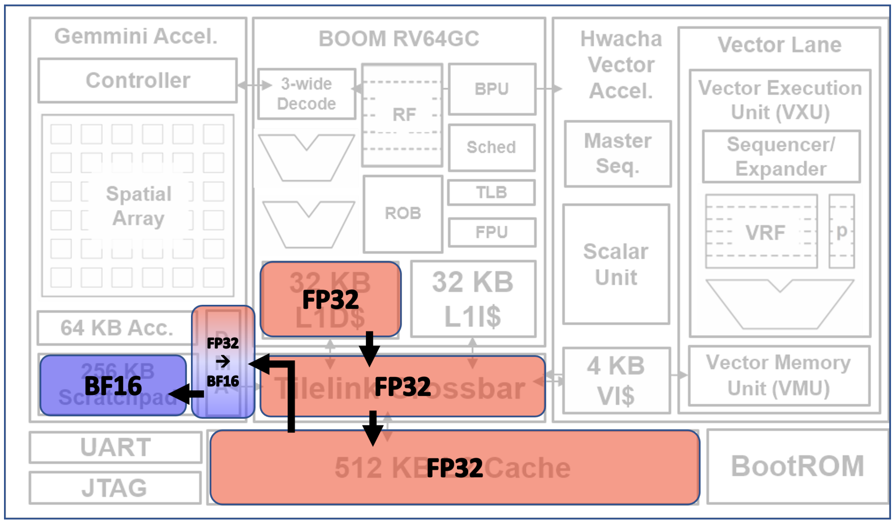
Both approaches present trade-offs: The "on-the-fly" approach is "wasting" shared-cache capacity and memory bandwidth between shared memory and the accelerator private scratchpad. This is because the accelerator DMA accesses 32-bit values from shared memory, but uses only 16-bits of them, essentially halving the effective bandwidth of the accelerator DMA. Nevertheless, if the arithmetic intensity of the operation is high enough, then this loss of effective bandwidth will not be felt by the accelerator since it will be hidden by data re-use within the private scratchpad.

In contrast, software conversion using the vector-unit carries the cost of additional communication (and additional memory) through shared memory between the vector unit and the Gemmini accelerator, and therefore necessitates cache blocking. Furthermore, while the vector unit and the matrix engine can run independently, potentially overlapping their operation, they still require a synchronization mechanism in order to maintain sequential consistency due to a lack of unified memory management unit. Therefore, in this approach, a `fence` needs to be used after the conversion of each BLIS packed-panel, which adds a burden of synchronization and prevents full-throughput overlap between the vector unit and matrix engine.

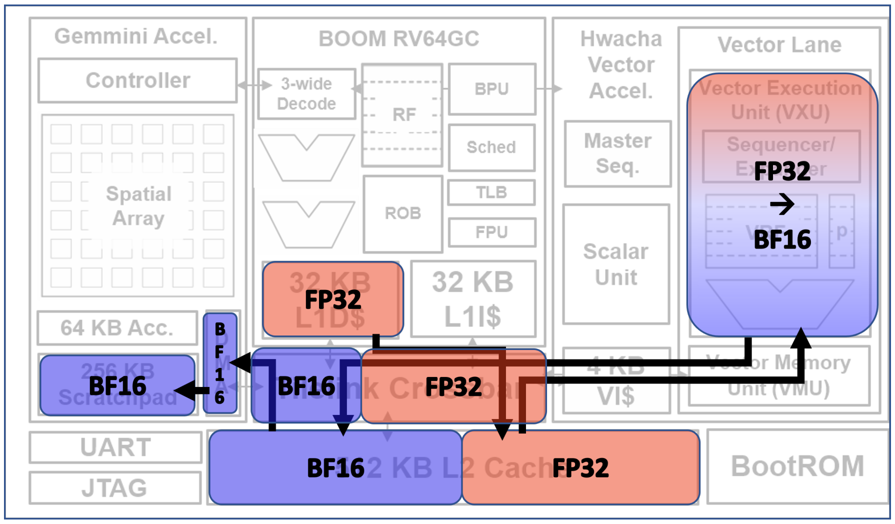
A survey of commercial matrix engines which utilize the bfloat16 format confirms that the choice of conversion approach between single-precision floats and bfloat16 differs across systems. The Google TPUv2 and TPUv3 support hardware conversion within the MXU (matrix multiplication systolic array) [201, 270]. As such, it appears that values can be stored as single-precision floats within the on-chip memory and be converted "on-the-fly" when entering the MXU systolic array. In contrast, the "on-the-fly" hardware conversion supported by Gemmini converts the values when performing DMA transactions between shared memory and private memory rather than between the private memory and the systolic array. This effectively doubles the capacity of the private memory compared to the hardware-supported conversion in the TPU. We note that the TPU also supports software conversion and direct storage of bfloat16 values in the private memory, providing it with the same effective capacity improvement, at the cost of performing the conversion in software on the host processor (potentially during offline pre-processing) [270]. A different matrix engine, the Intel AMX extension, does not support bfloat16 conversion within the matrix extension [140]. Nevertheless, the AVX vector extensions do support such conversion, which leads to the conclusion that the intended conversion usage model is to use the AVX vector unit for such conversions, under the assumption the communication through the L1 cache is likely of low overhead.

Software datatype conversion can be performed efficiently by integrating it with the panel-packing stages of the BLIS algorithm. The BLIS algorithm packs operand matrix elements into panels organized in contiguous memory in order to improve memory bandwidth utilization, and TLB and cache locality. This packing is performed only on micro-kernel operands, while the micro-kernel result matrix is assumed to be laid-out in memory in its original format. Hence, the packing functions are a prime opportunity for performing the conversion from single-precision floats to bfloat16. Having the panel be packed with 16-bit elements as opposed to 32-bit elements improves cache performance and memory bandwidth utilization between the CPU memory system and Gemmini DMA transactions, as intended with the software conversion approach. Nevertheless, this conversion in software doubles the number of operations the CPU performs during the packing routine - instead of just simple data movement, an additional conversion needs to be performed for every element. While "on-the-fly" conversion in the DMA can be performed in conjunction with panel-packing, this approach reduces the arithmetic intensity and matrix sizes the accelerator would operate on, while not helping improve the effective bandwidth of the DMA.

Figure 7.7, which illustrates the performance of square matrix multiplication using kernels and micro-kernels as well as the two approaches to conversion, exhibits an interesting phenomenon. It is clear that the native Gemmini SDK and hardware scheduler (using "Sandbox loops"), which implicitly uses on-the-fly hardware conversion in the DMA, outperform the micro-kernel-based BLIS algorithm loops. This confirms that for high-arithmetic-intensity matrix multiplications, the reduction in effective memory-bandwidth due to conversion in



(a) Conversion of single-precision floating-point to bfloat16 using the "on-the-fly" method in the Gemmini DMA. Communication bandwidth between the shared L2 cache and the Gemmini accelerator is 32-bits-per-element.



(b) Conversion of single-precision floating-point to bfloat16 using the software panel packing in the Hwacha vector unit. Communication bandwidth between the shared L2 cache and the Gemmini accelerator is 16-bits-per-element, while communication bandwidth between the shared L2 cache and the Hwacha vector unit is 32-bits-per-element for loads (non-unit stride) and 16-bits-per-element for stores (unit stride)

Figure 7.8: Datatype conversion approaches from 32-bit single-precision floating-point to 16-bit bfloat16 on the target SoC.

the DMA is not a limiting factor since it is hidden by data re-use within the accelerator with a large enough private scratchpad memory. However, when using the BLIS algorithm loops, which split the matrices into panels with lower arithmetic intensity that fit in the different levels of the memory hierarchy, we observe that software conversion using the vector units outperforms on-the-fly conversion in hardware using the Gemmini DMA. This is since in this case, the matrix multiplication performed by Gemmini has lower arithmetic intensity due to the shape of the panel generated by the BLIS loops, which means that there is insufficient data re-use within Gemmini, making the problem memory-bound and therefore sensitive to the reduction in effective memory bandwidth.

Nevertheless, when using the sandbox and the native Gemmini SDK, "on-the-fly" conversion is superior to vector-unit-based software conversion in all scenarios, including both high-arithmetic-intensity and low-arithmetic-intensity matrix shapes. In high-arithmetic-intensity regimes, cache blocking does not benefit Gemmini, since the Gemmini controller is able the hide the L2 miss penalty latency. In low-arithmetic-intensity regimes, Gemmini utilization is bound by L2-DRAM memory bandwidth, which similarly affects both Gemmini and the vector unit. At the same time, cache blocking limits the degrees of freedom afforded to the Gemmini controller scheduler in high-arithmetic-intensity regimes, since it breaks double buffering and hardware zero padding sequences in the controller.

We note that both the vector unit load/store unit and the accelerator DMA access shared memory through the SoC system bus, which has a width of 128-bits in our experimental setup. The cost of on-the-fly hardware conversion in the Gemmini DMA is relatively inexpensive – since the system bus width is 128-bits, only four 32-bit single-precision conversion units are required in order to utilize and maintain maximal effective bandwidth. Furthermore, conversion for single-precision float to bfloat16 requires a very simple unit, since the only action required is truncation and rounding of the significand/mantissa bits (the exponent and sign bits remain exactly the same, so they can just be "passed through").

Data Layout

The Gemmini accelerator DMA is implemented under the assumption of row-major matrix representations. This design choice was made since the accelerator was designed for deep learning workloads, with the primary programming interface of the accelerator being C and C++ arrays. Matrices in DNN inference can be expected to be processed in contiguous memory, and not require data layout transformations at runtime. In contrast, the BLAS interface is tolerant of a wider variety of data layouts, supporting both column major and row major representations. More precisely, the BLAS interface assumes a column-major layout (due to its reference Fortran implementation), while the CBLAS interface (the C language binding for the BLAS interface) assumes a row-major layout. However, since the BLAS interface also supports a "transpose" indicator argument for each matrix operand, it de-facto defines support for both column-major and row-major layouts on both BLAS and CBLAS, as illustrated in Figure 7.9.

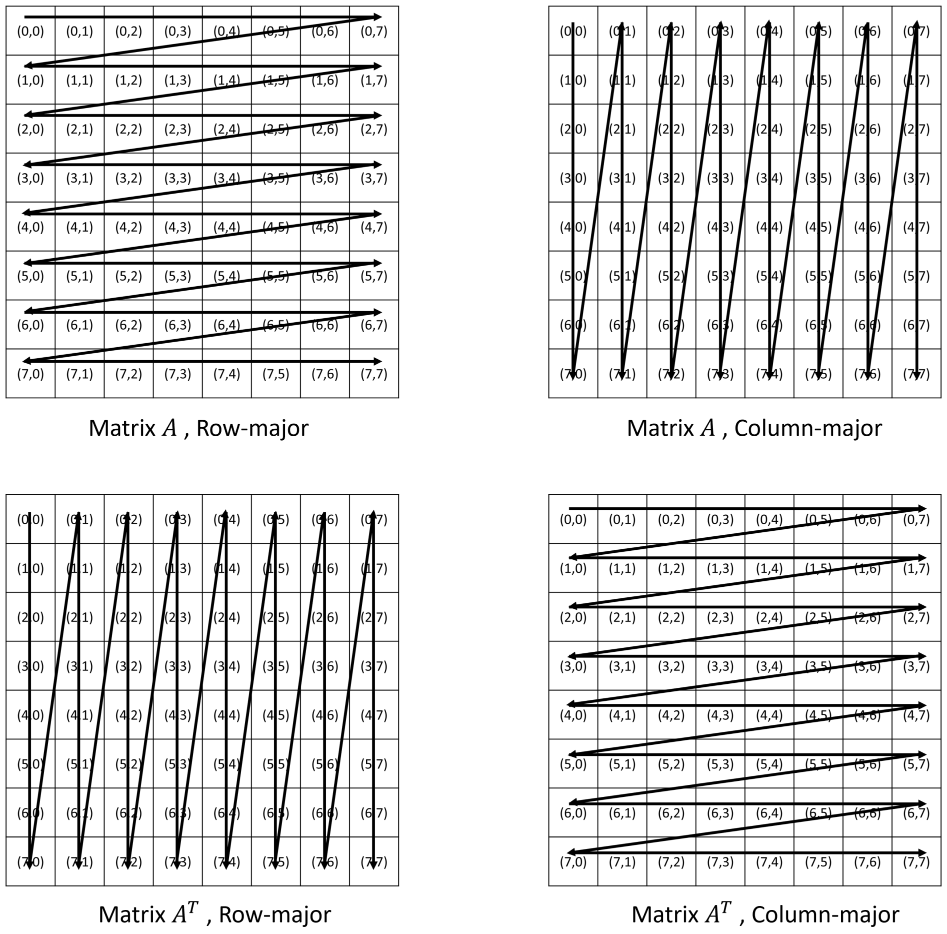


Figure 7.9: Row-major and column-major matrix data layouts in contiguous memory for matrices and their transpose.

In addition, regardless of the data layout of the full operand matrices called using the BLAS function interface, BLIS micro-kernel operand matrix panels are organized using both row-major and column-major representations: the packed operand panel of the A matrix is arranged in column-major representation, while the packed operand panel of the B matrix is arranged in row-major representation. This is done in order to arrange data in an optimal contiguous layout for outer-product computation on scalar processors or packed-SIMD extensions. The results of the micro-kernel (C) can be output in either row-major or column-major representations (depending on row-stride and column-stride parameters to the micro-kernel).

Thankfully, Gemmini has a built-in hardware transposer between the scratchpad memory and the spatial compute array, which can used to transpose one of the operands at the entry to the spatial compute array. The transposer was originally added to Gemmini in order to support the output-stationary (OS) dataflow of the systolic array, under the row-major data layout assumption. By adding a software interface to enable and disable this transposer for a selected operand, we enable our implementation to support almost all data layout combinations enabled by the BLAS interface. Together with the algebraic properties of transposed matrices ($A^T B^T = (BA)^T$, $(A^T)^T = A$), we can enable and disable transpose operands in Gemmini and re-arrange the the order of operands (for example, use B^T as the first operand and A^T as the second operand) in order to obtain our desired data layout.

In the case of BLIS micro-kernels, we allow the Gemmini DMA to move data from both operand panels as if they were in a row-major organization, while performing a computation of $A^T \cdot B$ instead of $A \cdot B$. While at first glance this approach may seem redundant since it transposes the A matrix twice, it does not have a noticeable cost, since there is no additional data-movement when transposing A during packing, and since using A^T within Gemmini is zero-overhead in terms of performance due to overlap of compute operations (but indeed wasteful in terms of energy).

The transposer is implemented as a systolic array where each element of the array includes a simple pipeline register and two multiplexers, as illustrated in Figure 7.10. The multiplexers alternate direction every `DIM` cycles (where `DIM` is the dimension of the array). The alternating direction performs the transposition between rows and columns, while enabling the array to maintain full throughput by alternating the input and output directions. Figure 7.10 illustrates the flow of data within the hardware transposer array. The cost of the transposer is relatively small, as the area it consumes is only 1% compared to the size of the spatial compute array. Exposing the transposer to software is also beneficial for other workloads, as matrix transposition is used in a large number of linear algebra operations, as well as in DNN training.

The BLIS framework additionally provides an even more flexible function interface than the legacy BLAS API. In the BLIS framework, matrix operands (and results) can specify a stride across both dimensions of the matrix (rather than only the leading dimension in the legacy BLAS interface), allowing for potentially non-contiguous data layouts in memory. The Gemmini DMA is not robust enough to tolerate such cases, even with the help of the hardware transposer. In such cases, we must fallback to micro-kernel-based implementations (for both `GEMM` and the rest of the BLAS-3 operations) in which operand panels are packed contiguously, and the micro-kernel result matrix is small enough to have its data layout re-arranged by the host CPU and the vector unit. Tighter integration between Gemmini and a general-purpose vector unit with support for arbitrarily strided and segmented load and store memory operations would help Gemmini support this broader set of use-cases and make the Gemmini DMA redundant. Alternatively, the Gemmini DMA could add support for such strided operations across both the row and column dimensions, as opposed to the current support of only a single strided dimension.

TRSM

The `TRSM` (triangular matrix solve) is the only BLAS-3 kernel that cannot be composed completely out of smaller `GEMM` micro-kernels. A `TRSM` kernel is of the form $C = A^{-1} \cdot B$, where

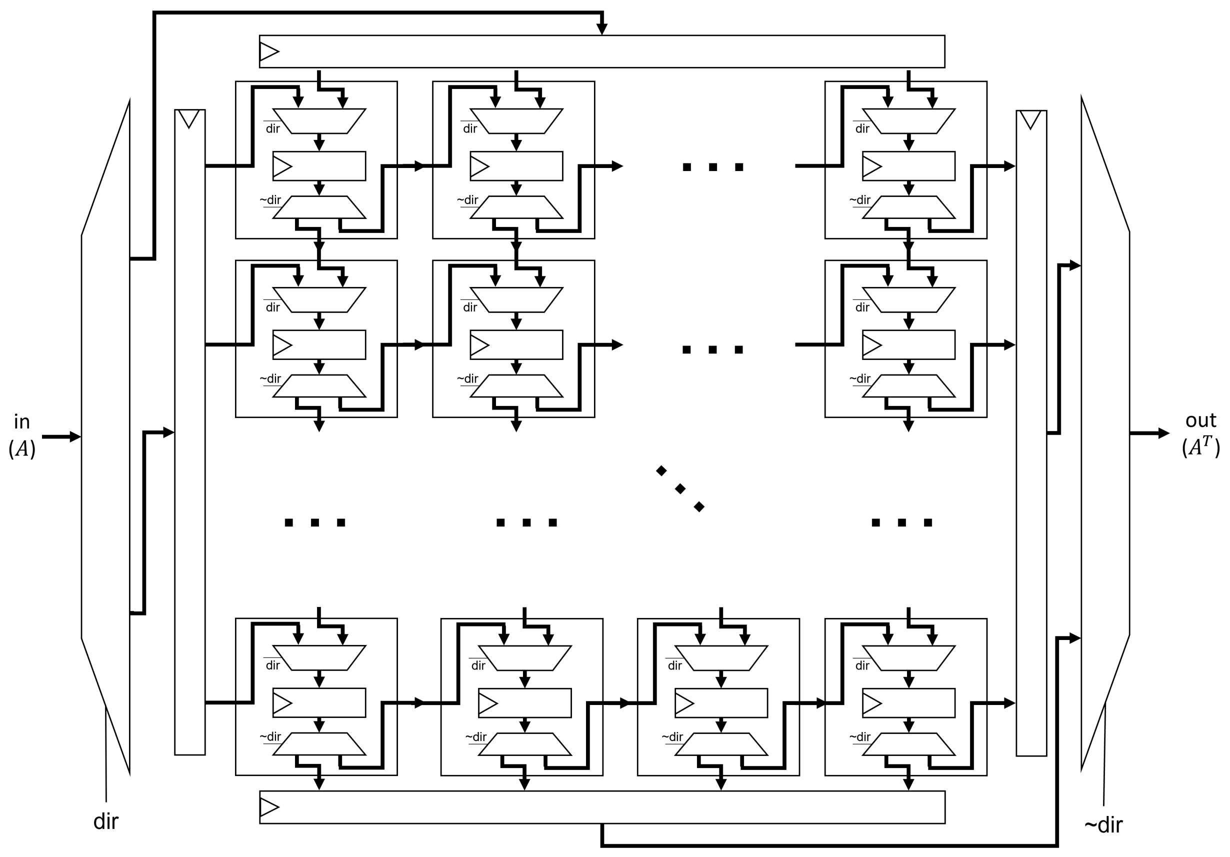


Figure 7.10: Gemmini Hardware Transposer

A is either a lower or upper triangular matrix. This is in-effect backwards substitution, which is an iterative process. The `TRMS` kernel in BLAS-3 can be efficiently implemented using a fused sequence of small `TRSM` micro-kernels and large updates to a trailing matrix using `GEMM` operations. As such, the `GEMMTRSM` micro-kernel in BLIS is a fused micro-kernel in which `TRSM` and `GEMM` subproblems are fused together in a single routine. An efficient implementation of a `GEMMTRSM` micro-kernel is intended to help avoid redundant memory operations that would be incurred if the `TRSM` and `GEMM` micro-kernels were executed separately. In the case of a vectorized implementation, the fused micro-kernel enables keeping data in the vector register file between the `TRSM` and `GEMM` operations, as opposed to communicating it through memory.

Within the context of Gemmini, the `GEMMTRSM` micro-kernel does not allow a simple Gemmini mapping since Gemmini does not have native support for a `TRSM` operation. As a result, we implement the `TRSM` component of the fused `GEMMTRSM` micro-kernel on the CPU or Hwacha vector unit. This brings about several challenges that are a result of the structure and implementation assumptions of the `GEMMTRSM` micro-kernel. The `GEMMTRSM` for a lower

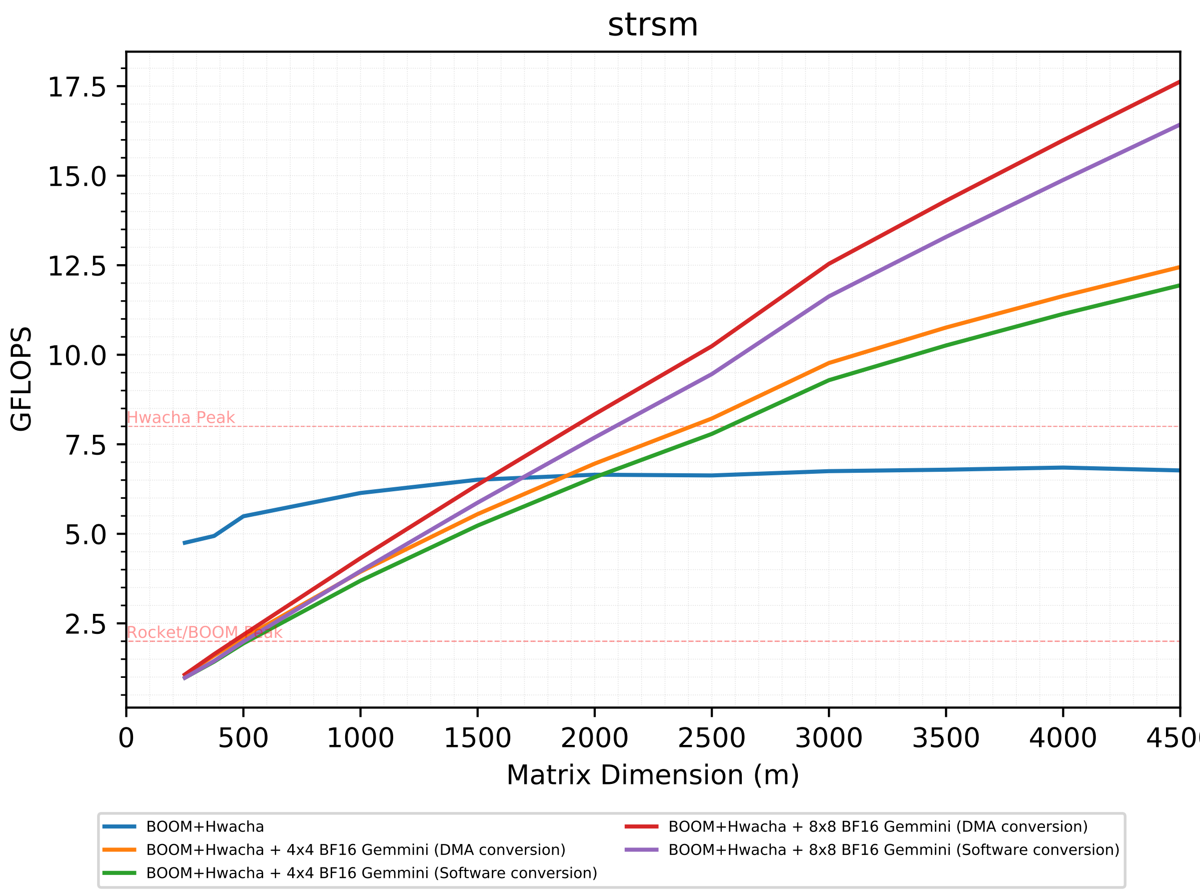


Figure 7.11: `STRSM` kernel performance on square matrices using a 1 GHz SoC with DDR3 backing memory.

triangular matrix performs the following operation:

$$
\begin{aligned}
B_{11} &:= \alpha B_{11} + A_{10}B_{01} \\
B_{11} &:= A_{11}^{-1}B_{11} \\
C_{11} &:= B_{11}
\end{aligned}
$$

Without loss of generality, a similar set of operations applies to `GEMMTRSM` for an upper triangular matrix as well. As the equation shows, the `GEMMTRSM` assumes that the iterative backwards substitution is performed **in-place** by updating the B_{11} sub-matrix, in addition to assigning it to the output C_{11}. These are part of the implementation assumptions for tiled implementations of the full `TRSM` kernel which utilize the `GEMMTRSM` micro-kernel.

The first implication of this implementation assumption is that the mixed-precision aspects of the Gemmini computation is exposed with the `GEMMTRSM` micro-kernel. A conventional `GEMM` computation in Gemmini operates on bfloat16 operands, but results in a single-precision floating-point output (thanks to single-precision accumulation).

Nevertheless, since BLIS micro-kernel operands are packed into panels as part of the BLIS algorithm, and since in this micro-kernel B_{01} and B_{11} are considered operands that are part of the same panel, they will both pass through the packing routine, which will convert them from single-precision to bfloat16. However, B_{11} is also considered to be an accumulation operand, which is performed in single-precision in Gemmini. This brings about again the question of datatype conversion in hardware as part of the DMA transaction vs. in software within the micro-kernel packing function. Figure 7.11 illustrates the performance of the `STRSM` kernel when using different SoC configurations with Hwacha and Gemmini and using the different conversion options. Unlike in the case of the `GEMM` kernel when using BLIS micro-kernels (rather than the "sandbox" native implementation with the hardware scheduler and zero padding), in which the software conversion in the packing function resulted in higher performance due to better utilization of DMA memory bandwidth, in the case of the `TRSM` kernel, using on-the-fly hardware datatype conversion in the DMA provides a performance advantage compared to performing the conversion in software within the packing function despite the reduction in the effective memory bandwidth of the DMA. The reason for this goes back to the in-place update of the B_{11} matrix result of the `GEMM` component within the `GEMMTRSM` micro-kernel. While Gemmini is able to return the result in the form of bfloat16 representation which can fit in-place, the host scalar processor which needs to execute the $A_{11}^{-1}B_{11}$ operation does not support bfloat16, and therefore needs to convert the data into a single-precision floating-point representation. This additional conversion overhead incurs a performance penalty, which outweighs the benefit in maintaining high effective DMA bandwidth. When we use the on-the-fly hardware conversion support, Gemmini can output the results in full 32-bit single-precision representation into the packed panel, and the host processor can act directly on the single-precision float data.

The second implication of in-place updates of the backwards substitution component within the implementation of `GEMMTRSM` regards the overlap of CPU and Gemmini operations. Since the CPU must operate on, **and modify**, the same memory addresses which are written by the result of the `GEMM` performed by Gemmini, we must use a `fence` between the Gemmini `GEMM` operation and the backwards substitution performed by the CPU, in order to make sure that Gemmini has completed its operation and written the results before starting the CPU updates. This means that the CPU is sitting idle throughout most of the operation time of Gemmini, instead of overlapping operations through double-buffering or similar techniques. The `fence` instruction also prevents additional double-buffering within Gemmini itself, since the finite-state-machine which controls the internals of Gemmini data-movement halts and resets upon a fence instruction in order to allow for data to be transferred completely back to the main memory system.

As a result, the performance of the `GEMMTRSM` micro-kernel and the `TRSM` kernel are limited by the CPU, and therefore the kernel does not fully utilize all of the compute resource available to it. We can see this in the performance profile observed in Figure 7.11, which follows Amdahl's law. The number of operations in the `GEMM` component increases with the dimension of the BLIS packed panel (k), while number of operations in the `TRSM` component on the CPU remains fixed with the blocking factor (the size of the micro-kernel). Hence, the speedup Gemmini can contribute to a `GEMMTRSM` micro-kernel (and the resulting `TRSM` kernel) depends on the panel dimension k. In particular, when plugging into Amdahl's law (where s is the speedup Gemmini provides for `GEMM` operations, and C is the constant factor representing the number of operations in the `TRSM` component), we see that the performance of the `GEMMTRSM` micro-kernel is a function of k:

$$f(k) = \frac{1}{1 - p + \frac{p}{s}} = \frac{1}{1 - \frac{k}{k+C} + \frac{\frac{k}{k+C}}{s}} = \frac{s(k+C)}{sC+k} \tag{7.1}$$

Tighter integration between the Gemmini private memory and the vector unit register file could help reduce communication costs between the two computation units to alleviate this bottleneck. This tighter integration can be in the form of dedicated data communication pipelines or through shared register files which would alleviate the need to communicate data through the shared L2 cache.

Minimal Matrix Dimensions

Accelerators cannot attain high utilization without a minimal amount of data which enables the amortization of the overheads of using the accelerator rather than the host processor. Furthermore, accelerators might actually cause a performance degradation in comparison to simply using the host processor if the overheads of calling the accelerator are higher than the speedup the accelerator provides on the computation. These overheads can materialize both in hardware, as can be the case for multiplication of very small matrices using throughput-oriented accelerators such as Hwacha and Gemmini, as well as in software, in which panel-packing and conversion or other special codepaths can represent a non-negligible overhead for a small matrix multiplication operation.

Figure 7.12 illustrates this tradeoff point with a BOOM host processor, Hwacha, 4×4 Gemmini, and 8×8 Gemmini. Notably, when using Gemmini's native SDK and special hardware features such as hardware zero padding and on-the-fly DMA datatype conversion (noted in the figure as "Sandbox loops"), Gemmini outperforms the alternative options even for small matrices with no performance degradation. However, when using the BLIS micro-kernels and software datatype conversion, we observe a crossover point with BOOM at a matrix dimension of 25-30 elements. We observe a similar crossover point for Hwacha at a smaller matrix dimension of 16-20 elements. We note that the jagged pattern of Gemmini performance for small matrices is due to the alignment of matrices to 4 bytes.

We further characterize this behavior by analyzing additional BLAS-3 kernels for which Gemmini does not have native support. Figure 7.13 illustrates the performance of the `TRSM` kernel for small matrices across BOOM, Hwacha, 4×4 Gemmini and 8×8 Gemmini. Notably, in this case, the crossover point between Gemmini outperforming simple CPU execution occurs at a much larger matrix size, with a matrix dimension over 100 elements. We also

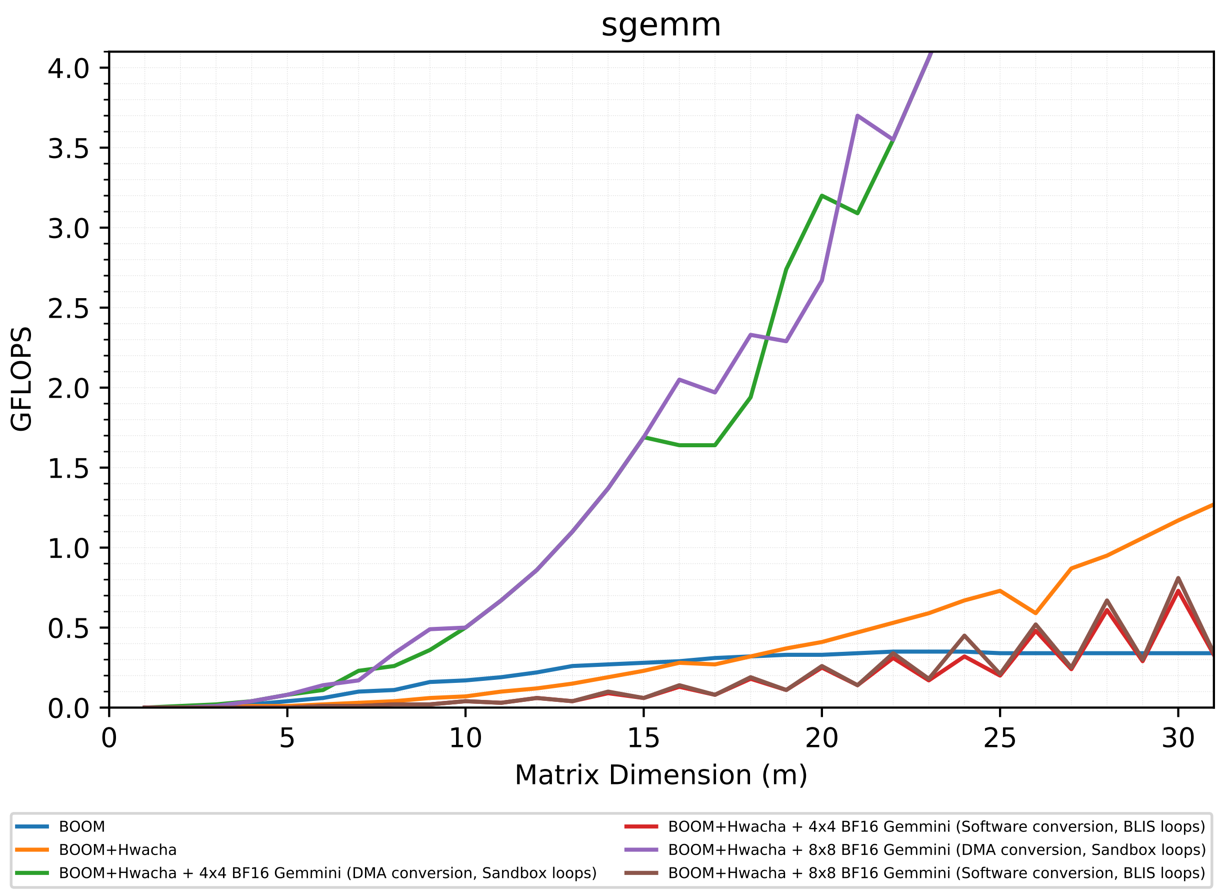


Figure 7.12: `SGEMM` kernel performance on small square matrices using a 1 GHz SoC with DDR3 backing memory.

observe a crossover point with respect to execution on Hwacha, this time for a square matrix dimension of over 20 elements. Since execution on Hwacha is finer grained than execution on Gemmini, and also requires less data movement for the `TRSM` kernel thanks to data re-use within the vector register file, it is not surprising that the crossover point is at a smaller matrix dimension than with Gemmini.

The BLIS framework provide a configuration option which allows developers to set a special handling of matrices with dimensions under a certain threshold. When using Gemmini's native SDK and hardware conversion, there is no need to set such a threshold for GEMMs since Figure 7.12 illustrates the Gemmini's large number of spatial arithmetic units and native support for zero padding result in very low overheads for direct execution of small matrices. However, for the remaining BLAS-3 kernels, we indeed use this BLIS configuration option and execute smaller matrices directly on the BOOM host processor due to the overheads of panel-packing. While it is possible to implement a finer-grained option

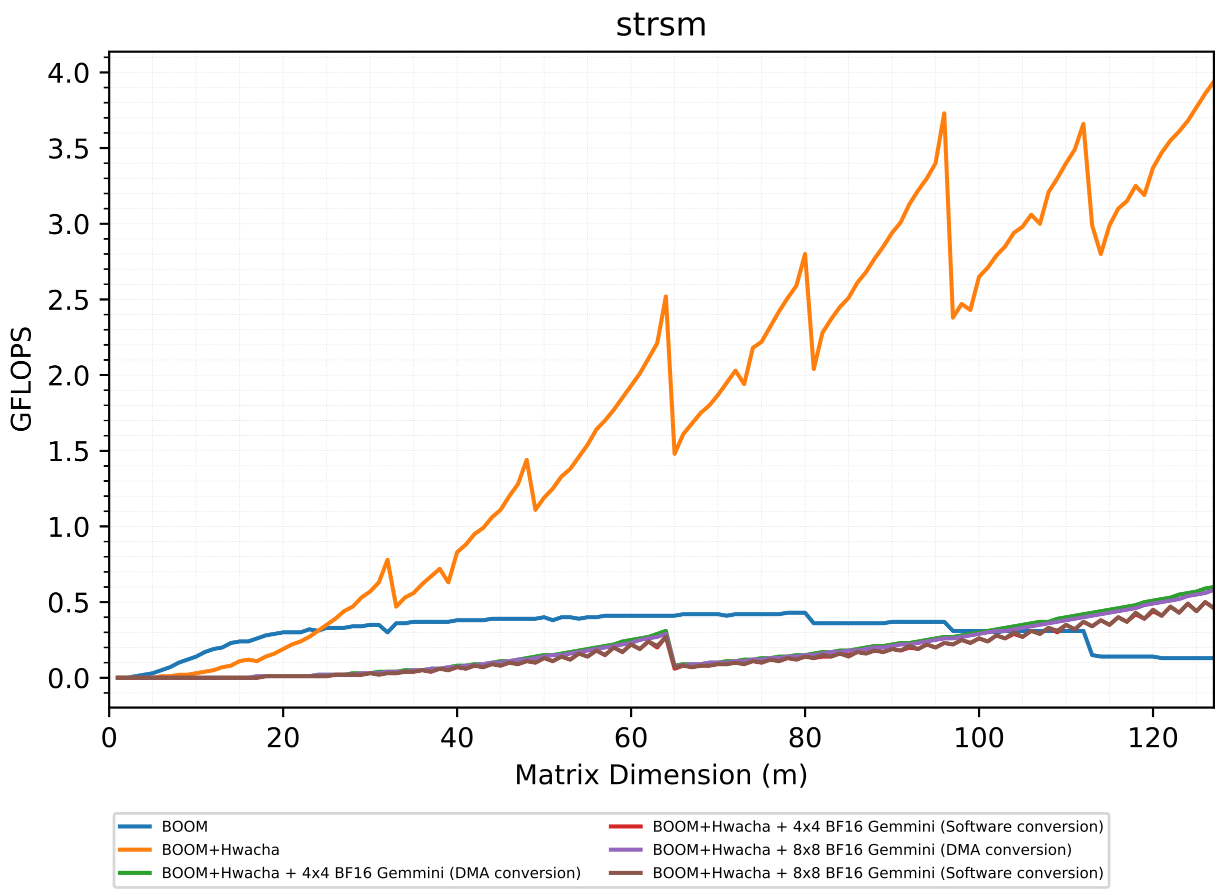


Figure 7.13: `STRSM` kernel performance on small square matrices using a 1 GHz SoC with DDR3 backing memory.

which includes using Hwacha for a subset of matrix sizes between the two crossover points, which would provide the highest performance across all cases, we choose to maintain a single threshold for simplicity of use.

The question of minimal matrix dimensions has importance with respect to micro-kernel implementations since it has a direct impact on the size of the micro-kernels within BLIS. We observe in Figure 7.13 a slight drop in performance for the Gemmini-based configurations at a matrix dimension of 65 elements. This is due to the fact that the configured BLIS panel dimension for this experiment was 64, and therefore matrix dimensions with 65, 66, 67 elements experience high overheads for handling their boundaries. We now further elaborate on how register-blocking techniques and the micro-architecture of the accelerator relate to the size of the micro-kernel and its impact on performance across different matrix sizes.

Accumulator-based Register Blocking

High performance matrix multiplication and BLAS-3 algorithms, including the BLIS algorithm, rely on multiple levels of loop tiling/blocking based on the target processor's memory hierarchy. In this hierarchy, the loops are tiled based on cache sizes down to the inner most loops which are typically tiled based on the processor's register file [102].

Similarly, BLIS micro-kernels are intended to be designed based on register-blocked tile sizes, m_r and n_r. In a standard CPU implementation, the BLIS m_r parameter is typically set based on the number of registers, and the n_r parameter is set based on the width of SIMD registers. This type of register blocking is designed under the assumption of conventional single-element registers or spatial packed-SIMD registers (such as Intel AVX/AVX2/AVX512 or ARM Neon), with typical possible widths of 1-16 elements and up to 32 registers. In fact, the largest possible micro-kernel tile size allowed in the baseline BLIS implementation was limited to 31 elements, based on this type of assumption for register-blocking.

Since Gemmini does not use the explicit concept of a register abstraction, we observe two options for register-blocking equivalences in Gemmini: (1) In output-stationary (OS) dataflow mode, the inner-most loop stores the accumulated computation output in accumulation registers within the PEs of the spatial compute array. In such a case, the register-block size is directly tied to the size of the spatial array, since each PE effectively has only one accumulation register (there are in-fact two physical accumulation registers to support compute overlap, but the programmer sees only a single register). This leads to a small block size which is beneficial for fine-grained micro-kernels. However, this blocking scheme does not fully utilize the Gemmini private scratchpad since there is very little data re-use within the private scratchpad. (2) In weight-stationary (WS) mode, the register-block size is directly tied to the size of the Gemmini SRAM accumulators. A typical size for the SRAM accumulators ranges from tens to hundreds of kilobytes, with the default size in the default Gemmini configuration set as 64 KiB. Assuming double buffering within the SRAM accumulators (i.e., an effective tile of 32 KiB), this leads to potential block sizes of up to 88×88 single-precision floating-point elements, which are significantly larger than typical register-block sizes in BLIS.

A large register-block size for micro-kernels has a negative effect on the efficiency of the BLAS-3 library for matrices which are not close to multiples of the register-block size or on small matrices or, since the micro-kernel is the finest-grained matrix multiplication enabled by the library. As a result, any matrix smaller than the register-block dimension (for example, 88×88), or any matrix which is not aligned with the block size therefore requiring handling of edges and boundary cases, will perform a non-negligible number of non-useful multiplications by zero within the micro-kernel. This is one of the reasons that CPU micro-kernel implementations often have multiple implementations of finer-grained tile sizes to enable fine-grained handling of non-aligned matrix edges and small matrices. While Gemmini has hardware-support for zero-padding, this support cannot be utilized within a micro-kernel since a micro-kernel is assumed to be of fixed dimensions based on the register-blocked size. We indeed utilize the hardware support for zero-padding in the native `GEMM` implementation with on-the-fly hardware datatype conversion in the DMA ("Sandbox loops"), however, we still use panel-packed BLIS implementations, which cannot use hardware zero-padding, for the remaining BLAS-3 kernels. We choose to use a larger micro-kernel which utilizes a large part of the accelerator SRAM accumulator resources under the assumption that the the primary use-case of the BLAS library would be for computations on relatively large matrices, in which the benefits of high accelerator utilization will outweigh the overhead of lower utilization on non-aligned matrices.

7.6 BLAS-3 Performance Evaluation

We present the performance of our custom BLIS implementation using the reference SoC configuration illustrated in Figure 7.3 and the FireSim FPGA-accelerated simulation framework within Chipyard. The SoCs were configured with a single clock domain running at 1 GHz, resulting in BOOM theoretical peak performance of 2 FLOPS (using a single floating-point unit with one double- or single-precision fused multiply-add per cycle), a Hwacha theoretical peak performance of 8 FLOPS (with four fused multiply-adds per cycle), 4×4 Gemmini theoretical peak performance of 32 FLOPS (with 16 PEs each performing a single multiply-accumulate each cycle), and 8×8 Gemmini theoretical peak performance of 128 FLOPS (with 64 PEs each performing a single mixed-precision multiply-accumulate each cycle). Performance evaluation was performed using the BLIS test suite.

Figures 7.7 and 7.11 demonstrate the performance of the `GEMM` and `TRSM` kernels on square matrices. We conclude that using the native Gemmini SDK in a BLIS "sandbox", we are able to reach over 90% utilization on `GEMM` using both 4×4 and 8×8 Gemmini configurations for square matrices with dimensions greater than 1000 elements. We also conclude that using a Gemmini-based `TRSM` is beneficial for square matrices with a dimension greater than 1500. While Gemmini exhibits low utilization (under 50%) for all the matrix dimensions that were evaluated, it is still beneficial in terms of absolute performance compared to using the Hwacha vector unit, exhibiting more than double the performance of Hwacha for matrices larger than 4000×4000.

We characterize the performance of additional BLAS-3 kernels such as `TRMM` (triangular matrix multiplication) and SYRK (symmetric rank-k update) on square matrices in Figures 7.14 and 7.15 respectively. We observe that they can achieve 60%-70% utilization on 8×8 Gemmini and 80%-90% utilization on 4×4 Gemmini. We note that both `TRMM` and `SYRK` benefit from software datatype conversion within the BLIS packing routines rather than on-the-fly DMA conversion due to their micro-kernel-based implementations. We also note that the 8×8 Gemmini spatial array exhibits lower utilization compared to the 4×4 variant, with an approximate drop of 20%. This is partially due to the challenge of sustaining latency-hiding when using small fixed-size micro-kernels as the number of arithmetic units increases.

In term of numerical accuracy, we observe that our kernels implemented using mixed-precision Gemmini with bfloat16 operands and single-precision accumulation generally ex-

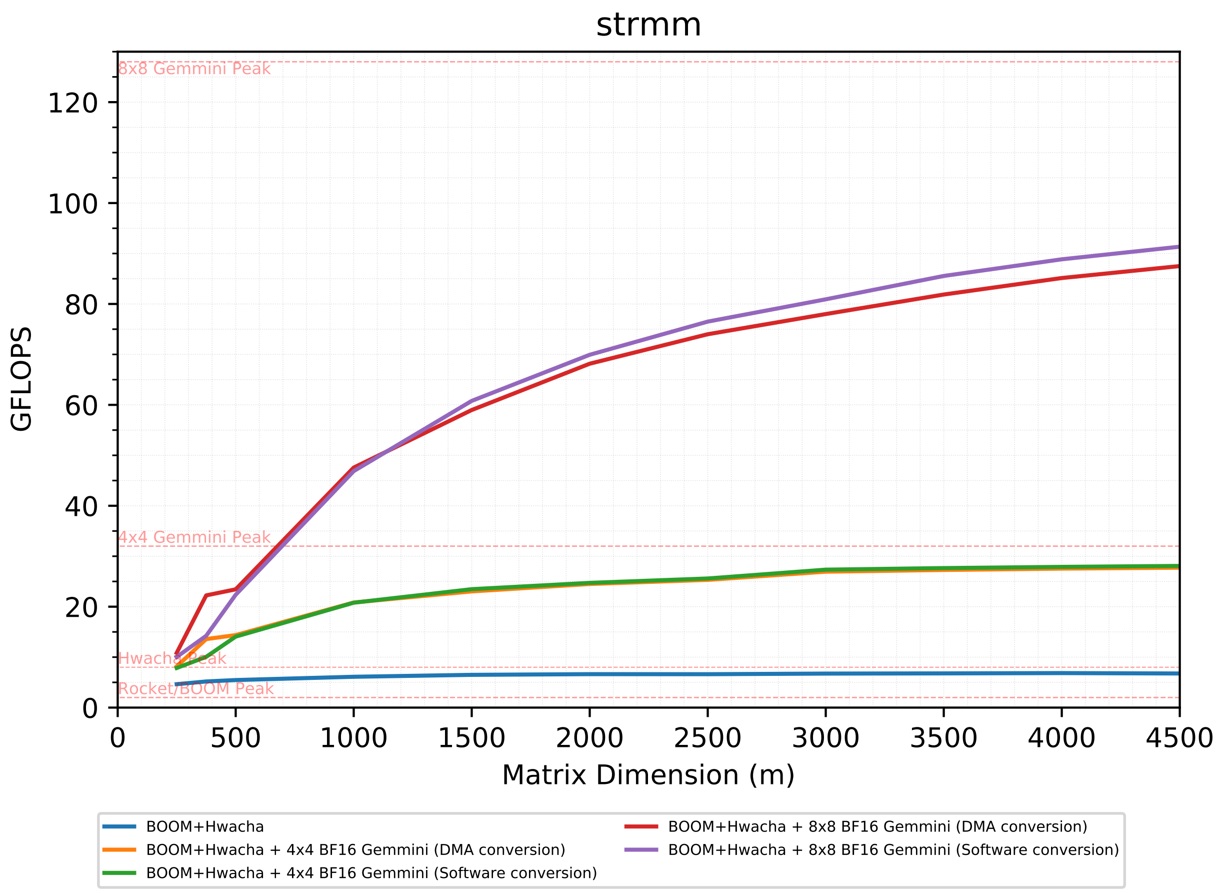


Figure 7.14: `STRMM` kernel performance on square matrices using a 1 GHz SoC with DDR3 backing memory.

hibit a residual norm $10^{-6} - 10^{-7}$, compared to a residual norm of 10^{-8} observed when using full single-precision arithmetic on BOOM or Hwacha. This provides data scientists with an estimate of the potential loss of accuracy when using Gemmini mixed-low-precision computation, and demonstrates that in-practice it is within 1-2 orders of magnitude compared to traditional single-precision computation on the evaluated matrices. Due to the use of single-precision accumulation, we did not observe any cases of result overflow.

7.7 Application-Level Performance

As noted earlier in this chapter and in Figure 7.2, data scientists primarily use libraries and packages embedded in high level languages such as Python, R, Julia, and Matlab. These packages then bind their functions to high performance numerical computing libraries which implement the BLAS and LAPACK interfaces. Therefore, we choose to perform application-

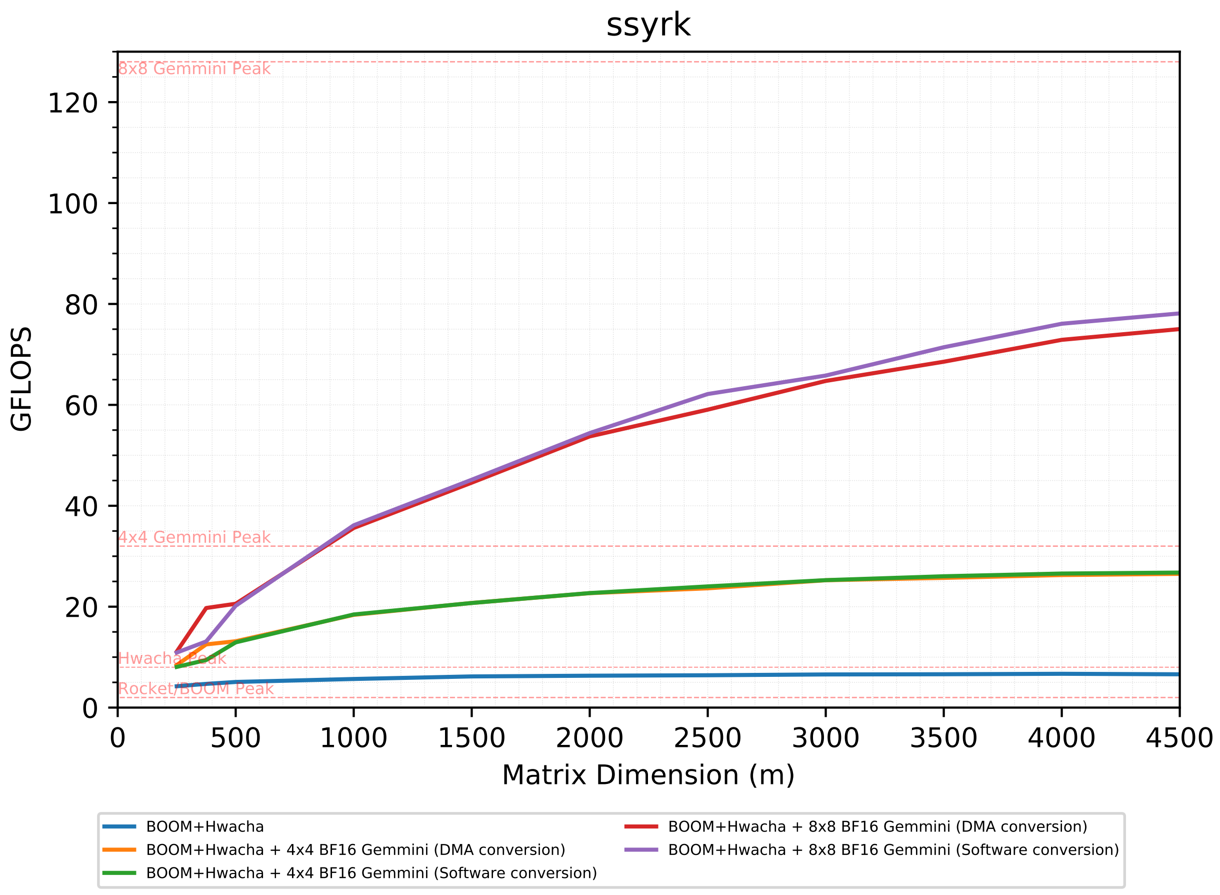


Figure 7.15: `SSYRK` kernel performance on square matrices using a 1 GHz SoC with DDR3 backing memory.

level performance evaluation of our SoC configurations with Gemmini and Hwacha accelerators together with our custom BLAS implementation from the perspective of the data scientists, using the Python SciPy and Scikit-Learn libraries [267, 209]. SciPy and Scikit-Learn are two of the most popular scientific computing, data analysis, and machine learning libraries within the Python ecosystem. We use the custom BLAS implementation described in the previous sections, together with the open-source LAPACK library (version 3.9) as dynamically linked system libraries within a Linux system based on a Fedora-32 Linux distribution. The Fedora-32 distribution provides a complete Linux package management system, together with Python and its associated ecosystem. We run this Fedora distribution on top of our SoC configuration implementation using the FireSim FPGA-accelerated evaluation platform.

LAPACK Challenges

Since LAPACK is an open-source project which is not optimized for a particular target hardware architecture, it uses algorithms and blocking parameters which have been found to provide good, yet potentially non-optimal, performance on a diversity of target platforms. LAPACK assumes that a target-optimized BLAS library implementation is found in the system, and that this optimized BLAS library is linked to LAPACK either statically or dynamically.

We observe that the open-source LAPACK project introduces several challenges to performance tuning of our high-performance software stack based on Gemmini and Hwacha. We elaborate on four of these challenges:

- Recursive algorithms
- Blocking factors
- BLAS decoupling
- Implementation diversity (and mapping to higher levels of the software stack)

Recursive Algorithms

Recursive matrix algorithms have been proposed as a “cache-oblivious” solution to the problem of taking advantage of a processor’s memory hierarchy without having the size of the cache act as an explicit parameter. Thus, cache-oblivious algorithms should be able to achieve high performance on a diversity of machines without explicit tuning or modification. This is in contrast to blocked/tiled algorithms which break the problem into blocks based on explicit block-size parameters derived from the sizes of memories in the memory hierarchy. Since the open-source LAPACK project does not target an explicit micro-architecture, cache-oblivious recursive algorithms are a reasonable approach for it to achieve high performance on a broad spectrum of machines. Thus, multiple core LAPACK kernels are implemented using recursive or divide-and-conquer algorithms. For example, the `xGETRF` LAPACK function for LU decomposition implements a recursive algorithm by Sivan Toledo [258]. Similarly, the Cholesky decomposition function (`xPOTRF`) is also implemented as a recursive algorithm.

While recursive algorithms may indeed be beneficial for a simple CPU with a simple memory hierarchy, they actually present some challenging use-cases for modern architectures with wide SIMD or vector registers and dedicated accelerators. Most notably, recursive algorithms continuously decrease the size of the matrices being operated on, often down to a base case of a 1×1 matrix. This approach is problematic for hardware units which extract performance from internal data re-use within the smallest level of the memory hierarchy. For example, machines with wide SIMD or vector registers assume data re-use within such registers. Similarly, machines with wide datapaths and arithmetic units assume that data can fill that datapath in order to obtain full utilization.

Both Gemmini and Hwacha are throughput processors which rely on latency-hiding and start-up cost amortization. Small operations are very costly using such architectures, and would likely cause a performance degradation compared to execution on a simple scalar processor. In the case of Gemmini this disadvantage is amplified even further, since in order to obtain high utilization from the spatial compute array, Gemmini must reach a minimal level of data re-use within its private scratchpad and accumulators, which are not exposed to the recursive algorithm. Hence, when a recursive algorithm calls a large number of small matrix operations of sizes 1×1, 2×2, and 4×4 towards the base of the recursion, it is effectively repeatedly thrashing the accelerator.

One potential solution for such a problem would be to choose a larger dimension for the base case of the recursion – rather than continuing the recursion all the way down to the base case of size 1, the recursion could stop at a larger matrix size. However, while this will reduce the level of thrashing of the accelerator, it is still not optimal since the accelerators are at their peak utilization when the startup cost is incurred a minimal number of times. Furthermore, choosing a recursion base greater than 1 means that these algorithms would not be oblivious anymore to tuning parameters, taking away one of their main advantages.

Blocking Factors

The open-source LAPACK project provides the ability to tune the implementation through tuning parameters collected within a function called `ilaenv`. These tuning parameters include the minimal and optimal block size for the target platform, the crossover point between blocked and unblocked implementations, and the maximum problem size at the base of some divide-and-conquer computation trees.

At first glance, we might assume that given the latency-hiding properties of both Hwacha and Gemmini, we would like to increase the blocking parameters to make better use of their increased computational resources. However, this is not always the case – while some blocking algorithms only re-arrange computation, other in-fact require an additional number of operations in order to re-arrange the computation, hence generating a tuning parameter tradeoff between floating-point operations and communication. This means that while certain kernels will benefit from increased block sizes, other will see a performance degradation due to an increased number of operations. We elaborate on this topic in Chapter 8.

BLAS Decoupling

LAPACK was designed under an assumption of decoupling between the implementation of the BLAS functions and the implementation of LAPACK functions. However, this decoupling may obscure some of the underlying assumptions hidden underneath the BLAS layer of abstraction. For example, in certain cases, a BLAS implementation may set a minimal matrix size threshold for matrices to be packed in memory or executed on an accelerator, while allowing smaller matrix sizes to be executed directly on the host scalar processor with lower overheads. Knowledge of these thresholds may assist the LAPACK implementation in setting its tuning parameters, or even making different algorithmic choices.

Conversely, LAPACK function implementations sometimes reduce BLAS-3 operations to BLAS-2 or BLAS-1 operations but continue using the BLAS-3 API. For example, scalar multiplication and matrix-vector multiplications can be represented as matrix-matrix multiplications with one or both dimensions being of size 1. While mathematically equivalent, their underlying implementations may be very different and incur unnecessary overheads.

While open-source BLAS libraries such as OpenBLAS and ATLAS provide integrated and better optimized implementations of several LAPACK functions, it is unrealistic for these libraries to provide an implementation of the entire LAPACK API optimized for a particular BLAS implementation, due to the sheer number of routines in LAPACK and the algorithmic complexity of their operations. Commercial numerical libraries such as Intel MKL and Apple Accelerate provide a larger number of integrated LAPACK routines but may still make use of some general-purpose routines originating from the open-source LAPACK implementation which do not require tight integration. A common choice made by several libraries is to only jointly optimize the implementations of several core matrix decomposition routines such as LU, QR, and Cholesky deocompositions together with a specific BLAS implementation, since a large number of LAPACK routines rely on these core decomposition functions, resulting in a high marginal gain from optimizing these few functions.

Implementation Diversity

LAPACK has a high diversity of implementations for multiple classes of problems. For example, the default LU deocomposition implementation uses a recursive algorithm, but there is also an alternative implementation based on traditional blocking found within the codebase. Similarly, there are two SVD implementations, and four linear least squares problem implementations. Implementations may trade off accuracy robustness for performance efficiency, or simply use different algorithmic techniques which may perform differently on different hardware architectures (which is especially important when considering reduced-precision hardware). This diversity of implementations puts an onus both on the end user (the data scientist) as well as other layers of the software stack, which often set a default implementation to use. For the majority of end users which are not numerical analysis experts and are not familiar with the nuances of LAPACK routines, the default driver binding within a high-level framework such as Matlab, Julia, R or SciPy will be their only interface with the LAPACK implementation of this routine. As such, implementations which use accelerators should attempt to optimize for the default implementations used in high-level frameworks.

Matrix Decompositions

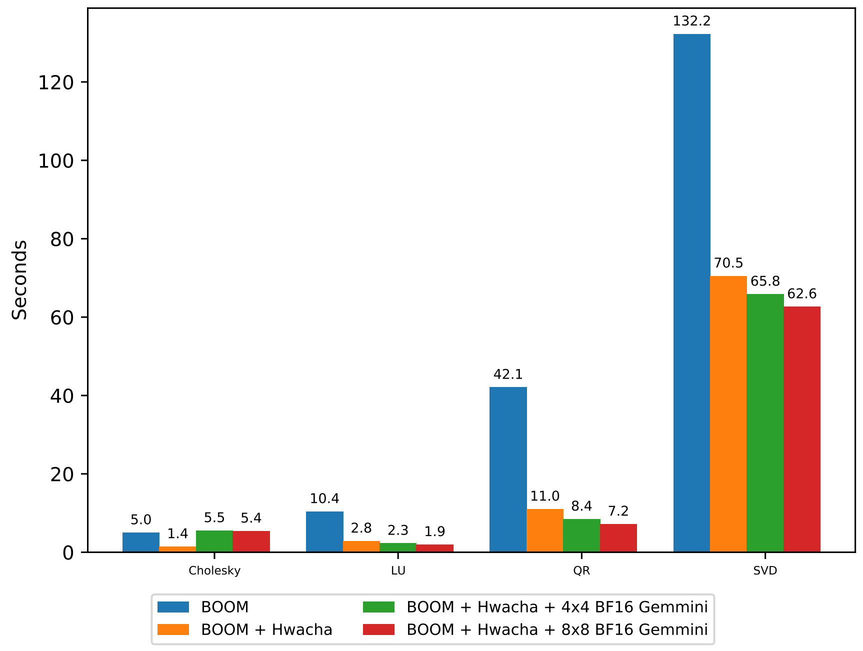
Matrix decompositions form the core of many applications in numerical analysis. Linear least squares problems can be solved by using the QR decomposition, principal component analysis and low rank approximation are based on the singular value decomposition, and linear systems of equations and matrix inversions are typically solved using the LU decomposition. We focus our analysis on four primary matrix decompositions:

- LU decomposition ($A = LU$) - factors a matrix A into a product of a lower triangular matrix L and an upper triangular matrix U.
- QR decomposition ($A = QR$) - factors a matrix A into a product of an orthogonal matrix Q and an upper traingular matrix R.
- Singluar value decomposition (SVD, $A = U\Sigma V^T$) - factors a matrix A into a product of a unitary matrix U, a rectangular diagonal matrix Σ, and a unitary matrix V^* (or V^T for real matrices).
- Cholesky decomposition ($A = L\Sigma L^T$) - factors a Hermitian positive definite matrix A into a product of a lower triangular matrix L and its conjugate transpose L^* (or L^T for real matrices).

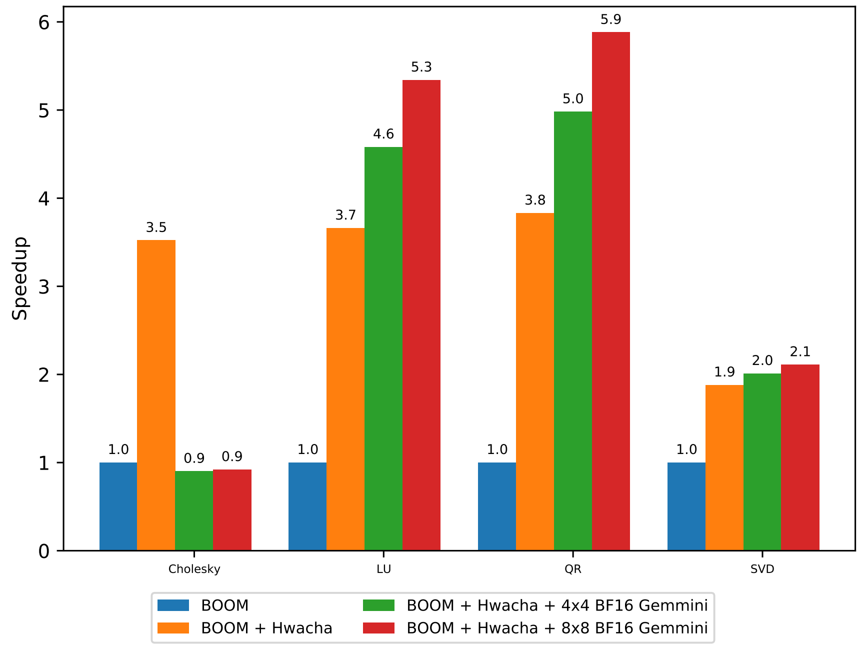
As noted at the beginning of Section 7.7, we evaluate matrix decompositions by using the SciPy Python package in order to maintain the perspective of the data scientists end-users. SciPy is a popular scientific computing package within Python, which uses many LAPACK functions. In many cases, the SciPy library simply acts as a Python binding to the LAPACK shared library found in the system. Specifically, the SciPy LU decomposition calls the `xGETRF` LAPACK function, the SciPy QR decomposition function calls the `xGETQRF` LAPACK function, the SciPy Cholesky decomposition function calls the `xPOTRF` LAPACK function, and the SciPy SVD function calls the `xGESDD` LAPACK function.

We note that in some cases, such as SVD, LAPACK provides additional function variants which can be more accurate or more efficient under certain assumptions. For example. the `xGESVD` function is slower than `xGESDD` but can provide more accurate results than `xGESDD` for ill-conditioned matrices. Runtime can also be impacted by the choice between computing only the singular values vs. also computing the singular vectors. If computing the singular vectors, `xGESDD` will compute both the left and right singular vectors, while `xGESVD` may let the user choose whether to compute only one side. The SciPy (and NumPy) implementations of SVD default to using `xGESDD` due to its faster performance, while other numerical platforms such as Matlab, Octave, and Julia default to `xGESVD`. For advanced users, SciPy provides an optional function argument to explicitly call an alternative LAPACK driver routine.

Figure 7.16 illustrates the performance of several matrix decompositions run in SciPy on a 1600×1600 matrix using SoC configurations consisting of the BOOM core, a Hwacha vector unit, a 4×4 Gemmini accelerator and an 8×8 Gemmini accelerator. As expected, we observe that the Hwacha and Gemmini accelerators provide noticeable speedups on matrix decomposition which rely heavily on BLAS functions, and specifically level 3 BLAS functions. The Hwacha vector unit yields a $3.5\times - 3.8\times$ speedup over BOOM for LU, QR and Cholesky decompositions, while Gemmini provides a $4.6\times - 5\times$ speedup for a 4×4 configuration and $5.3\times - 5.9\times$ speedup for an 8×8 configuration.



(a) Execution time of 1600 × 1600 matrix decompositions from the SciPy package using custom BLAS implementations utilizing SoC configurations of the Hwacha and Gemmini accelerators.



(b) Speedup (compared to BOOM baseline) of 1600 × 1600 matrix decompositions from the SciPy package using custom BLAS implementations utilizing SoC configurations of the Hwacha and Gemmini accelerators.

Figure 7.16: Matrix decomposition performance using custom accelerator BLAS.

Notably, we observe that the SoC configuration with the Gemmini accelerator results in a slowdown of the Cholesky decomposition as opposed to the expected speedup (and the speedup observed when using the Hwacha accelerator). This is an example that combines multiple challenges of mapping LAPACK and BLIS to accelerators on custom SoCs. Specifically, the recursive nature of the Cholesky decomposition in the default open-source LAPACK implementation generates significant overheads to the micro-kernel-based BLIS implementation of the symmetric rank-k update function (`SYRK`) due to the small sizes of the recursive base calls compared to the size of the minimal micro-kernel operands. In contrast to the LU decomposition (`GETRF` which is also implemented as a recursive algorithm), the Cholesky decomposition relies heavily on the `SYRK` function rather than the `GEMM` function, as it takes advantage of the symmetry of the matrix. While `SYRK` is generally considered to be more efficient than `GEMM`, and consumes less operations, it is not implemented natively by our accelerator and not supported directly by the accelerator controller. Therefore, the `SYRK` BLAS function is implemented based on the BLIS loops and the `GEMM` micro-kernel within BLIS. However, since the minimal size of this micro-kernel is on the order of the Gemmini accumulator SRAMs, the recursive algorithm thrashes the micro-kernel by calling it with fixed-size operands with a large number of unused zeros. In contrast, the LU decomposition implementation uses a direct implementation of the `GEMM` kernel and is able to utilize Gemmini's native hardware zero-padding, therefore not generating any micro-kernel overheads. Similarly, the Hwacha-based implementation performs better than the Gemmini-based implementations since Hwacha micro-kernel dimensions are smaller than Gemmini's. In this evaluation, the Hwacha micro-kernel dimensions are 16×32 while Gemmini micro-kernel dimensions are 64×64. While both accelerators require data re-use in order to sustain their arithmetic units, Gemmini's larger number of arithmetic units requires larger micro-kernel dimensions in order to sustain utilization higher than 90%. One possible avenue to improve the performance of our Gemmini-based micro-kernels would be to reduce the micro-kernel dimensions, motivating better support in Gemmini for smaller matrices. We further discuss these topics in Chapter 8. Nevertheless, we conclude that a micro-kernel-based approach to BLAS does not present performant results when used together with throughput accelerators and recursive algorithms resulting in small matrix operands, to the extent that in certain cases it presents a significant slowdown compared to usage of only the CPU.

We also observe that the speedup obtained for SVD is generally lower than the speedup obtained for other decompositions, ranging between $1.9\times - 2.1\times$ for all three evaluated configurations, compared to $3.5\times - 5.9\times$ for Cholesky, LU, and QR decompositions. This is since only 50% of operations in SVD can be expressed as BLAS-3 operations, as bidiagnolization can be expressed only as BLAS-2 operations rather than BLAS-3 [79]. This is in contrast to LU, QR, and Cholesky decompositions in which the majority of operations can be expressed as BLAS-3 operations.

We confirm this conclusion by analyzing the observed performance results with respect to Amdahl's law. When we analyze the results in relation to Amdahl's law, we observe that the accelerators achieve 80%-100% of the maximal theoretical speedup available through acceleration of BLAS-3 operations. Table 7.3 presents an analysis of the speedup limit

Kernel	p*	s_H*	Hwacha Speedup Limit	s_{G4}*	4×4 Gemmini Speedup Limit	s_{G8}*	8×8 Gemmini Speedup Limit
LU Decomp.	0.86	7×	3.8×	25×	5.73×	120x	6.8×
QR Decomp.	0.87	7×	3.93×	25×	6.07×	120x	7.291×
SVD	0.54	7×	1.9×	25×	2.03×	120×	2.13×

Table 7.3: Speedup limits based on Amdahl's Law for 1600x1600 matrix factorizations.

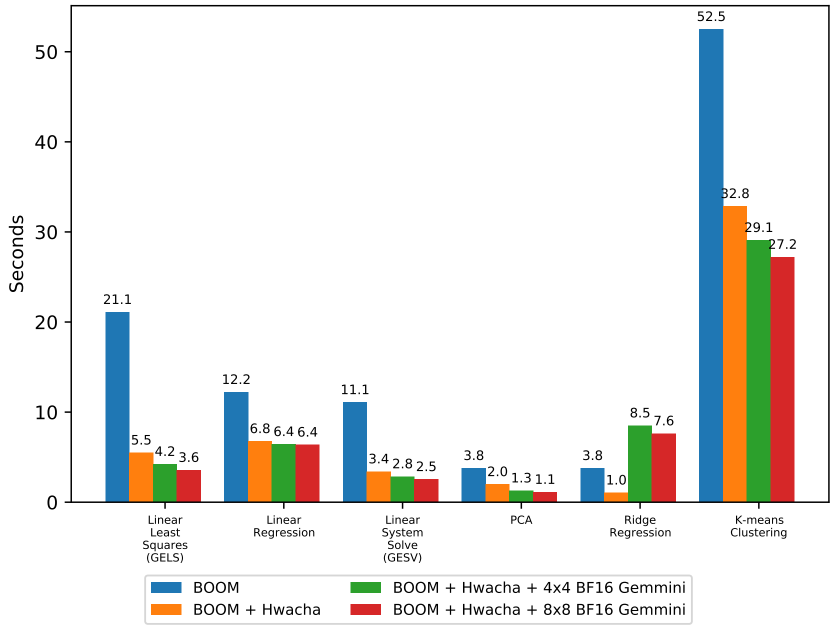
*Both the s and the p parameters are based on empirical measurements. The speedup compared to BOOM is based on empirical measurement of the BLAS GEMM implementations, and is therefore greater than the number of additional arithmetic units since the BOOM baseline BLAS implementation achieves 1 GFLOPS on GEMMs, representing only 50% utilization of its single FPU at 1 GHz.

based on Amdahl's law. The proportion of computation being sped-up (the p parameter) has been obtained through empirical profiling of the kernel on the BOOM processor. The theoretical speedup is computed based on Amdahl's law: $\frac{1}{1-p+\frac{p}{s}}$. Unsurprisingly, we observe the diminishing returns of using the more powerful matrix accelerators as speedup is bounded by the fraction of matrix multiplications within the end-to-end workloads (which in this case, are the matrix decompositions).

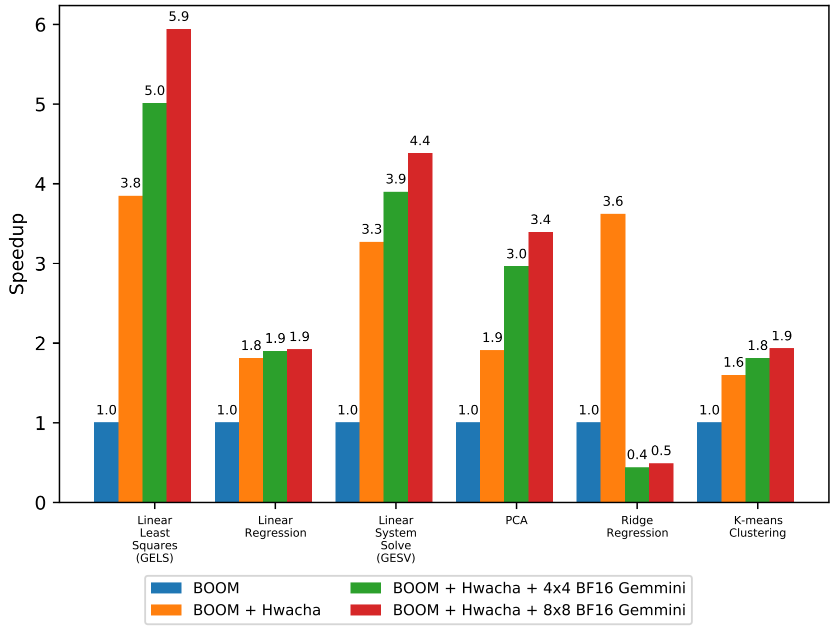
SciPy and Scikit-Learn Applications

We continue evaluating the performance of the accelerated data analysis applications from the perspective of the end-users, the data scientists, using the SciPy Python package, and further add the Scikit-Learn Python package. Scikit-Learn is a popular machine learning and data analysis package within the Python ecosystem [209]. It is built on top of the SciPy and NumPy libraries and provides implementations of fundamental data analysis and machine learning operations such as clustering algorithms, linear models, dimensionality reduction and matrix decompositions, random projections, support vector machines, mixture models, and others.

Figure 7.17 illustrates the performance of several data analysis and modeling workloads running on the UCI Human Activity Recognition dataset [18]. This dataset includes numerical measurements of smartphone embedded inertial sensors from 30 subjects performing human activities (walking, sitting, standing, etc.). The dataset is split into a training partition and a test partition, with the test partition being composed of 561 features (columns) and 2948 samples (rows). Specifically, we evaluate the linear system solve (`scipy.linalg.solve`, based on LAPACK `xGESV`) and linear least squares (based on LAPACK `xGELS`) functions from the SciPy package, and the linear regression, Ridge regression, PCA, and K-means clustering functions (`sklearn.linear_model.LinearRegression`,



(a) Execution time of data analysis functions from the SciPy and Scikit-learn packages, analyzing the UCI human activity recognition dataset, using custom BLAS implementations with the Hwacha and Gemmini accelerators.



(b) Speedup (compared to BOOM baseline) of of data analysis functions from the SciPy and Scikit-learn packages, analyzing the UCI human activity recognition dataset, using custom BLAS implementations with the Hwacha and Gemmini accelerators.

Figure 7.17: Application-level performance using custom accelerator BLAS.

`sklearn.linear_model.Ridge`, `sklearn.decomposition.PCA`, `sklearn.cluster.KMeans`, respectively) from the Scikit-Learn package.

We observe a difference between the speedup of the linear least squares function based on `GELS`, and the linear regression function in Scikit-Learn, despite the fact that a linear regression is an ordinary least squares problem. While the linear least squares function in SciPy using the `xGELS` driver exhibits a $3.8\times - 5.9\times$ speedup with the accelerators, the linear regression function in Scikit-learn is limited to a speedup of only $1.8\times - 1.9\times$. The Scikit-Learn linear regression function internally calls the SciPy linear least squares function (`scipy.linag.lstsq`). The default LAPACK driver for this function is `xGELSD`, which in fact uses the more robust SVD to compute the solution to the linear least squares problem, as opposed to a more efficient QR decomposition which is used in the LAPACK `xGELS` driver function. Specifically, while SciPy provides an option to use alternative LAPACK drivers such as `xGELSY` (uses complete orthogonal factorization) or `xGELSS` (non-divide-and-conquer SVD), it does not expose the option to use the `xGELS` LAPACK driver as part of the `scipy.linag.lstsq` function (the `xGELS` function is exposed independently in SciPy, as part of the LAPACK drivers module in SciPy, which was also used in our evaluation for comparison). This is since `xGELS` can be used only for full-rank matrices, while the `scipy.linag.lstsq` targets a more general case, which may include rank-deficient matrices. Hence, the linear regression function is limited to the speedup enabled by SVD (which was up to $2\times$), while the `xGELS`-based least squares function exhibits speedups similar to the QR decomposition ($3.8\times - 5.9\times$). This example demonstrates the importance of understanding the complete software stack from the perspective of the data scientists, and how domain knowledge can be lost across the software stack which can result in performance implications.

Furthermore, we observe how problems relating to kernel size and recursive algorithms propagate through the software stack. For example, the Ridge regression function in Scikit-Learn uses by default the Cholesky decomposition as a closed-form solution to the regularized least squares problem (since for the Ridge Regression, regularized form of normal equations $(A^TA+\lambda I)x = A^TY$ is used, with $(A^TA+\lambda I)$ being positive definite). While it is possible to use alternative solvers within the Ridge regression function (such as SVD or dedicated sparse solvers), making this selection requires an understanding of the source of the performance anomaly, which traces back to the micro-kernels of the BLIS implementation using custom accelerators.

Interestingly, the PCA in Scikit-learn exhibits significantly greater speedups when using the accelerators, compared to the speedups of the SVD in SciPy. Specifically, the SciPy SVD demonstrates a speedup of $1.9\times - 2.1\times$, which was limited by Amdahl's law due to bidiagnolization, while the Scikit-learn PCA demonstrates a $1.9\times - 3.4\times$ speedup when using the accelerators. At first impression, this should be a surprising result, since a PCA performs the exact same operations as an SVD (together with some additinal data centering). However, for certain scenarios, Scikit-learn uses a *randomized* SVD rather than a full SVD, when the number of desired principal components is small and the data matrix is large enough. This was indeed the case for our example workloads, since we extracted only five principal components from a relatively large data matrix. In a randomized SVD, the data matrix is projected to a smaller space by multiplying it with a smaller random projection matrix (based on probabilistic guarantees of random linear algebra), and then the SVD is performed only on the much smaller projected matrix. Therefore, the speedup we observe is a mix of the limited speedup obtained from the small SVD, together with the greater speedup obtain from the matrix multiplication during the random projection. This is an example of how integration of accelerators within low levels of the general-purpose numerical computing software stack can present speedups through a variety of channels, rather than only through dedicated functions with customized implementations optimized for the accelerator.

Finally, within the context of K-means clustering, we have found datatype precision to be a contributing factor to limited speedup. The Lloyd algorithm for K-means clustering uses matrix multiplications to compute the pairwise distances between the samples and the centroids during each iteration of the algorithm. As such, the higher the number of clusters, the higher the arithmetic intensity would be. However, empirical profiling indicates that within the internal implementation of the algorithm there is use of both single-precision of double-precision matrix multiplications, likely due to datatype inference within the Python implementation. Since the BLAS implementation for our custom SoC configuration was designed to accelerated only single-precision floating-points, the obtained speedup does not apply to the full matrix-multiplication potential within the algorithm.

In conclusion, we observe that our custom BLAS implementation utilizing SoC accelerators is able to relay the accelerator capabilities up the numerical computing software stack, providing application-level speedup to user without a need to change application code. However, we also conclude that many potential speedup gains are lost while propagating up the software stack, due to conservative default choices of function implementations and datatype precisions. Finally, we note that from the perspective of supplemental-use applications, the use of template-based and micro-kernel-based BLAS implementations is beneficial, but exposes the intricacies of integration between layers across the software stack, such as the problematic interaction with recursive algorithms. This motivates more robust support within accelerators for smaller micro-kernels, which will be discussed in the next chapter.

Chapter 8

Hardware/Software Co-Design for Numerical Data Analysis

Numerical data analysis has been an important workload in scientific computing for many years but was not of enough economic importance to justify custom acceleration in modern computing platforms. However, with the emergence of ubiquitous deep learning on all new compute platforms, customizing deep learning accelerators to meet the requirements of numerical data analysis applications could prove beneficial. Therefore, we explore HW/SW co-design of numerical data analysis workloads with deep learning accelerators to identify customization opportunities to better support numerical data analysis applications on this class of accelerators.

8.1 Hardware/Software Co-Design for Supplemental-Use

The number of dedicated accelerators on SoCs has steadily increased in the past decade. Specialized accelerators comprise a major part in SoC architectures, accounting for over 60% of the area in recent SoC designs [236]. A primary reason for the continuously increasing number of accelerators on SoCs is their energy and power efficiency, used as a countermeasure against the limits of dark silicon [84]. At the same time, these accelerators experience extremely low utilization profiles (with respect to absolute time), often characterized by bursts of usage, due to their single-function/few-function design points limiting their usage to a handful of applications. Since accelerators are often the most energy efficient computing unit to perform their particular type of computation on the SoC, we would like to increase the utilization of such accelerators if possible by expanding their domain. Together with a predicted ceiling (or wall) in further accelerator specialization [92], it becomes necessary to explore HW/SW co-design methods for increasing the utility of SoC accelerators through supplemental-use applications. We would therefore like to explore whether there are certain hardware or software features that could be customized in existing SoC accelerators to make them better suited for supplemental use. Specifically, we would like to evaluate potential modifications or customizations to DNN accelerators in SoCs to improve their performance for numerical data analysis workloads.

8.2 Matrix Engines for Numerical Data Analysis vs. DNNs

Numerical data analysis methods and tools typically rely on several core numerical linear algebra problems such as exact or least-squares solutions of linear systems, together with basic matrix operations such as matrix multiplications and matrix factorizations. At first glance, this makes them appear quite amenable for execution using deep learning accelerators. However, while both DNNs and the linear algebra kernels at the basis of numerical data analysis workloads are dominated by matrix-matrix operations, these differ in several key characteristics which impact the efficient use of matrix engines within DNN accelerators for such a mix of workloads.

Matrix Shapes

The first difference between DNNs and the broader category of numerical data analysis workloads is in the diversity of matrix shapes and sizes that need to be processed by the accelerator. In order to obtain high-performance in processors with a memory hierarchy, optimized linear algebra library implementations use blocking techniques or recursive implementations to reduce communication across the memory hierarchy. The most well-known example of a blocking technique is the use of cache blocking through outer-products in matrix multiplication. However, as seen in Chapter 7, blocking techniques and recursive implementations are also used in more complex matrix decompositions such as Cholesky, LU, and QR factorizations. In LAPACK, the goal of blocking techniques is generally to convert a series of BLAS-2 operations (matrix-vector operations) into BLAS-3 operations (matrix-matrix operations) by delaying and grouping certain steps of the algorithm. BLAS-3 operations generally have a higher arithmetic intensity than BLAS-2 operations, which enables higher data re-use and reduced communication within the algorithm. However, the matrix operations that result from these communication-reduction techniques can differ in their shapes and sizes compared to matrices found in typical DNN models. In most blocked algorithms, matrix sizes and shapes can be partially controlled through tuning parameters. DNN models and matrix factorizations both have tuning parameters and hyperparameters which can impact data re-use, communication, and the arithmetic intensity of matrix-matrix operations – in DNN models, the *batch size* (batching together several input feature-vectors) acts as a tuning parameter that can be used to increase arithmetic intensity, while in matrix factorizations the *block size* is the typical tuning parameter used to control arithmetic intensity and data re-use.

To demonstrate the difference in matrix shapes and arithmetic intensity between DNNs and numerical data analysis methods, we study the distribution of arithmetic intensity of matrix multiplication operations in a typical benchmark computer vision DNN such a ResNet-50 [118] compared to common matrix decompositions used in data analysis applications, performed on a data matrix from the UCI Human Activity Recognition dataset [18]. The ResNet-50 DNN model consists of 53 convolution layers and one fully connected layer, operating on images from the ImageNet dataset. The Human Activity Recognition dataset consists of 561 numerical attributes (columns) for over 10000 samples (rows). Data matrices often have a "tall-and-skinny" shape due to having large amounts of observations (rows) but a small number of attributes as a result of limited sensor capabilities or other feature measurement facilities. Figure 8.1 illustrates the normalized distribution of arithmetic intensity (as defined in Section 2.4) of matrix-matrix operations (BLAS-3) for ResNet-50 with a default batch size of 1, and the default LAPACK implementations of SVD and QR decomposition (based on blocked Householder transformations) with a default block-size of 32.

We observe that despite matrix-matrix operations being the dominant compute kernel within all the workloads, the shapes and sizes of the matrices represented in the workloads result in very different arithmetic intensities. The arithmetic intensities of matrix multiplication operations within the matrix decomposition workloads are noticeably lower compared to the arithmetic intensities of matrix multiplication within the DNN inference workload. A deeper analysis into the shapes of the matrices within each of the workloads (using a notation of $M \times K$ times $K \times N$ matrix operations) reveals a high diversity of shapes and sizes within the matrix decompositions, compared to a more limited variety within the DNN model. For example, in the QR decomposition we observe many operations with small values of M and N but large values of K, as well as operations with a large value of M but with small values of K and N. The triangular matrix multiplication (`TRMM`) operations in the matrix decompositions particularly represent a series of smaller operations, with the triangular matrix dimensions being equal to the block size (32) and multiplied by relatively small matrices. In contrast, for the ResNet-50 Inference workload we observe that the dimensions of the majority of layers are of the same order of magnitude, with exceptions being the first few layers exhibiting large values of M with small values of K and N, and the last few layers exhibiting small values of M with larger values of K and N.

Increasing the batch-size and block-size tuning parameters can increase the respective arithmetic intensities of both workload types. Nevertheless, tuning parameters comes with trade-offs than can impact different aspects of execution. In the case of DNN models, increasing the batch size improves data-reuse, which can increase inference throughput, but at the same time batching may impact inference latency of a single image due to the increased number of operations, and in cases with small numbers of inputs (for example, classification of a single image) due to the need to wait for a full batch of inputs. In some deployment scenarios, there might not be sufficient input data for a batch size greater than one (for example, low power sensor deployments with occasional wake-up events). This tradeoff in DNN input batching can be considered a system-level tradeoff that depends only on the

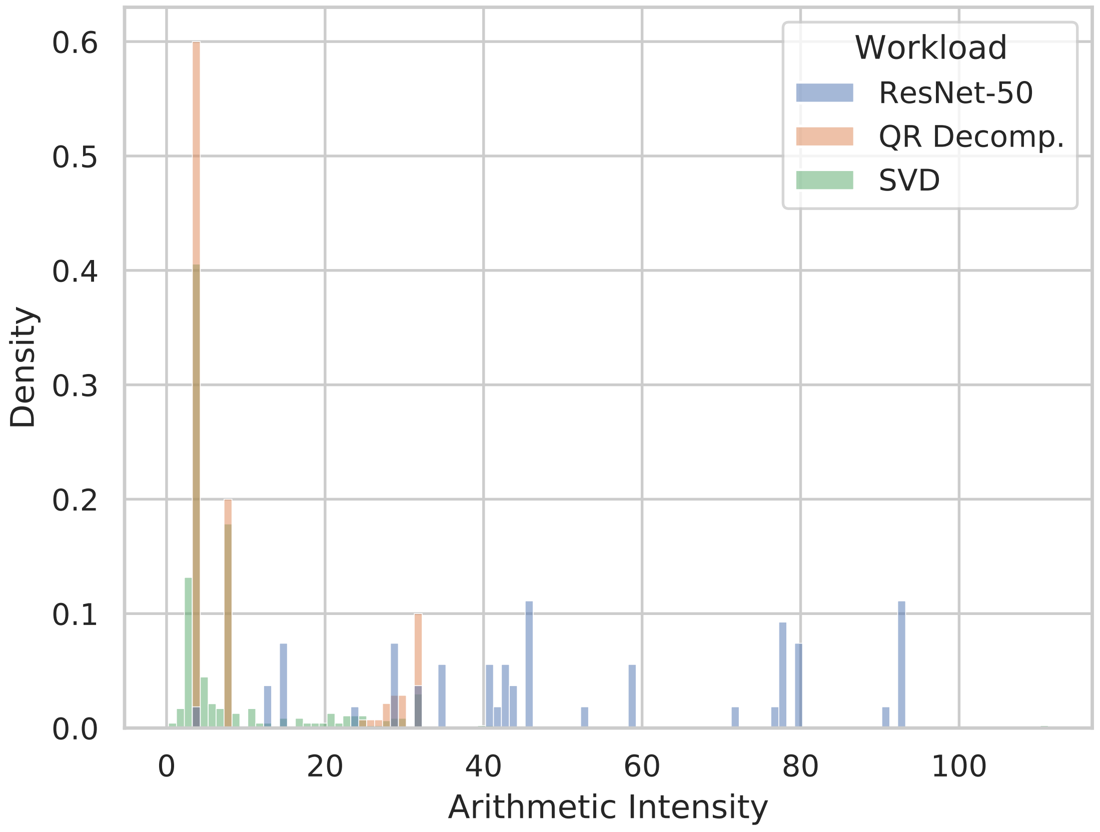


Figure 8.1: Arithmetic intensity histogram of BLAS-3 operations (GEMM, TRMM) within a batch-1 ResNet-50 DNN forward pass, and a blocked-Householder QR decomposition and SVD of the UCI Human Activity Recognition training dataset with block-size 32 (the LAPACK default). Operands are 16-bit floating-point datatypes, and results are 32-bit floating-point datatypes. Density (or "frequency density") is defined as the frequency per unit for a particular range of arithmetic intensities ($frequency/width$), providing for normalized histograms across different workloads.

deployment environment, since the total number of compute operations grows in proportion to the batch-size tuning parameter. Hence, server environments can provide large batches without a significant impact on quality of service, while certain edge deployment scenarios are more limited, as also noted by the MLPerf benchmark [217].

In contrast, in matrix factorizations the block-size tuning parameter may impact the total floating-point operation count that the blocked algorithm performs on the exact same input. As such, it is no longer simply a system-level tradeoff, but rather has a direct impact on the efficiency of an algorithm (in terms of run-time and number of operations) for a particular input matrix. This generates a tradeoff between the desire to increase the block size in order to achieve higher utilization of the matrix engine in the DNN accelerator through higher arithmetic intensity vs. potentially increasing the operation count so much that it outweighs the benefits of faster and more efficient execution on the accelerator. We demonstrate this tradeoff by using the blocked-Householder QR decomposition algorithm. The majority of operation count (FLOP count) analysis of blocked Householder QR decomposition assume that the block size b is significantly smaller than the matrix dimensions m and n, and therefore focus on the highest order operation count, leading to the well-known expression of $2mn^2 - 2n^3/3$ FLOPs. A more detailed analysis of the expression for operation count in a blocked Householder QR decomposition, which takes into account the low-order terms, reveals the dependency on the block size parameter b when it is not orders of magnitude smaller than the matrix dimensions.

We follow the blocked Householder QR algorithm, which uses the $Q = I - YTY^T$ representation to represent the orthogonal matrix Q [231]. A blocked Householder QR decomposition with a block size of b requires approximately n/b steps (for a sufficiently large m), with each step performing a QR panel factorization and a trailing matrix update. The j^{th} blocked Householder QR step with a block size of b requires a panel factorization costing $3b^2(m - jb) - 2b^3/3$ FLOPs [28]. Then, in order to perform a trailing matrix update $I - YTY^T$ (where T is a triangular matrix), the LAPACK function `xLARFB` uses two GEMMs and three TRMMs. The first GEMM multiplies a $b \times (m - jb)$ matrix by a $(m - jb) \times (n - jb)$ matrix, resulting in a $b \times (n - jb)$ matrix. The second GEMM multiplies a $(m - jb) \times b$ matrix by the previous $b \times (n - jb)$ result matrix, resulting in a $(m - jb) \times (n - jb)$ matrix. Therefore, in total, GEMMs contribute $4b(m - jb)(n - jb)$ FLOPs to the total FLOP count of the j^{th} step. The TRMMs contribute another $2b^2(n - jb)/2$ each, resulting in an additional $3b^2(n - jb)$ FLOPs in the j^{th} step. Since a blocked Householder QR decomposition requires n/b steps, an approximate expression for the total number of FLOPs is:

$$S_{HouseQR}(b) = \sum_{j=0}^{\frac{n}{b}} \left[3b^2(m - jb) - \frac{2}{3}b^3 + 4b(m - jb)(n - jb) + 3b^2(n - jb) \right] \tag{8.1}$$

$$= 2mn^2 - \frac{2}{3}n^3 + 5bmn + \frac{3}{2}bn^2 + \frac{3}{2}nb^2 + 3b^2m - \frac{2}{3}b^3 \tag{8.2}$$

Under an assumption that $m, n >> b$, we observe that this expression indeed matches the high-order result of $2mn^2 - 2n^3/3$ FLOPs. In the open-source LAPACK implementation, the default block size parameter for QR decomposition and SVD is 32. This default block size has been empirically found to work well for a large number of processors, but can also be tuned for other processors if needed using the LAPACK `ilaenv` parameter tuning function. If we plug the block size value of 32 to the expression in Equation 8.2, we see that a block size of 32 contributes approximately an additional $(160mn + 48n^2 + 1.5 \cdot 10^3 \cdot n + 3 \cdot 10^3 \cdot m - 2 \cdot 10^4)$ FLOPs (in addition to the $2mn^2 - 2n^3/3$ high-order operations), which for matrix dimensions greater than 1000 are indeed low-order terms, making this a worthy overhead for the potential benefit of reduced communication and greater data re-use.

However, the benefit of using matrix engines in DNN accelerators for such workloads depends on the utilization that can be extracted from the accelerators, and in-turn, this utilization depends on the arithmetic intensity of *each* of the matrix operations which get executed on the accelerator. Since the dimensions of the GEMM operations in the blocked Householder QR decomposition are $b \times (m - jb)$ times $(m - jb) \times (n - jb)$ and $(m - jb) \times b$ times $b \times (n - jb)$, a small b dimension limits the arithmetic intensity and data re-use within a dedicated matrix multiplication accelerator, particularly within the first GEMM in each Householder step. A simple way to increase the utilization of the accelerator would be to increase the arithmetic intensity of the factorization by using a larger block size. However, as noted by this analysis and the expression in Equation 8.2, this would also increase the number of operations that need to be performed, potentially negating any gains from increased arithmetic intensity. Specifically, if b is not at least an order of magnitude smaller than m and n, all the terms involving b become high-order cubic terms. For example, if we increase the block size to 128 and plug it into the equation, we see that the blocking overhead grows to an additional $(640mn + 192n^2 + 2.5 \cdot 10^4 \cdot n + 5 \cdot 10^4 \cdot m - 2 \cdot 10^6)$ operations, which means that now the matrix dimensions need to be at least of order 10^4 to overcome the additional FLOPs incurred by a block size of 128. In Chapter 7, we observed that the custom BLAS implementation using 8×8 Gemmini requires matrix dimensions of over 1000 to reach its peak utilization, demonstrating the challenge of picking a block size that would be able to efficiently utilize the matrix accelerator while maintaining low blocking overheads to achieve optimal performance. Zhang et al. [286] make a similar observation within the context of NVIDIA GPU tensor cores, and demonstrate the decrease in performance beyond a certain optimal block size tuned for NVIDIA GPUs.

The difference in the arithmetic intensity distribution profiles of matrix decomposition workloads vs. DNN inference workloads exposes nuances in the co-design and implementation of high-performance numerical data analysis algorithms and hardware matrix engines. Figure 8.2 illustrates the roofline model [277] of an 8×8 matrix engine (with bfloat16 operands and single-precision accumulators), compared to reference vector units (128-bit datapath, and 256-bit datapath) and a scalar CPU. The arithmetic intensity of matrix multiplication for several matrix shapes is noted on the diagram for reference, with the shared dimension remaining constant (1000), while increasing the product dimensions (emulating increasing the block size tuning parameter). As Figure 8.2 indicates, while blocking factors of 32 may well be within the realm of compute-bound problems for traditional CPUs and vector units, modern matrix engines which were designed for deep learning models require a higher level of arithmetic intensity to enable peak utilization. As such, blocking factors of 32 can be on the boundary of the bandwidth-bound regime for certain problem sizes, making them sensitive to scheduling decisions and memory system dynamics.

Heterogeneous matrix shapes and sizes come into play not only within matrix decomposition algorithms, but also within higher-level layers of the data analysis software stack. For example, even in cases which require performing simple matrix multiplication of a full data

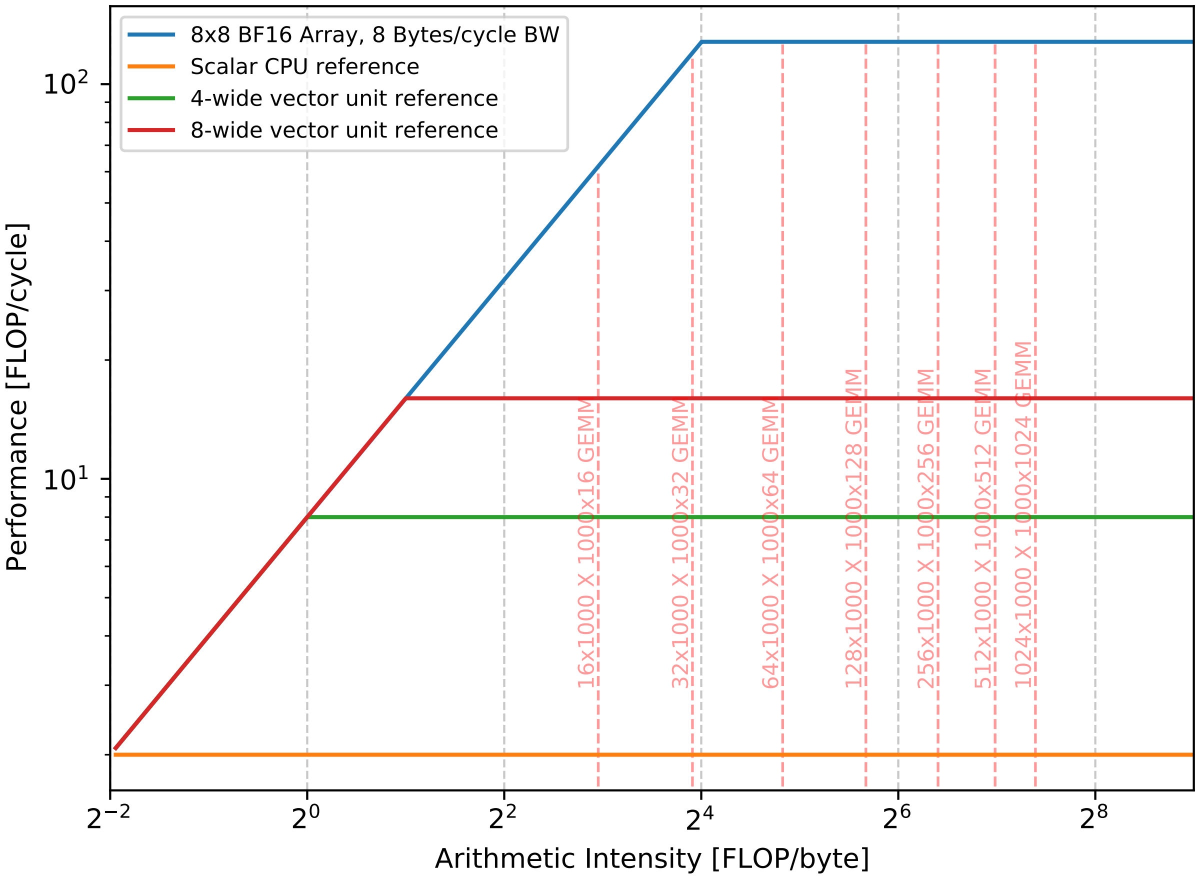


Figure 8.2: Theoretical peak performance roofline model for a Gemmini matrix engine, in comparison to reference CPU and vector units.

matrix, such as in the case of k-means clustering, “tall-and-skinny” data matrices can lead to low accelerator utilization due to lower arithmetic intensity of the multiplication operation when compared to square matrices.

Matrix Sizes

Besides cache-blocking, alternative communication-reduction techniques often used for matrix decompositions are recursive algorithms. For example, the LAPACK implementation of LU decomposition uses Sivan Toledo’s recursive algorithm [258]. A technique proposed for QR decomposition by Zhang et al. to address the problem of limited block sizes on GPU Tensor Cores is also recursive in nature [286]. Recursive algorithms, while providing some sense of spatial and temporal locality, expose a different set of challenges. Recursive algorithms can result in a large number of small matrix-matrix operations, especially when they get close to the base of the recursion stopping condition. Small matrix-matrix operations result in low arithmetic intensity since the surface-to-volume data re-use ratio decreases significantly in small matrix dimensions. Moreover, small matrix-matrix operations add an additional challenging aspect beyond arithmetic intensity, as the size of the matrix operation may be smaller than the size of the compute array within the DNN accelerator matrix engine. In fact, a quick analysis of the default LU decomposition implementation within LAPACK on a 1600×1600 matrix observed 1500 GEMM calls where at least one of the operand matrices was smaller than 16×16, while having only 100 GEMM operation calls where both operand matrices are larger than 16×16. When one of the operand matrices is smaller than the size of the compute array, the accelerator is fundamentally under-utilized, making memory operations and arithmetic intensity a second-order consideration when addressing accelerator utilization. This type of utilization deficiency is simple to resolve by not running these GEMM operations on an accelerator, but rather running them on the CPU. However, the decision regarding which hardware type to run a GEMM operation on must be considered within the relevant software stack.

Small matrices are not only the domain of recursive algorithms. Even within blocked matrix decomposition algorithms, the sizes of the matrix-matrix operations often get smaller as the algorithm progresses and the trailing matrix gets smaller. Even in cases where the small matrices are larger than the size of the compute array, small matrices can be especially challenging for accelerators which use latency-hiding techniques to support high-throughput. This is since the start-up latency cost of moving operand data from main memory into the accelerator cannot get amortized across large matrix sizes and cannot be hidden using double buffering techniques, making such latency-hiding methods ineffective.

Small matrices have been an area of research within the high-performance computing research community, with libraries such as LIBXSMM [119] and BLASFEO [91] highlighting high-performance BLAS operations on small matrices, generally defined as matrices with dimensions M, N, K where $(MNK)^{1/3} < 128$. Such libraries are often meticulously hand-scheduled for various matrix dimensions, with a mix of code generation and manually unrolled loops. LIBXSMM also includes support for the Intel AMX matrix extensions.

In order to obtain maximal utilization using matrix engines within both compute-bound and memory-bound regimes across a diversity of matrix shapes and sizes, the scheduling of compute and memory operations on these accelerators may need to change or be better customized to meet the requirement of this range of workloads. In Section 8.4 we discuss how the scheduling of these operations within a DNN accelerator controller can impact the utilization of the accelerator across different matrix shapes and sizes.

Transposition and Data Layout

Numerical linear algebra operations often require matrix transpositions. For example, the singular value decomposition factorizes a matrix to $U\Sigma V^T$, which means that extracting the right-side singular vectors may require transposition. Similarly, the blocked Householder QR decomposition, which uses the $I - YTY^T$ representation of a product of Householder transformations, requires a transposition of the triangular matrix T. In fact, as noted in Chapter 7, the matrix operand arguments in the BLAS interface definitions are accompanied with flags indicating whether they should be transposed or not, in order to accommodate such scenarios.

The vast majority of deep learning inference workloads do not require any support for transposition. Weight matrices are often efficiently laid-out in memory since they are known during compilation time, and activation matrices propagate through the network in their native layout. As a result, many deep learning accelerators do not need and do not have custom hardware support for matrix transposition. Nevertheless, training of deep neural networks typically does require matrix transpositions as part of the backwards pass of back-propagation, since the computation of partial derivatives for gradient descent may require such transpositions. Hence, accelerators which target DNN training such as the TPUv2 and TPUv3 have added zero-overhead hardware support for matrix transposition [148].

Transposition is also necessary when using software libraries which assume different data layout formats. For example, while the C/C++ language assumes data is stored in row-major layout, the Fortran BLAS API assumes data is stored in a column-major layout. Such discrepancies can often be resolved through appropriate usage of transposition flags.

Transposition is a memory-intensive operation, since it consists only of data movement with no actual arithmetic computation. However, through efficient use of the accelerator memory hierarchy, zero-overhead hardware transposition can become a simple and efficient operation, as demonstrated with Gemmini in Chapter 7. Figure 7.10 illustrates that a simple hardware transposer unit can provide complete overlap of operations to match the throughput of the arithmetic systolic array. By toggling the direction of dataflow within the transposer array (using the `dir` direction-selection signal), the transposer can select whether data flows from top-to-bottom or from left-to-right. The transposer switches the direction of dataflow every `DIM` cycles, with this switch of direction acting as the transposition operation. The use of a 2-D transposer array enables complete overlap between two transposition operations since the input of the second transposition is being fed into the array while the transposed result of the first transposition is fed out of the array. This simple yet effective hardware unit allows the Gemmini DNN matrix engine to handle a wide variety of matrix layouts and transposition combinations, supporting the diverse requirements of numerical data analysis applications as well as deep learning inference and training.

8.3 Matrix Engine Controllers

DNN accelerators typically need to utilize a controller in order to manage memory transactions and compute resources within the accelerator. These controllers can span the spectrum between full programmable processors to fixed hardware finite state machines (FSM), including potential hierarchies of controllers within an accelerator, enabling different levels of programmability [51, 50, 238, 149, 241].

These controllers typically divide large, arbitrarily-sized matrix operations into a series of smaller operations which can be scheduled onto small fixed-size compute arrays. For example, the Gemmini accelerator is equipped with a FSM which divides a large matrix multiplication problem (defined as $C = AB + D$), into a sequence of smaller matrix multiplications executed on a spatial array (of dimension $DIM \times DIM$). Each of the smaller operations, which we refer to as individual "Gemmini commands", can be at most $DIM \times DIM$ large, and is issued to either an execution queue, which performs these small matrix multiplications, or a load or store queue, which performs DMA transactions.

Instead of a FSM, a small fully-programmable processor can be used to schedule matrix multiplication and memory operations with software flexibility, but the software running on such processors typically lacks the dynamic runtime information that is easily available to hardware-managed controllers. For example, a hardware-managed controller can easily schedule matrix multiplication operations to avoid dependencies on memory operations which are currently stalling, while software-schedules will require more complex techniques to react to such stalls. Furthermore, programmable software schedules can often face bottlenecks due to limited instruction issue bandwidth by the control processor and software overheads of address calculation and command configuration.

The Gemmini accelerator can be programmed by both using fine-grained instructions sent by a software-programmable control processor, as well as by using coarse-grained instructions which use the FSM-based controller. The key difference between the Gemmini fine-grained instructions and Gemmini coarse-grained instructions is the size of the matrices they operate on: Gemmini fine-grained instructions operate on matrices of the size of the spatial compute array ($DIM \times DIM$, the granularity of Gemmini commands), while Gemmini coarse-grained instructions can operate on variable-size matrices up to the size of the accelerator private scratchpad memory. As such, the coarse-grained instructions require the issue and scheduling of a series of fine-grained commands. The hardware controller is responsible for issuing and scheduling fine-grained compute and memory micro-operations (commands), maintaining dependency ordering and coherency, and ensuring forward progress of the program.

The software overheads of address computation, custom instruction configuration and encoding, and control flow, often require high instruction-issue bandwidth by a software-programmable control processor. In contrast, a hardware FSM can compute multiple addresses and make multiple scheduling decisions in parallel in hardware, as well as efficiently encode and configure custom operations with zero-overhead. Furthermore, even with sufficient instruction-issue bandwidth, control software needs to include careful software pipelining, loop unrolling, and instruction interleaving in order to overcome the software overheads of the control processor, in contrast to hardware-based control. In our experience, we were not able to match the throughput of the Gemmini hardware-based controller by using a software-based schedule running on a Rocket in-order control processor, primarily due to the limited instruction issue bandwidth of the Rocket core.

Figure 8.3 illustrates the high-level structure of the Gemmini matrix multiplication hardware controller. The controller issues loop iterators based on a tiling schedule encoded in the loop unroller FSM. The loop iterators are fed into address generation units, which gen-

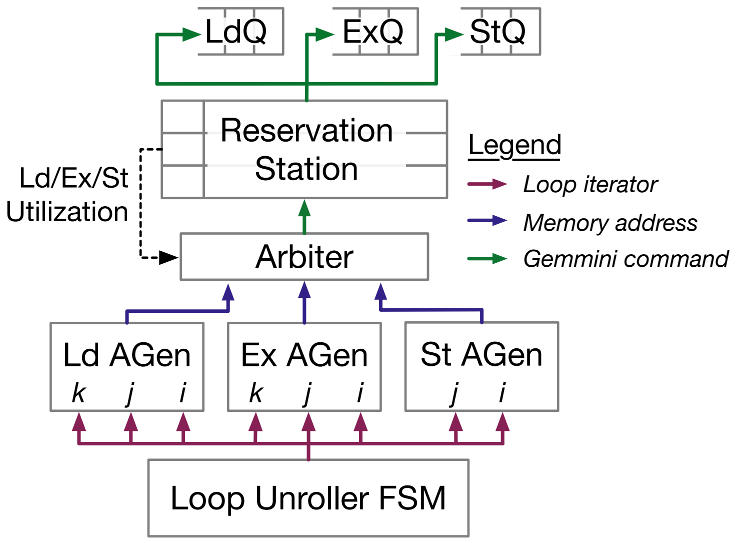


Figure 8.3: Gemmini matrix multiplication hardware finite state machine controller.

erate DMA commands that move operands from main memory to the accelerator private scratchpad, execution commands based on addresses within the private scratchpad, and DMA commands which move the results from the private accumulators back to main memory. Commands are arbitrated based on feedback from the controller reservation station in order to ensure a balanced mix of commands within the decoupled access-execute architecture. The reservation station then dispatches commands to the execution queue and the DMA load and store queues.

For matrix multiplication operations, the FSM implements the following loop ordering:

```
for (k = 0; k < K; k += DIM)
  for (j = 0; j < N; j += DIM)
    for (i = 0; i < M; i += DIM)
```

In this loop ordering, the dimensions of the first operand matrix (A) are $M \times K$, and the dimensions of the second operand matrix (B) are $K \times N$. In this notation, M, N, K are the dimensions of the operand matrices, while i, j, k are the loop iterators generated by the FSM. It is important to note that the controller FSM uses the shared dimension iterator k as the outermost loop iterator in order to maximize data re-use within the accelerator SRAM accumulators. Gemmini also includes an additional FSM which generates loop iterators for convolution operations using a similar mechanism.

8.4 Matrix Engine Controller Scheduling

The scheduling of memory and compute operations on accelerator resources has a direct impact on the overall utilization of the accelerator. The scheduling of matrix multiplications operations on CPUs has been extensively researched, with analysis evaluating choices and placements of stationary data and streaming data across the memory hierarchy [102]. The scheduling of matrix multiplication on DNN accelerators exhibits similar characteristics, with the addition of constraints set by the accelerator's spatial resources and fixed interconnects, as well as the private accelerator memory management policy [128].

Recent work by Jeong et al. [145] has identified some of the under-utilization challenges of systolic array accelerators that are tightly integrated with CPUs. Specifically, they identify under-utilization as a result of the dimensions of the matrix multiplication problem (mapping inefficiency), memory latency, and pipeline fill/drain delays. These are challenges that are amplified in the context of matrix engines that are tightly integrated with CPUs and CPU memory systems due to smaller private memories (register files or scratchpads) and smaller spatial array sizes associated with them. Both the Gemmini matrix engine controller as well as the techniques proposed by Jeong et al. help address many of the causes of systolic array pipeline fill/drain under-utilization through overlapping and bypassing of operations. However, the diversity of matrix shapes and sizes exhibited by numerical data analysis workloads highlights under-utilization dominated by the other two factors - memory latency and mapping inefficiency due to problem dimensions.

Unlike DNN models, where tensor shapes and weight values are known at compilation time and can therefore be optimized and scheduled based on static analysis, LAPACK and BLAS are traditionally used as dynamically linked libraries which can take arbitrary matrix shapes as inputs during runtime. This runtime flexibility requirement entails that the DNN accelerator matrix engine controller needs to be able to make independent matrix compute and memory operations scheduling decisions, without relying on an optimal statically analyzed software schedule.

We assume that the systolic array is of size $DIM \times DIM$. In order to perform a matrix multiplication operation, $C = AB + D$, in which the operand matrix dimensions are $M \times K$ and $K \times N$ and the output matrix dimensions are $M \times N$, the matrix engine controller must schedule a series of $DIM \times DIM$ computes operations and DMA transactions. This schedule is constrained by the size of the accelerator's private memory, the memory bandwidth of the DMA, and the throughput of the compute array. With an ideal schedule, the accelerator throughput should reach the performance bounds set by the roofline model in Figure 8.2.

The Gemmini controller uses hardware-managed double buffering as a latency-hiding technique. As such, the private scratchpad memory and accumulators are split by the controller into two partitions, where the data in one partition gets used for computations while the data in the other partition gets used by the DMA for moving data to and from main memory. This approach can sustain full utilization of the accelerator as long as the number of cycles it takes to perform computation on the data in a partition is longer than the number of cycles it takes to move data from main memory into the second partition. However, this technique is not able to hide latency in cases where the operand matrices are smaller than the size of the accelerator private scratchpad memory. In such cases, there is not enough data to overlap the compute of one partition with the memory movement of the other partition. In those scenarios, the utilization of the accelerator becomes sensitive to the scheduling decisions made by the controller.

While Gemmini is capable of supporting both output-stationary (OS) and weight-stationary (WS) systolic array dataflows, we note that the majority of commercial systolic-array-based accelerators support only weight-stationary dataflow, so we will focus primarily on this dataflow configuration. In a WS dataflow, $DIM \times DIM$ data elements of the second operand matrix (the B matrix, also referred to as a "weights" matrix in deep learning workloads) are resident within the systolic array processing units (PEs), while data elements of the first operand matrix (the A matrix, or "activations" matrix in deep learning workloads) stream into the array from the private scratchpad memory. The output data elements (matrix C) propagate through the systolic array and accumulate into a wide accumulator SRAM within the accelerator [1].

As such, when the target matrix operation is of high arithmetic intensity, given this streaming schedule the controller should schedule more memory operations for the A matrix rather than the B matrix in order to minimize latency, since a single $DIM \times DIM$ segment of the B matrix will get re-used across multiple segments of the A matrix while that $DIM \times DIM$ segment of the B matrix is resident in the compute array. This is a similar streaming pattern to the one used in the Goto Algorithm for high-performance CPU software BLAS implementations [102]. However, when the matrix operation is of low arithmetic intensity, there is less data re-use within the accelerator memory hierarchy. In fact, as the roofline model dictates, the operation may be bound by memory bandwidth. Hence, while the dimensions of a matrix might be large enough to fully utilize the compute array ($M, N, K >= DIM$), if the operand matrix shapes are not conducive to data re-use within the accelerator memory hierarchy (and specifically, the accelerator private memory), it may be more beneficial to schedule them differently. For example, for the case where $K >> N$, it would be more beneficial to schedule A and B memory operations at a "symmetric" rate, as opposed to the method of choice for high arithmetic intensity operations of an "asymmetric" rate with more A memory operations due to the streaming nature of A in a WS dataflow systolic array.

We demonstrate this idea by co-designing the Gemmini hardware controller with a programmable arbitration parameter for setting the ratio of DMA transactions between the A matrix and the B matrix. Figure 8.4 illustrates a more detailed schematic of the load address generator listed in Figure 8.3. An arbiter, controlled by an arbitration parameter listed as `WeightA`, regulates a weighted arbitration scheme for DMA transactions generated by two address generators for each of the operand matrices in the matrix multiplication: the A matrix address generator (LdA AGen) and the B matrix address generator (LdB AGen). For example, if the value of the `WeightA` parameter is 3, then for every one DMA transaction dispatched to the reservation station by the B matrix address generator, there will be three DMA transactions dispatched to the reservation station by the A matrix address

[1] We note that if we were to use an OS dataflow, data re-use is limited to the size of the compute array rather than an accumulator SRAM. As a result the data re-use of both the A and B matrices is significantly lower.

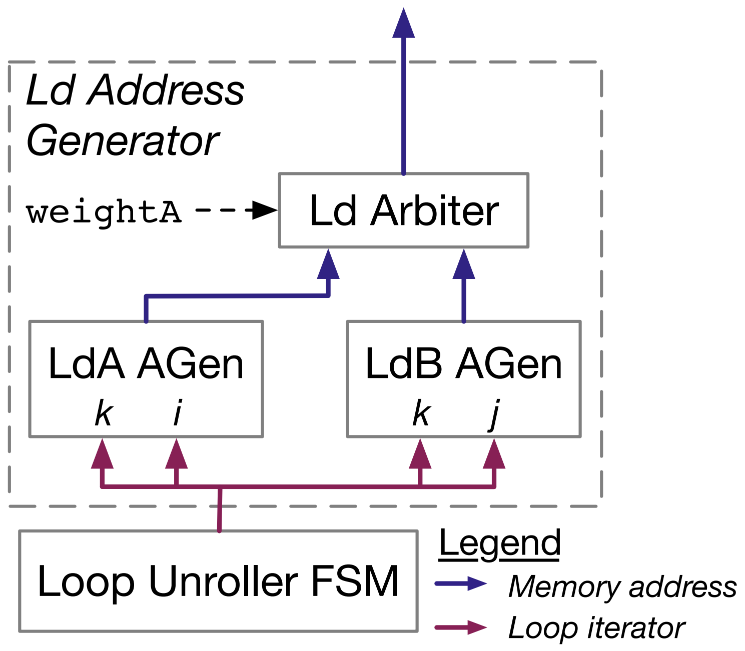


Figure 8.4: Arbitration mechanism between A matrix and B matrix loads in Gemmini's load-address generator (which is part of the matrix multiplication FSM illustrated in Figure 8.3).

generator. This parameter is software-programmable, enabling co-design between software and hardware when exploring the utilization of Gemmini for different matrix shapes and sizes.

Figure 8.5 characterizes the utilization of an 8×8 Gemmini compute array across different matrix shapes, including shapes extracted directly from the ResNet-50 DNN model or from matrix decomposition algorithms, using different values of the programmable `WeightA` parameter. Tall-and-skinny matrix shapes which fit into the private accelerator scratchpad memory exhibit the highest variability from this programmable static scheduling parameter, with up to 20% difference in utilization. Large matrices do not exhibit any sensitivity to this scheduling parameter, as their performance is completely dominated by double-buffering. Other matrix shapes present variable degrees of sensitivity to this scheduling parameter, with few percent differences in utilization.

An additional observation from Figure 8.5 is that there is no single value of `WeightA` that can provide top performance across a diversity of matrix shapes. We observe that within the matrix shapes presented in Figure 8.5, each of the five evaluated values of `WeightA` was found to demonstrate the highest utilization for at least one matrix shape. This leads to the understanding that more adaptive support for such a static scheduling parameter could provide measurable benefits when processing the variety of matrix shapes and sizes found in numerical data analysis applications.

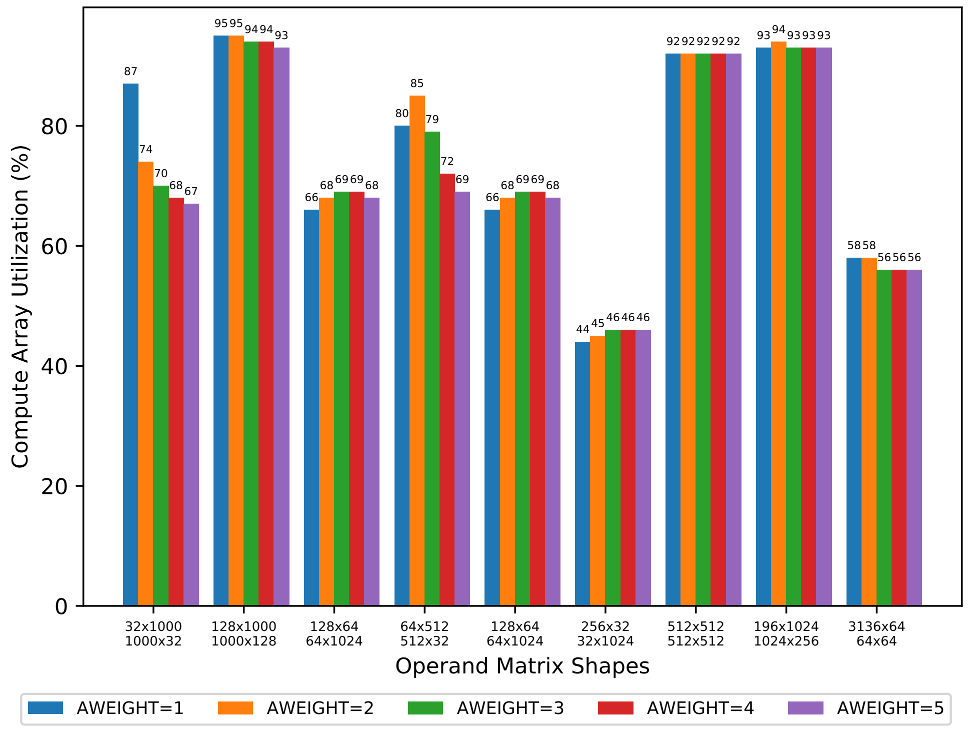


Figure 8.5: Gemmini utilization under different hardware controller operand matrix command arbitration parameter values

8.5 Memory Access Tail Latency

While the Gemmini hardware controller FSM can potentially improve utilization by addressing static scheduling decisions based on the shapes of the matrices, utilization can also be impacted by dynamic properties of shared resources within the SoC.

In our Chipyard SoC evaluation system, the main shared resource used by Gemmini is the SoC memory system. The Gemmini DMA can generate many load and store requests to the TileLink SoC memory system, and the responses to these requests can return in variable latency and out-of-order. The Gemmini DMA keeps track of these memory transactions and handles their ordering upon their return. Unlike the DMA, the Gemmini execution command queue was designed only for in-order execution. This is an efficient design point under the assumption of double-buffering and high data re-use within the scratchpad, which together mean that data should be readily available within the private scratchpad when commands are dispatched to the execution command queue. However, when the operand matrices are smaller than the size of the accelerator private memory and do not enable double-buffering, the variable latency of the shared memory system can impact the utilization of the compute array due to front-of-line blocking of the execution command queue.

As an example, Figure 8.6 illustrates the results of an experiment with two different SoC

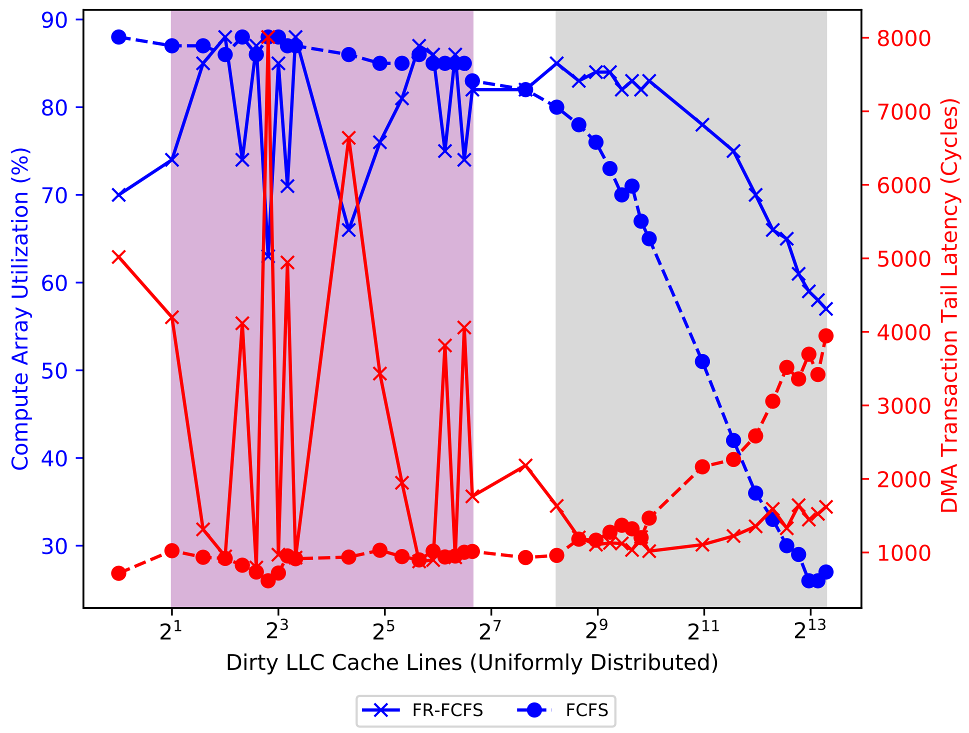


Figure 8.6: Gemmini utilization and observed DMA tail latency across different shared SoC memory system scenarios: different number of dirty cache lines within the L2 cache, and different memory system schedulers for a 32×1000 times 1000×32 matrix multiplication.

shared memory systems, with different internal scheduling policies. In this experiment, we vary the number of dirty cache lines within the shared L2 cache, which acts as a shared resource within the SoC, in order to generate different eviction scenarios which could potentially impact latency within the shared memory system. Each repetition of the experiment executed a 32×1000 times 1000×32 matrix multiplication. As expected, the different number of dirty cache lines indeed generate different tail latencies, as observed by the Gemmini DMA (illustrated by the red lines in Figure 8.6). As a result, Gemmini experiences head-of-line blocking within the in-order execution command queue, which degrades Gemmini utilization (illustrated by the blue lines in Figure 8.6). Figure 8.6 also shows how this type of variability in tail latency can be a result of different memory scheduling policies by the SoC DRAM controller, for example First-Come-First-Serve (FCFS) or First-Ready-First-Come-First-Serve (FR-FCFS), and the way they interact with the rest of the SoC memory system. The FCFS is a fair scheduling policy with predictable patterns, and therefore results in relatively consistent tail latencies compared to the FR-FCFS scheduling policy which exhibits higher variability due to the policy's attempt to optimize for open DRAM row-buffers. The regions highlighted in grey and purple in Figure 8.6 demonstrate different tail latency patterns as a result of the underlying memory systems (with the grey region being more predictable than the purple region). The grey region has a higher effect on utilization since the number of long-latency transaction occurrences is higher.

While in the example in this experiment the variable DMA transaction tail latency (as observed by Gemmini) is generated by L2 cache evictions and various SoC memory access scheduling policies, this type of variable DMA transaction tail latency can occur for a variety of reasons, including quality of service (QoS) policies across SoC buses and fabrics, as well as interrupts and other asynchronous events within the system.

When the operand matrices are big enough to be double-buffered, this type of tail latency can be hidden. For example, when using a 256 KiB scratchpad memory, operands with the sizes 128×1000 and 1000×128 in a bfloat16 representation (which total approximately 512 KiB) would be too big to fit in the private scratchpad, and will therefore be double-buffered, with each double-buffering partition being 128 KiB. If we assume the partition is divided equally between both operands, this means that each operand could fit a 128×256 or 256×128 tile, allowing for sufficient data re-use to drive almost 4 million MAC operations, which on an 8×8 systolic array could hide a DMA tail latency of over 65,000 cycles. This simple calculation demonstrates why an in-order execution command queue is generally a sufficient and efficient choice for DNN accelerators. Nevertheless, when considering the diversity of matrix shapes and sizes that would need to be supported for supplemental-use of such matrix engines for numerical data analysis workloads, we conclude that out-of-order execution command issue should be considered.

8.6 Co-Design of Matrix Engine Controller

We demonstrate how a HW/SW co-design analysis of numerical analysis workloads on DNN matrix engines, and specifically, the observation regarding more relaxed matrix shape and size ranges, can lead to simple and inexpensive customizations and improvements within such DNN matrix controllers, which make them more amenable to use for this class of applications. Such changes can be integrated into generators, allowing for customization of SoCs for multiple classes of application while re-using common core accelerator components.

Software-Configurable and Hardware-Managed Static Scheduling

Accelerator controllers with full processor-based software capabilities are flexible enough to enable any combination of static compute and memory scheduling decisions. However, performing scheduling operations by using processor-based accelerator controllers comes with software overheads of computing addresses, strides, pointers, bound-checking and control flow, which can often be limited by instruction-issue bandwidth and the throughput of the control processor itself. Even in a control processor which is theoretically capable of supporting full-throughput scheduling, implementing such a software schedule requires significant high-performance software expertise in order to correctly perform software pipelining, loop unrolling, and control flow minimization in order to overcome software overheads. In con-

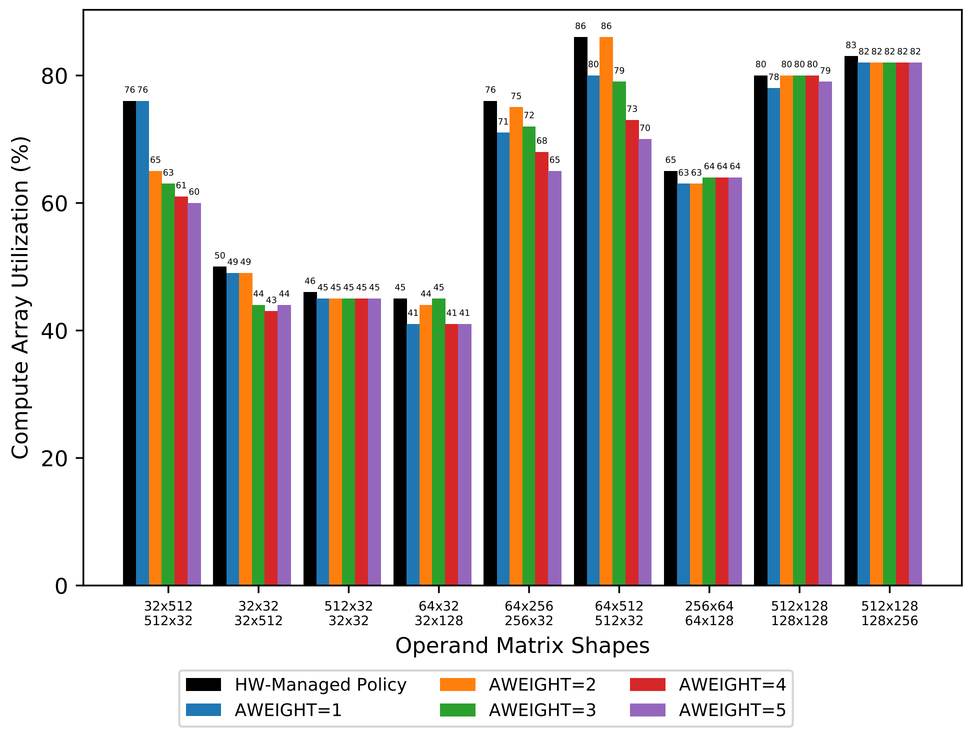


Figure 8.7: Gemmini (8×8) utilization using a hardware-managed static scheduling policy in comparison to different hardware controller operand matrix arbitration parameter values.

trast, fixed hardware controllers, such as the one implemented in Gemmini in the form of an FSM for coarse-grained instructions, perform address calculations, bound-checking, and control flow, all in parallel to issuing operations, resulting in zero-overhead scheduling decisions. Zero-overhead hardware control can also better utilize feedback from the execution pipeline in order to assist in schedule decisions.

In most cases, such fixed hardware controllers in-fact have sufficient information to perform low-cost hardware-managed static scheduling decisions based on the shapes and sizes of the operand matrices. Specifically, we focus on the `WeightA` arbitration parameter highlighted in Section 8.4. While this arbitration parameter could be hard-coded in the hardware FSM, which would result in sufficiently high utilization for most large matrices that are double-buffered, it could also be a programmable parameter that is configured in software by the programmer for a particular matrix size and shape. Alternatively, this arbitration decision can also be set by the hardware controller finite state machine, based on the loop iterators generated by the FSM for a particular matrix shape. This approach, which we refer to as "hardware-managed static scheduling", due to the fact that a hardware-only control loop sets the arbitration decision based on the software-controlled shape of the matrix, would potentially make the DNN accelerator more robust to handling a diversity of small and rectangular-shaped matrices.

We set a relatively simple hardware-managed static scheduling policy within the Gemmini hardware controller. In this policy, the arbiter starts by issuing a load command for the first block of the second operand matrix (B). This is since the systolic array is a weight-stationary systolic array, in which the second operand matrix is the "static" (stationary) operand within the array. The controller policy will then continue to dispatch load commands based on the values of the k iterators within the address generator for the A matrix and the address generator of the B matrix.

The loop unroller FSM maintain independent copies of the loop iterators for the A address generator and B address generator, allowing each address generator to progress autonomously based on the independent iterator values issued to them by the FSM. As such, based on the iterator loop ordering listed in Section 8.4, when the k iterator associated with the B address generator is greater than the value of the k iterator associated with the A address generator, this is an indication that the DMA transactions being issued are related to the inner most loop in the nested loops. Similarly, when the k iterator of the A address generator is greater than the value of the k iterator of the B address generator, this is an indication of a value increment in the middle loop of the nested loops. Therefore, in this hardware-managed static scheduling policy, the transaction arbiter will issue DMA transactions from the B address generator as long as the value of the k iterator associated with the A address generator is greater than the value of the k iterator of the B address generator.

The hardware cost of this adaptive hardware-managed policy is relatively inexpensive, and is primarily reflected in wiring (since the iterator values need to be wired to the arbiter), and a pair of multiplexers and comparators used to implement the adaptive policy decision.

Figure 8.7 presents a comparison of the utilization of the 8×8 Gemmini accelerator when using the hardware-managed static scheduling policy vs. using software programmable values of the `WeightA` parameter. Notably, the hardware-managed static scheduling policy demonstrates equal or better utilization compared to the best software programmable value in each of the evaluated cases. More importantly, the hardware-managed adaptive static scheduling policy achieves this utilization without additional programmer intervention or domain knowledge about the shape of the operand matrices.

Dynamic Scheduling in Matrix Engines

In order to improve dynamic scheduling and alleviate variable-latency head-of-line blocking experienced by small matrix operations in Gemmini due to its integration with shared resources in the SoC, we would like to add out-of-order execution support within Gemmini. Out-of-order execution is a common technique for exploiting ILP in CPUs which face variable-latency pipeline environments. Out-of-order execution helps unblock the execution pipeline when processing a long-latency operation by parallel scheduling of additional independent instructions on other available execution units. Execution of the instructions may be out-of-order, but the instructions commit and update the architectural state in-order. This type of ILP-extraction can be very beneficial in superscalar CPUs which have high diversity of instructions with variable latencies (integer arithmetics, integer multiplication and division, floating-point arithmetics, memory operations, etc.). In contrast, Gemmini has a very small instruction set, consisting primarily of two types of operations: fixed-latency execution operations, and variable latency DMA operations. DMA operations are variable latency due to Gemmini's integration with the coherent SoC memory system which includes a cache hierarchy and coherence protocols.

As such, out-of-order execution within Gemmini does not need to encompass the entire pipeline and all instruction types, but rather only those that may experience head-of-line blocking due to a variable-latency instruction and a data dependency. Specifically, we identify two operation types which would benefit from out-of-order execution within the Gemmini controller:

- Compute (matmul) - Reordering of independent or commutative matrix multiplication and accumulation operations, as a result of variable-latency operand load latency

- Store (mvout) - Reordering of DMA transactions from the Gemmini accumulator to main memory as a result of a reordering of compute operations.

Most importantly, unlike CPUs, the Gemmini matrix engine would not benefit from out-of-order execution of memory load commands, since the Gemmini hardware controller dictates a *static* schedule. The static schedule means that there are no dynamic address computations, which means there are no load-after-load dependencies within the instruction stream. Gemmini's decoupled access-execute design further supports this scheme of independent execution orders of memory and compute operations. Similar patterns have been identified in other data-parallel processors with statically predictable memory addresses. As an example, the Hwacha vector accelerator also implements out-of-order execution of compute operations, while maintaining in-order memory operations.

We implement out-of-order execution in Gemmini for the execution and store command queues. Out-of-order execution is implemented as a generator parameter within Gemmini, enabling designers to choose whether this feature should be included within a particular instance for use for a specific application domain. Since Gemmini currently does not support precise exceptions, the cost of adding support for out-of-order execution of the compute commands consists of simply modifying the in-order execution constraint for compute commands in the reservation station, costing only a couple of logic gates. We evaluate our implementation by running a series of experiments on a 32×1000 by 1000×32 matrix multiplication, in order to evaluate its benefit for small matrices which cannot be double-buffered by the controller. We use a similar methodology as was used in Section 8.5, of using dirty cache lines as a method of inducing variable tail latencies while maintaining complete system integrity (as opposed to isolated trace-driven testing of the accelerator). We center our experiments on small numbers of dirty cache lines (1-10), in order to focus on our variable-latency regime of interest, which was highlighted in purple in Figure 8.6. We generate dirty cache lines in random, uniformly distributed addresses across the memory space, and run 10 experiments with different random seeds for each number of dirty cache lines. Figure 8.8 compares Gem-

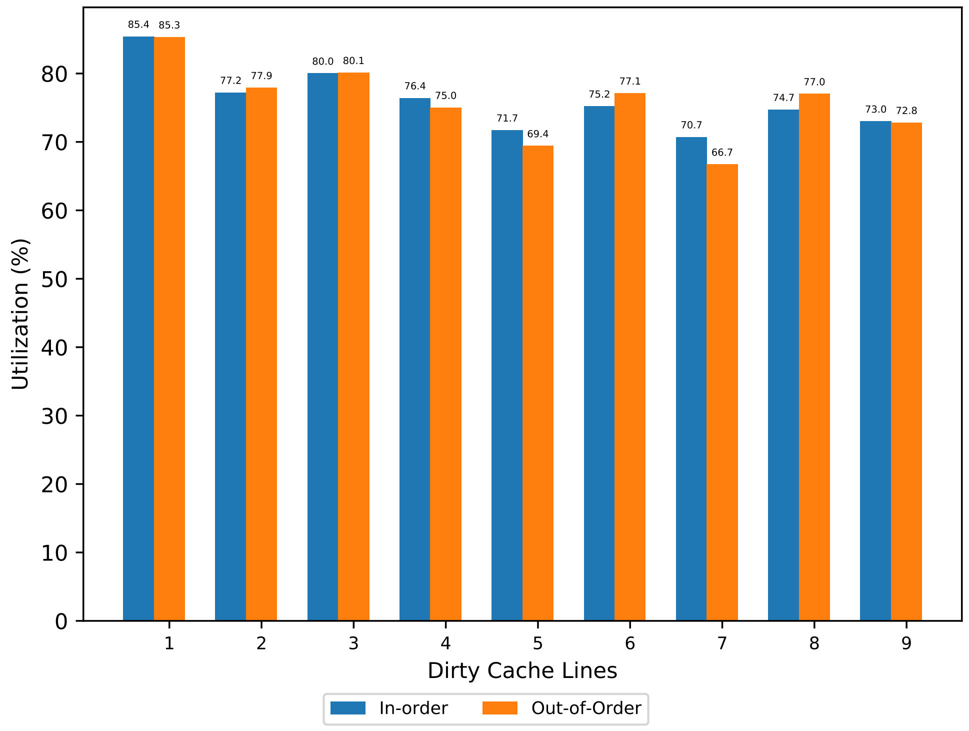


Figure 8.8: Gemmini (8×8) in-order vs. out-of-order execution average utilization across different numbers of shared L2 cache dirty cache lines for a 32×1000 times 1000×32 matrix multiplication.

mini in-order execution vs. out-of-order execution and illustrates the average utilization of each set of experiments for a given number of dirty cache lines.

Notably, our out-of-order execution implementation in Gemmini appears to provide no consistent benefit, and in some scenarios may even degrade average utilization compared to in-order execution. As to the reason for the performance degradation, while out-of-order execution does not change the order and memory access pattern of Gemmini memory load operations, it *does* change their timing, since Gemmini reservation station slots can become available at different times as execution operations progress. This can result in different behaviors across the SoC memory system and interconnect fabric which can sometimes lead to longer tail latencies than if Gemmini would have executed in-order. Similarly, out-of-order memory store operations actually *can* change the memory access patterns (since some accumulations can complete before others, enabling their store operations to be issued earlier), which can also lead to different behaviors across the SoC memory system and interconnect fabric (due to more dirty lines and potential evictions in the caches caused by the store operations), which can also sometimes lead to longer tail latencies for subsequent load operations.

More interestingly, we investigate the reason for the absence of the expected utilization improvement with out-of-order dynamic scheduling. We find that the reason for lack of improvement is due to the static schedule of commands issued by the controller. The nested loop schedule ordering used by the controller is intended to maintain locality across the loops. Since the i and j dimensions of the operand matrices generate the highest level of data reuse, they act as the internal levels of the loop, as also noted in the previous section. However, this choice of loop ordering also means that a sequence of operations which all depend on the same long-latency memory transaction can once again quickly cause head-of-line blocking if the out-of-order issue window is not large enough. This intersection between static scheduling and dynamic scheduling in matrix engine controllers means that additional micro-architectural adjustments are required in order to support efficient static and dynamic scheduling of small matrix operations in DNN accelerators, techniques which may not be necessary in traditional scalar processor out-of-order execution.

Intersection Between Static and Dynamic Scheduling in Matrix Engine Controllers

The Gemmini co-design process for small and diversely-shaped matrices has exposed key design decisions at the intersection of static and dynamic scheduling within the matrix engine controller in the context of a cache-based SoC memory system. If we assume that the size of a shared cache line (CL) is greater than the dimension of the spatial array (and hence, the dimension of compute operations), a static schedule for matrix multiplication should be able to take advantage of spatial locality within the cache line for at least one of the two operand matrices. Nevertheless, this advantage of spatial locality can also become a detriment when tail latencies are caused by the shared cache memory system.

If we assume the granularity of each controller command is a block of $DIM \times DIM$ elements, while a cache line contains CL elements, we can see that if an operand matrix is represented in a row-major layout, a long-latency arrival of data from single cache line could delay the arrival of approximately $\frac{CL}{DIM}$ blocks from that operand matrix. Specifically, if we assume that both operand matrices are represented in a row-major layout, we observe that a long-latency arrival of data from a single cache line would delay the arrival of approximately $\frac{CL}{DIM}$ blocks from the second operand matrix (B), depending on data alignment, since they are all resident in the same cache line (as illustrated in Figure 8.9). As a result, assuming a static schedule based on the loop ordering listed in Section 8.3, we see that a long-latency arrival of a single cache line would delay at least $\frac{CL}{DIM} \times \frac{CL}{DIM}$ compute operation, since outer products expect to re-use the same blocks.

Figure 8.9 illustrates an example in which $\frac{CL}{DIM} = 4$. The single long-latency cache line (highlighted with the label CL) impacts the memory load of four blocks of the B operand matrix, each of which would be multiplied with four blocks of the A operand matrix. As a result, sixteen compute operations would be delayed due to a dependency on a single cache line. These sixteen operations would consume precious slots within the out-of-order execution reservation station, effectively requiring very large reservation stations in order for out-of-

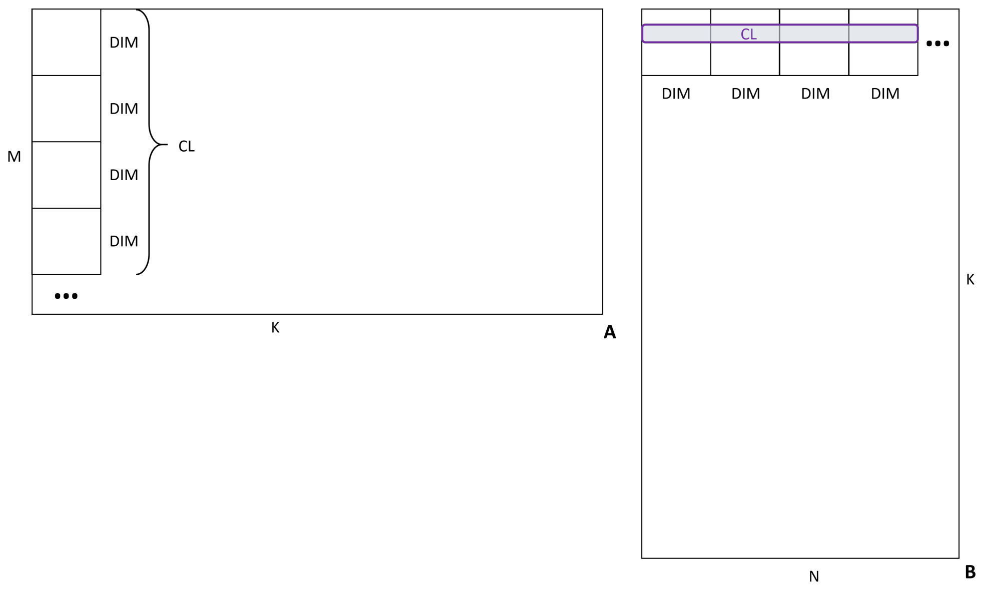


Figure 8.9: Example matrix operation blocks depending on a single cache line.

order execution to be effective in hiding long-latency memory accesses through dynamic scheduling. A typical Gemmini configuration uses reservation stations with approximately 10 slots for the execution queue. As such if 16 slots can be blocked due to a single long-latency memory load, Gemmini can only rarely take advantage of the out-order-order execution, as was indeed observed in in the results in Figure 8.8.

One potential solution to this problem is to increase the size of the execution queue reservation station. However, the all-to-all dynamic dependency tracking within the reservation station makes this an expensive choice in terms of area and energy costs. Another potential solution to this problem would be to change the static schedule such that there is interleaving between operations that depend on different cache lines. However, this static schedule would eliminate the benefits of spatial locality within a cache-based memory system. An alternative solution takes advantage of the observation that a large batch of compute operations all depend on a single group of memory load operations, and therefore it is possible to compress both the load operation commands and compute operation commands into coarser grained commands that would consume less slots in the reservation station structure. This is since the costly wiring and logic used for dynamic dependency tracking within the reservation station structure do not get utilized within this batch of commands since the commands all effectively depend on the same single memory access. This solution could also be described as hierarchical or independent reservation stations for each operation type (load, store, compute), since dependencies can be managed internally within each reservation station as opposed to an all-to-all dependency relationship between all possible commands.

We initially chose to compress matrix operation commands across a single dimension - the i dimension (the number of rows in the output matrix). While matrix operation commands can be compressed across all three dimensions (i, j, k), this would require more intricate static dependency management. Since in a weight-stationary dataflow only one $DIM \times DIM$ block of the second operand matrix (B) is resident in the compute array at a time, the priority is to compress commands across the dimensions of the first operand matrix (A). Since the i dimension is the inner most iterator in the nested loop which pertains to the dimensions of the first operand matrix, we expect that performing command compression across this dimension alone would suffice in providing the required flexibility within the reservation stations, while maintaining an inexpensive implementation.

After implementing command compression, we do not observe a significant change in utilization, and we notice that the majority of commands that fill the execution reservation station are still commands that depend on the j dimension. However, adding this dimension as well to the compression scheme will either remove the B matrix from its stationary position within the systolic array, or require more complex compression and dependency logic (akin to the "larger reservation station" solution). We therefore re-visit the idea of manipulating the static schedule in order to generate some dynamic interleaving of memory and compute operations across different cache lines.

Commutative Hardware-Managed Micro-Threads

After observing that command-compression alone is not able to relieve the excessive pressure on the execution queue reservation station caused by long-latency load operations, we re-visit the idea of interleaving commands which operate on different cache lines. However, we would like to do that while maintaining the weight-stationary dataflow (hence, maintaining the same loop ordering and static schedule), and utilizing as much data locality as possible. We observe that we can take advantage of the commutative nature of accumulation, and the fact that accumulation in matrix multiplication is always performed across the shared dimension (the k dimension), which is the most external loop in our static schedule. We further note that by keeping the static schedule and load operations in their original order, we are able to maintain maximal use of data locality. Therefore, if we take advantage of commutative interleaving across the k dimension of operations which do not depend on the same cache lines only within the execution queue, we can maintain both the WS dataflow and maximal data re-use, while providing a different mix of commands within the execution queue issue window. This can be incorporate into the accelerator in the form of hardware-controlled commutative micro-threading of the execution queue within the accelerator controller.

This idea is similar to an observation suggested by Shomron & Weiser [239] in the context of simultaneous multi-threading (SMT) processing on systolic arrays, in which they note that the SMT threads could be part of the same DNN execution flow, as opposed to independent threads of independent execution flows. Since our hardware accelerator controller manages a single execution flow of matrix multiplication, the controller is able to split this execution flow into multiple hardware-managed micro-threads in an attempt to hide the latency generated by a sequence of data-dependent commands. Notably, these are not full-fledged threads, since memory load and store operations are still performed according to the original

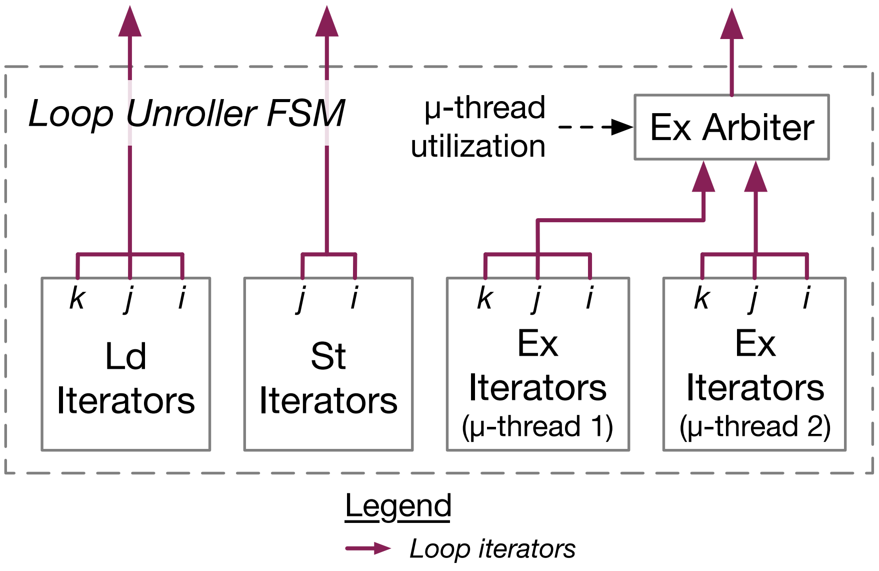


Figure 8.10: Commutative micro-threads within the loop unroller FSM (illustrated in Figure 8.3). "µ-thread utilization" is a feedback channel from the reservation station, and describes the number of instructions from each micro-thread which are stored within the reservation station but which have not yet been issued to the Execution Queue (ExQ).

static schedule. Only compute execution commands can be interleaved using these micro-threads, therefore making them both opportunistic and inexpensive in term of additional required state. The controller generates hardware-managed micro-threads by splitting the nested loops across the most *external* loop-level (the k dimension). The controller maintains the loop iterator indices for each of the micro-threads, and can feed them into the execution address generator, as illustrated in Figure 8.10. Slots are allocated in the execution queue reservation station only for micro-threads for which the relevant memory load commands have already been issued. As such, micro-threads are opportunistic, and do not hurt data locality. These hardware-managed micro-threads are effectively independent within the execution flow, since accumulation within the SRAM accumulators is commutative. Equations 8.4 and 8.5 demonstrate the independence of the hardware-managed micro-threads (for the cases of 2 and T threads, respectively) from the perspective of the execution flow within a single controller-managed matrix multiplication instruction.

$$c_{ij} = \sum_{k=0}^{K}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} \tag{8.3}$$

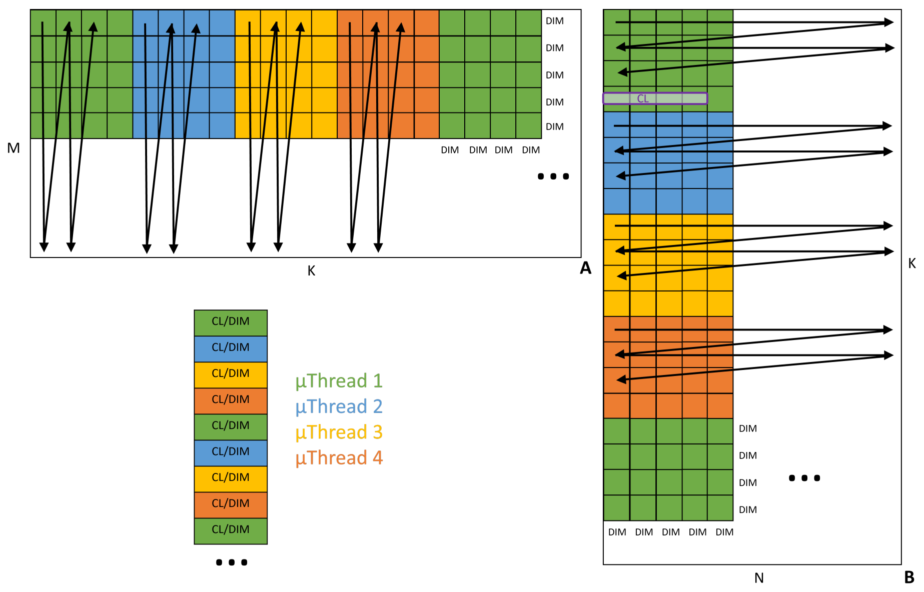
$$= \sum_{k=0}^{(K/2)-1}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} + \sum_{k=K/2}^{K}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} \tag{8.4}$$

$$= \sum_{t=0}^{T}\sum_{k=t\frac{K}{T}}^{(t+1)\frac{K}{T}-1}\sum_{i=0}^{M}\sum_{j=0}^{N} a_{ik}b_{kj} \tag{8.5}$$

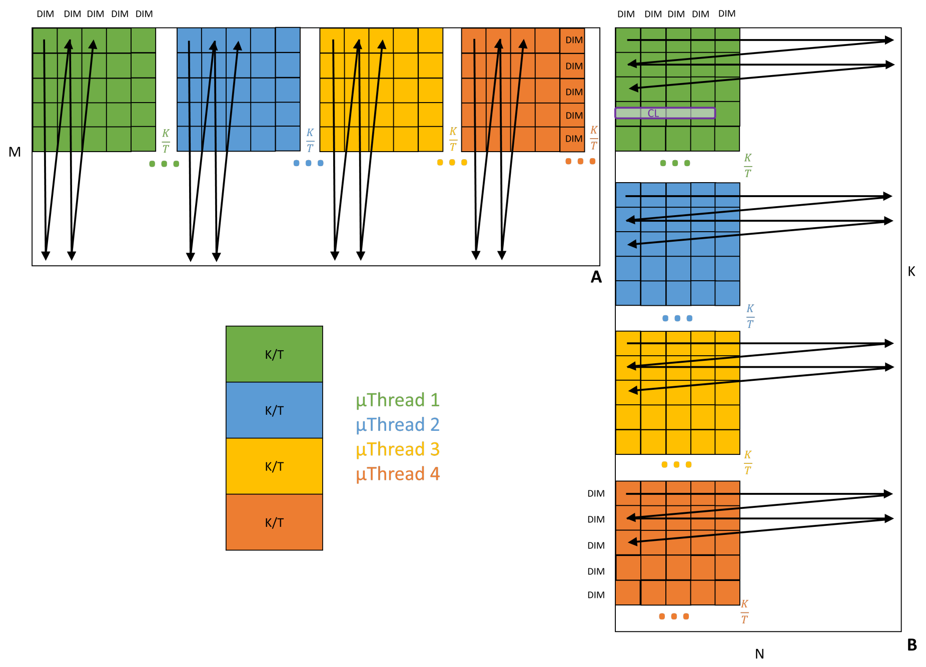
The controller simply manages multiple individual matrix multiplication execution subflows that accumulate into the same accumulator SRAM. By performing this hardware threading only for the execution queues rather than the memory queues, the controller does not need to maintain any additional state other than the indices tracking the state of the FSM generating addresses for execution commands on the systolic array, making this a relatively inexpensive feature.

The commutative property of the micro-threads also enables us to evaluate different thread interleaving schemes. Figure 8.11 illustrates two such schemes: fine-grained micro-thread interleaving, and coarse-grained micro-thread interleaving. In coarse-grained micro-thread interleaving, the k dimension is partitioned into a number of partitions equal to the number of threads (T), with each thread responsible for processing K/T consecutive commands, as illustrated in Figure 8.11b. This is the simplest possible thread partitioning scheme since it is based on a static number of partitions, but it can result in a potential bottleneck since it depends on the progression of memory load command issue within the controller. If memory loads haven't yet been issued for the blocks needed by a thread assigned to the "high indices" of the k dimension, that thread will remain idle and will not be able to overcome head-of-line blocking in other threads. In fine-grained micro-thread interleaving, the k dimension is partitioned into $(K \cdot DIM)/CL$ partitions, where each partition consists of CL/DIM blocks of size $DIM \times DIM$. Each partition is assigned to a different micro-thread in a periodic pattern, as illustrated in Figure 8.11a. Each thread is responsible for processing CL/DIM consecutive commands before switching to the next partition it is assigned to. This partitioning scheme is slightly more complex than the coarse-grained partitioning scheme since it requires tracking a variable and larger number of partitions. However, it provides for a more balanced distribution of partitions across the micro-threads, which can potentially help overcome the problem of reservation-station blocking by dependencies on a single cache line. We note that in this scheme, each micro-thread is responsible for handling CL/DIM commands, since we know all of those commands will depend on the same cache line, and therefore will not benefit from further internal micro-threading.

We evaluate the two micro-thread interleaving schemes, with the results illustrated in Figure 8.12. Figure 8.12 once again demonstrates the average results of a series of experiments on a 32×1000 by 1000×32 matrix multiplication. We observe that fine-grained



(a) Gemmini micro-threads fine-grained interleaving. Each thread processes CL/DIM consecutive commands, with thread assignment repeating in a periodic pattern.



(b) Gemmini micro-threads coarse-grained interleaving. The k dimension is partitioned into T (number of threads) contiguous blocks of consecutive operations, each assigned to a micro-thread.

Figure 8.11: Gemmini micro-threads interleaving schemes[1]

[1] Arrows represent the progression of each micro-thread according to the static schedule nested loop ordering.

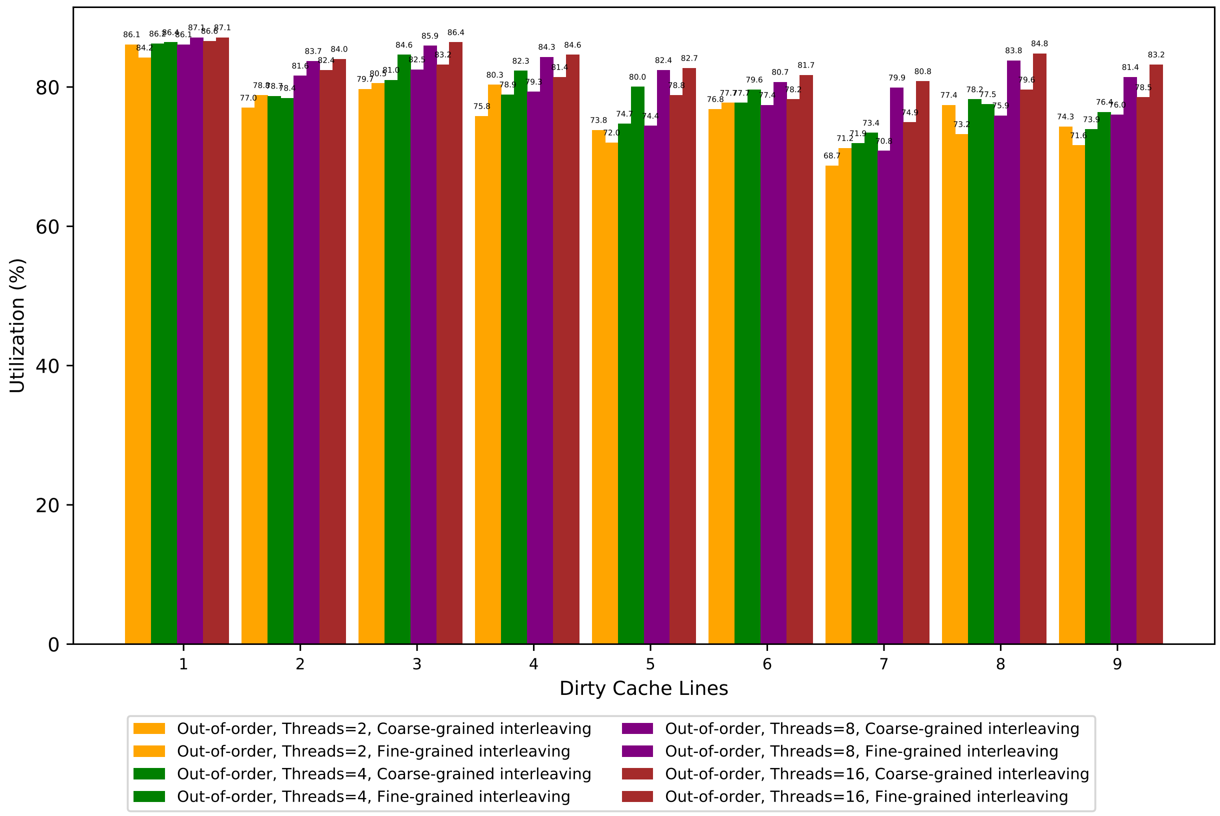


Figure 8.12: Average utilization across different numbers of shared L2 cache dirty cache lines for a 32×1000 times 1000×32 matrix multiplication, comparing fine-grained interleaving and coarse-grained interleaving of micro-threads in Gemmini (8×8).

micro-thread interleaving demonstrates more consistent benefits compared to coarse-grained interleaving. When the micro-thread-count is low (2 or 4 micro-threads), there is no clear choice between the two micro-threading schemes. However, as the thread-count increases, fine-grained interleaving becomes a clear winner, achieving up to 10% higher utilization than coarse-grained interleaving. In-fact, in some cases increasing the thread count does not provide any benefit with coarse-grained interleaving. This is since the fine-grained scheme indeed enables effective load balancing across the micro-threads, with the higher thread-count providing a substrate for load-balancing.

We therefore choose to proceed with fine-grained micro-thread interleaving, and we evaluate our micro-threading implementation by comparing the utilization of the series of experiments on a 32×1000 by 1000×32 matrix multiplication, in order to evaluate its benefit for small matrices which cannot be double-buffered by the controller. We vary the number of micro-threads and compare the utilization results to in-order and out-of-order execution in Gemmini, as illustrated in Figure 8.13, and we observe that for micro-thread counts greater than 4 we see consistent benefits in accelerator utilization when using the out-of-order exe-

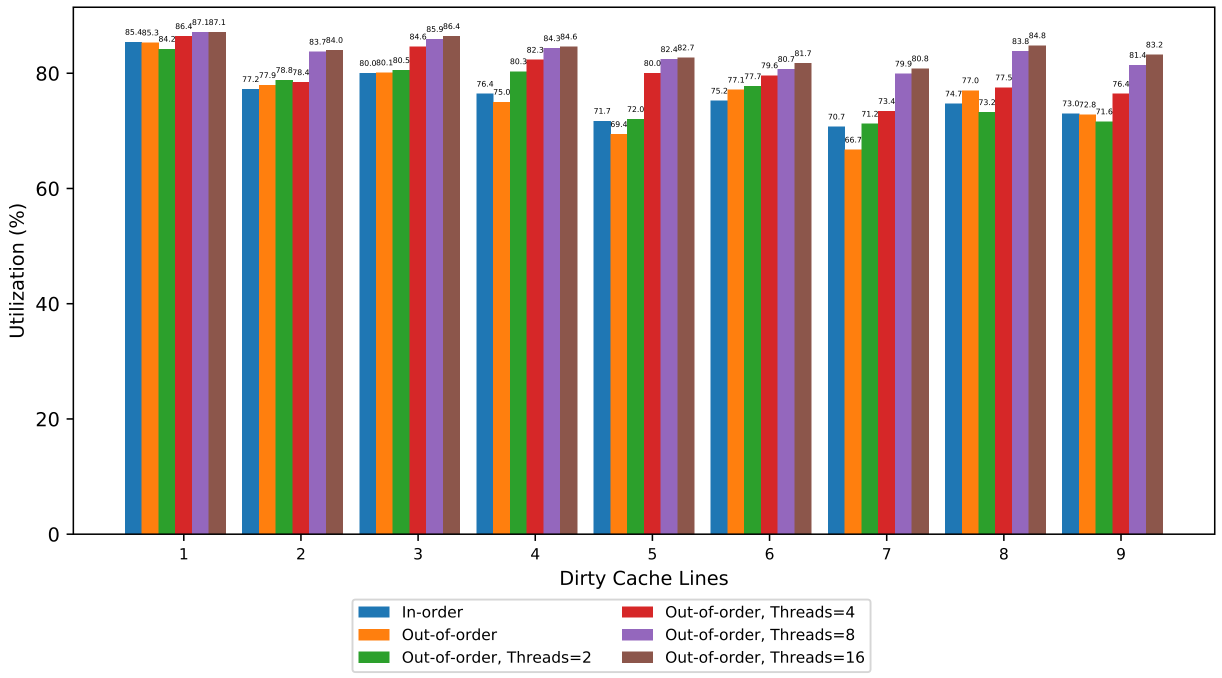


Figure 8.13: Average utilization across different numbers of shared L2 cache dirty cache lines for a 32×1000 times 1000×32 matrix multiplication, comparing fine-grained interleaved commutative micro-threads vs. simple out-of-order and in-order execution in Gemmini (8×8).

cution together with commutative hardware-managed micro-threading. Eight micro-threads appear to provide the optimal increase in utilization, with sixteen threads exhibiting diminishing returns with respect to the number of threads. Using eight micro-threads, we observe up to a 15% improvement in utilization compared to only in-order execution in Gemmini. We conclude that this approach to dynamic scheduling is a worthy customization of the DNN accelerator to better support numerical data analysis workloads which require repeated handling of such small matrices.

We evaluate this technique on a wider spectrum of matrix shapes and sizes, derived from the collection of matrix shapes identified in Figure 8.1. We repeat the series of experiments using dirty cache lines as a method of inducing variable tail latencies while maintaining complete system integrity, this time expanding our range of dirty cache lines to 1-100. Figure 8.14 illustrates the speedup distributions observed for each matrix shapes across the series of experiments using 8 commutative micro-threads, compared to the baseline in-order configuration. In order to evaluate the cost-effectiveness of this methods, we synthesize both configurations using Global Foundries 12nm FinFET process technology. We observe that the total Gemmini area for the baseline in-order configuration is 682,938 $(\mu m)^2$, while the total area for the configuration with our controller improvements is 685,555 $(\mu m)^2$, demon-

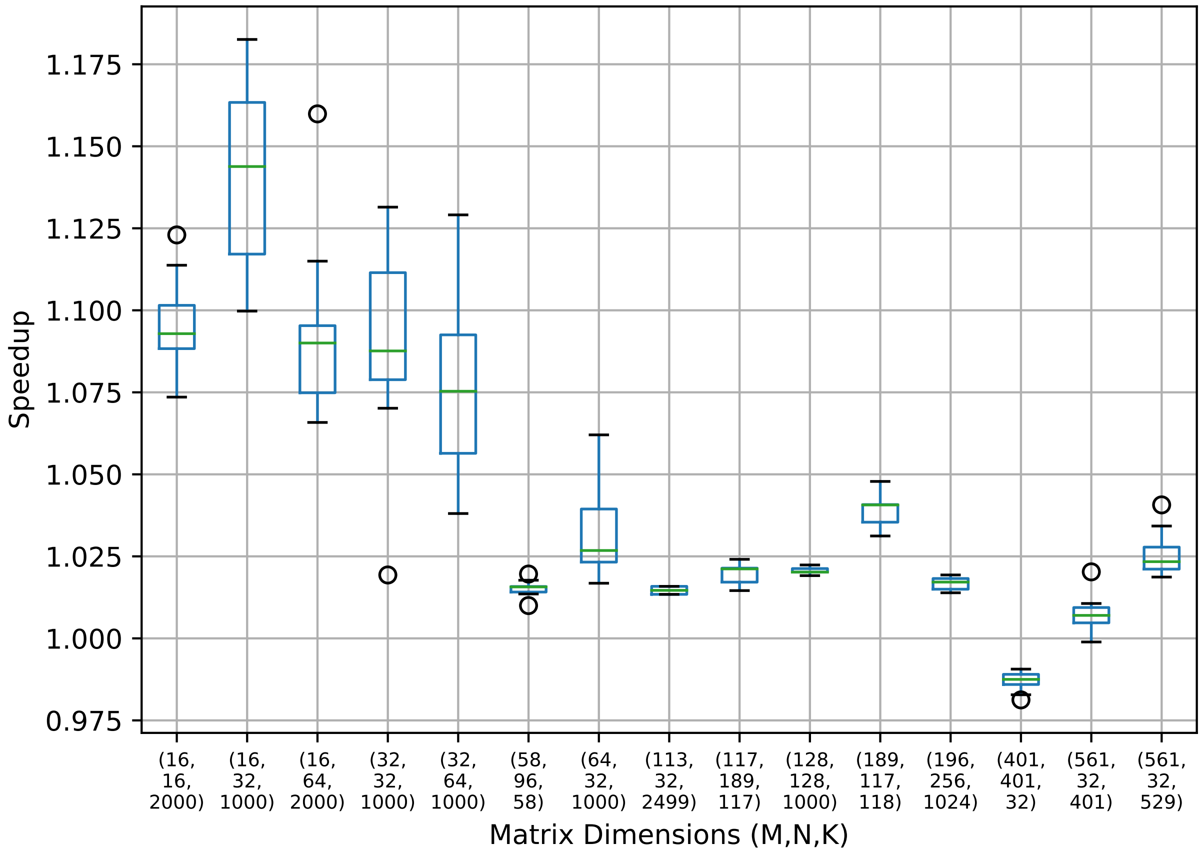


Figure 8.14: Speedup distributions (box plot) of an out-of-order 8×8 Gemmini controller with 8 commutative micro-threads compared to a baseline in-order controller across a collection of matrix shapes with different arithmetic intensities sampled from matrix decomposition workloads.

strating an area addition of only 0.38%. We therefore conclude that compared to the net speedup of these techniques, which is on the order of 1%-25%, their area cost is very low, making these an effective choice for matrix engine controllers.

8.7 Hardware/Software Co-Design for Numerical Data Analysis Summary

In this chapter, we characterized several key differences in the utilization of matrix engines for DNN inference vs. the broader numerical data analysis workloads category. We observed increased importance for processing of matrices with a higher variety of shapes and sizes, including small matrices. This property provides an opportunity for customization and co-design of matrix engines controllers in DNN accelerators, which are generally designed to support high-throughput of processing of large matrices through temporal latency-hiding techniques. We demonstrated how accelerator utilization can be impacted by static scheduling within the matrix engines controller, as well as system-level effects generating variable memory-latency behavior observed by the accelerator at small matrix size regimes. Finally, we propose several customizations to the matrix engine controller within the DNN accelerator to better support both static scheduling and dynamic scheduling of operations within the accelerator. We demonstrate up to a 1.25× improvement in utilization of the Gemmini matrix engine on small matrices through hardware-managed static scheduling, and up to a 1.15× improvement in utilization on small matrices through dynamic scheduling and hardware-managed commutative micro-threading. These improvements require only minor modifications to the current Gemmini micro-architecture, enabling them to be controlled through generator parameters and supporting generator-based hardware/software co-design of SoCs for numerical data analysis workloads.

Chapter 9

Conclusion

9.1 Summary and Contributions

The challenges brought on by the end of the traditional scaling of general-purpose computation have inevitably led to the pursuit of specialized and custom integrated systems-on-a-chip. In recent years, numerical data analysis and machine learning have been driving applications in the proliferation of custom hardware systems for efficient high-performance computing across a diversity of compute platforms. In this thesis, we demonstrated methods of lowering the burden and cost of development of custom SoCs, with a particular interest in numerical data analysis applications. Specifically, this work makes the following contributions:

- Overview and analysis of hardware architectures and design patterns for high performance numerical data analysis applications through data-parallel and spatial acceleration.

- Methodological improvements to generator-based processor design, and particular methodological flows for integrated generator-based SoC development.

- The Chipyard framework as a development and collection of tools guiding, implementing, validating, and automating generator-based SoC design flows.

- Demonstrations of lowering the expertise barrier required for full system SoC design through educational and accessibility features of the Chipyard framework.

- Generator-based hardware/software co-design methods and tools, including generator-based full-system design space exploration using FPGA-accelerated emulation, FPGA-accelerated emulation system profiling tools such as out-of-band performance counters and segmented execution tracing, and generator-based hardware/software performance tuning.

- Software customization methods enabling components of the numerical data analysis software stack to use a DNN accelerator matrix engine as a secondary-use application.
- Contrast the execution of numerical data analysis applications with DNNs on hardware matrix engines, and demonstrate an evaluation of micro-architectural customization of DNN accelerators to enhance their performance on such applications.

9.2 Future Work

Integrated generator-based development methodologies and tools still have a long way to go before they can reach wide adoption in order to meet their goal of lowering NRE for custom SoC design. While the Chipyard integrated SoC development framework has helped tighten the loop between architecture, design, verification and implementation, generated design collateral still transitions between multiple intermediate representations and formats, with each transition causing some loss of information that could have been used for optimization and automation of design decisions.

Additionally, while not academic in its nature, open-source hardware design is a key ingredient on the path towards reducing custom SoC development NRE costs, through sharing of non-competitive design collateral and distribution of verification and low-expertise development resources. However, the structure of the electronic design automation and fabrication tool ecosystems still leads to both organizational and technical hurdles which impact both open-source hardware development and integrated tooling.

Future work into the formalization of more open specifications and intermediate representations across the EDA stack can enable higher levels of abstraction which can in turn lower the expertise barrier required for hardware development. For example, the Chipyard framework currently uses Verilog as the interchange representation between hardware generators and physical design tools. New intermediate representations which can capture physical design abstractions as part of the generator can enable generators which can generalize across a wider range of fabrication technologies. Common intermediate representation could also help with ECOs (engineering change orders, late changes into the netlist), which are currently challenging in a generator-based environment. Similarly, the Chipyard framework currently integrates several third-party designs into the framework, but these often need to be accompanied with pre- and post- processing scripts due to some design assumptions. Higher-level intermediate representation can enable formalized and automated transformation of such third-party blocks, rather than ad-hoc script-based solutions.

The FIRRTL intermediate representation currently provides a substrate for many automated transformations of designs within the Chipyard framework. The uses of these transformations currently include hierarchy changes, debug and visibility wiring, and emulation modeling. Future work would focus on the application of such automated transformations to SoC design for test (DFT) features, such as automatically adding scan chains, bypass networks and redundancy arrays to the design based on the design flow being pursued.

Additional future research directions involve methods and specifications for custom accelerators and functional units. While many specifications with similar goals have been proposed in recent years (OpenCAPI, CXL, etc.), a particularly important direction of research would study interfaces which enable different levels of integration of custom acceleration units with the main CPU. While the RoCC-based accelerators investigated in this work are relatively tightly-coupled to the CPU using virtual memory and a partially shared memory system, coherency and memory management between the accelerators and the main CPU pipeline are still decoupled, involving explicit `fence` instructions by the software. Further research into additional accelerator and specialized functional unit integration methods within generator-based designs would focus on providing variable levels of memory system integration and coherency.

An additional area of future work with respect to the memory systems of custom accelerators would focus on software abstractions rather than hardware interfaces. Within the context of SoCs for numerical data analysis, we observed that the Gemmini matrix engine and the Hwacha vector unit used different memory systems and could communicate only through the shared last level cache, which incurred a communication cost. The RISC-V vector extension, while much constrained compared to its original polymorphic proposals, still provides a prime platform for customization of SoCs for numerical data analysis. The 64-bit extension could be extended with new datatype representations such as matrices and tensors which could provide a substrate of a unified register-file-based memory system across multiple custom accelerators with custom instructions.

With respect to software, future research directions involve investigating more time-efficient optimization methods for accelerator-based high-performance library generation. While the custom BLIS-based BLAS implementation analyzed in this work was sufficiently high performance, and mostly contained to micro-kernels or native calls to the Gemmini SDK, we have noted that there is still room for end-to-end optimization of BLAS and LAPACK kernels for matrix-engine accelerators. Recent work into user-scheduled domain specific languages (such as Halide or TVM) provides avenues for reducing the development costs of custom high-performance software in conjunction with hardware generator-based parameterization. Future work would attempt to generate high-performance software-based accelerator schedules and evaluate their efficiency in relation to native hardware FSM-based control schemes.

Finally, there are additional examples of dense linear algebra supplemental-use applications that can utilize customized matrix engines on edge SoCs. One attractive such example involves optimization, control and robotics applications. Early work on the characterization of this class of applications has identified that they perform many relatively small iterations of matrix multiplication, making them prime candidates to benefit from matrix engine customization tailored to accelerate small matrices. Future work would evaluate this class of applications both in terms of performance requirements for such real-time applications, as well as the numerical properties required to safely support them using the numerical precision of DNN matrix accelerators.

Bibliography

[1] Rashmi Agrawal et al. “The BRISC-V Platform: A Practical Teaching Approach for Computer Architecture”. In: *Proceedings of the Workshop on Computer Architecture Education*. WCAE’19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450368421. DOI: 10.1145/3338698.3338891.

[2] John M. Airey et al. *Display System Having Floating Point Rasterization and Floating Point Framebuffering*. US Patent 6,650,327. Nov. 2003.

[3] Tutu Ajayi et al. “Experiences Using The RISC-V Ecosystem to Design an Accelerator-Centric SoC in TSMC 16nm”. In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2017.

[4] Tutu Ajayi et al. “Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC ’19. Las Vegas, NV, USA: ACM, 2019, 76:1–76:4. ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3326334. URL: http://doi.acm.org/10.1145/3316781.3326334.

[5] Ayaz Akram and Lina Sawalha. “A Survey of Computer Architecture Simulation Techniques and Tools”. In: *IEEE Access* 7 (2019), pp. 78120–78145. DOI: 10.1109/ACCESS.2019.2917698. URL: https://doi.org/10.1109/ACCESS.2019.2917698.

[6] Ayaz Akram and Lina Sawalha. “x86 Computer Architecture Simulators: A Comparative Study”. In: *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 2016, pp. 638–645.

[7] Elad Alon et al. “Open-Source EDA Tools and IP, A View from the Trenches”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC ’19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3323481. URL: https://doi.org/10.1145/3316781.3323481.

[8] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560. URL: https://doi.org/10.1145/1465482.1465560.

[9] Alon Amid. “Nested-Parallelism PageRank on RISC-V Vector Multi-Processors”. MA thesis. University of California, Berkeley, 2019.

[10] Alon Amid, Hasan Genc, and Sophia Yakun Shao. *EE290-2 Hardware for Machine Learning. Lab 3: Tiling and Optimization for Accelerators*. 2020. URL: `http://www-inst.eecs.berkeley.edu/~ee290-2/sp20/assets/labs/lab3.pdf` (visited on 04/09/2020).

[11] Alon Amid and Borivoje Nikolić. “Preventing Babel: Rectifying the Trend of Programming Language Divergence”. In: *The 8th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at SPLASH 2017*. PLATEAU ’17. Vancouver, BC, Canada, 2019.

[12] Alon Amid et al. “Chipyard - An Integrated SoC Research and Implementation Environment: Invited”. In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*. DAC ’20. Virtual Event, USA: IEEE Press, 2020. ISBN: 9781450367257.

[13] Alon Amid et al. “Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs”. In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: `10.1109/MM.2020.2996616`.

[14] Alon Amid et al. “Co-Design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications”. In: *IBM J. Res. Dev.* 63.6 (2019), 6:1–6:14. DOI: `10.1147/JRD.2019.2942284`. URL: `https://doi.org/10.1147/JRD.2019.2942284`.

[15] Alon Amid et al. “Nested-Parallelism PageRank on RISC-V Vector Multi-Processors”. In: *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2019.

[16] Alon Amid et al. “Vertically Integrated Computing Labs Using Open-Source Hardware Generators and Cloud-Hosted FPGAs”. In: *Proceedings of the 2021 International Symposium on Circuits and Systems (ISCAS)*. Daegu, Korea (South), 2021. ISBN: 9781728192017. DOI: `10.1109/ISCAS51556.2021.9401515`.

[17] Ed Anderson et al. “LAPACK: A Portable Linear Algebra Library for High-Performance Computers”. In: *Proceedings Supercomputing ’90, New York, NY, USA, November 12-16, 1990*. Ed. by Joanne L. Martin, Daniel V. Pryor, and Gary Montry. IEEE Computer Society, 1990, pp. 2–11. DOI: `10.1109/SUPERC.1990.129995`. URL: `https://doi.org/10.1109/SUPERC.1990.129995`.

[18] Davide Anguita et al. “A Public Domain Dataset for Human Activity Recognition using Smartphones”. In: *21st European Symposium on Artificial Neural Networks, ESANN 2013, Bruges, Belgium, April 24-26, 2013*. 2013. URL: `http://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2013-84.pdf`.

[19] Krste Asanović. “Vector Extension Proposal v0.2”. 5th RISC-V Workshop. 2016. URL: `https://riscv.org/wp-content/uploads/2016/12/Wed0930-RISC-V-Vectors-Asanovic-UC-Berkeley-SiFive.pdf`.

[20] Krste Asanović. “Vector Microprocessors”. PhD thesis. University of California, Berkeley, 1998.

[21] Krste Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

[22] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016.

[23] Jonathan Bachrach et al. “Chisel: Constructing Hardware in a Scala Embedded Language”. In: *Design Automation Conference (DAC-2012), San Francisco*. June 2012.

[24] Felice Balarin et al., eds. *Hardware-software Co-design of Embedded Systems: The POLIS Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1997. ISBN: 0-7923-9936-6.

[25] Jonathan Balkind et al. “BYOC: A “Bring Your Own Core” Framework for Heterogeneous-ISA Research”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 699–714. ISBN: 9781450371025. DOI: `10.1145/3373376.3378479`. URL: `https://doi.org/10.1145/3373376.3378479`.

[26] Jonathan Balkind et al. “OpenPiton: An Open Source Manycore Research Framework”. In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’16. Atlanta, Georgia, USA: ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5.

[27] Grey Ballard et al. “Minimizing Communication in Numerical Linear Algebra”. In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901.

[28] Grey Ballard et al. “Reconstructing Householder vectors from Tall-Skinny QR”. In: *Journal of Parallel and Distributed Computing* 85 (2015). IPDPS 2014 Selected Papers on Numerical and Combinatorial Algorithms, pp. 3–31. ISSN: 0743-7315. DOI: `10.1016/j.jpdc.2015.06.003`. URL: `https://doi.org/10.1016/j.jpdc.2015.06.003`.

[29] Zvonimir Bandic and Robert Golla. “SweRV Cores Roadmap”. In: *Proceeding of the RISC-V Summit 2019*. 2019.

[30] Christopher Batten et al. “Cache Refill/Access Decoupling for Vector Machines”. In: *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Portland, OR, USA: IEEE Computer Society, 2004, pp. 331–342.

[31] Scott Beamer, Krste Asanović, and David Patterson. *The GAP Benchmark Suite*. 2015. arXiv: `1508.03619 [cs.DC]`.

[32] BK Bershad, Richard P. Draves, and Alessandro Forin. "Using Microbenchmarks to Evaluate System Performance". In: *[1992] Proceedings Third Workshop on Workstation Operating Systems*. Apr. 1992, pp. 148–153. DOI: `10.1109/WWOS.1992.275671`.

[33] *BFLOAT16 – Hardware Numerics Definition*. Tech. rep. 338302-001US. Intel Corp., Nov. 2018.

[34] David Biancolin et al. "Accessible, FPGA Resource-Optimized Simulation of Multi-Clock Systems in FireSim". In: *IEEE Micro* 41.4 (2021), pp. 58–66. ISSN: 0272-1732. DOI: `10.1109/MM.2021.3085537`.

[35] David Biancolin et al. "FASED: FPGA-Accelerated Simulation and Evaluation of DRAM". In: *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*. FPGA '19. Seaside, CA, USA: ACM, 2019. ISBN: 9781450361 378.

[36] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: `10.1145/2024716.2024718`.

[37] Pierre Blanchard et al. "Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores". In: *SIAM J. Sci. Comput.* 42.3 (2020), pp. C124–C141. DOI: `10.1137/19M1289546`. URL: `https://doi.org/10.1137/19M1289546`.

[38] David Blythe. "The Xe GPU Architecture". In: *Hot Chips 32: August 16–18, 2020*. 2020. URL: `https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_GPU_Intel_Xe_David_Blythe.pdf`.

[39] Hugo Brunie et al. "Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*. Ed. by Christine Cuicchi, Irene Qualters, and William T. Kramer. IEEE/ACM, 2020, p. 50. DOI: `10.1109/SC41405.2020.00054`. URL: `https://doi.org/10.1109/SC41405.2020.00054`.

[40] Aydin Buluç and John R. Gilbert. "On The Representation and Multiplication of Hypersparse Matrices". In: *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, 2008, pp. 1–11. DOI: `10.1109/IPDPS.2008.4536313`. URL: `https://doi.org/10.1109/IPDPS.2008.4536313`.

[41] Carl Burch. "Logisim: A Graphical System for Logic Circuit Design and Simulation". In: *J. Educ. Resour. Comput.* 2.1 (Mar. 2002), pp. 5–16. ISSN: 1531-4278. DOI: `10.1145/545197.545199`.

[42] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[43] Luca P. Carloni. “Invited - The Case for Embedded Scalable Platforms”. In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC ’16. Austin, Texas: ACM, 2016, 17:1–17:6. ISBN: 978-1-4503-4236-0. DOI: `10.1145/2897937.2905018`.

[44] Erin Carson and Nicholas J. Higham. “Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions”. In: *SIAM Journal on Scientific Computing* 40.2 (2018), A817–A847. DOI: `10.1137/17M1140819`. eprint: `https://doi.org/10.1137/17M1140819`. URL: `https://doi.org/10.1137/17M1140819`.

[45] Erin Carson, Nicholas J Higham, and Srikara Pranesh. “Three-Precision GMRES-Based Iterative Refinement for Least Squares Problems”. In: (2020).

[46] Matheus Cavalcante et al. “Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multiprecision Floating-Point Support in 22-nm FD-SOI”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.2 (2020), pp. 530–543. DOI: `10.1109/TVLSI.2019.2950087`.

[47] Christopher Celio, David A. Patterson, and Krste Asanović. “The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor”. In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167* (2015).

[48] Eric Chang et al. “BAG2: A Process-Portable Framework for Generator-Based AMS Circuit Design”. In: *2018 IEEE Custom Integrated Circuits Conference, CICC 2018, San Diego, CA, USA, April 8-11, 2018*. IEEE, 2018, pp. 1–8. DOI: `10.1109/CICC.2018.8357061`. URL: `https://doi.org/10.1109/CICC.2018.8357061`.

[49] Tianqi Chen et al. “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 578–594. URL: `https://www.usenix.org/conference/osdi18/presentation/chen`.

[50] Tianshi Chen et al. “DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 269–284. ISBN: 9781450323055. DOI: `10.1145/2541940.2541967`. URL: `https://doi.org/10.1145/2541940.2541967`.

[51] Yu-Hsin Chen et al. “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks”. In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: `10.1109/JSSC.2016.2616357`. URL: `https://doi.org/10.1109/JSSC.2016.2616357`.

[52] Yunji Chen et al. “DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 105–112. ISSN: 0001-0782. DOI: `10.1145/2996864`. URL: `https://doi.org/10.1145/2996864`.

[53] Chipyard. *Chipyard Mailing List*. `https://groups.google.com/g/chipyard`. Accessed: 2021-04-29. 2019.

[54] Jaeyoung Choi et al. "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers". In: *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 1992, pp. 120–127. DOI: `10.1109/FMPC.1992.234898`. URL: `https://doi.ieeecomputersociety.org/10.1109/FMPC.1992.234898`.

[55] Jack Choquette. "NVIDIA's Volta GPU: Programmability and Performance for GPU Computing". In: *Hot Chips 29: August 20–22, 2017*. 2017. URL: `https://old.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.132-Volta-Choquette-NVIDIA-Final3.pdf`.

[56] Jack Choquette and Wishwesh Gandhi. "NVIDIA's A100 GPU: Performance and Innovation for GPU Computing". In: *Hot Chips 32: August 16–18, 2020*. 2020. URL: `https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_GPU_NVIDIA_Choquette_v01.pdf`.

[57] Lawrence T Clark et al. "ASAP7: A 7-nm FinFET Predictive Process Design Kit". In: *Microelectronics Journal* 53 (2016), pp. 105–115.

[58] Phillip Colella. *Defining Software Requirements for Scientific Computing*. 2004.

[59] Jason Cong et al. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 30.4 (2011), pp. 473–491. DOI: `10.1109/TCAD.2011.2110592`. URL: `https://doi.org/10.1109/TCAD.2011.2110592`.

[60] Henry Cook. "Productive Design of Extensible On-Chip Memory Hierarchies". PhD thesis. EECS Department, University of California, Berkeley, May 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-89.html`.

[61] Henry M. Cook, Andrew S. Waterman, and Yunsup Lee. *TileLink Cache Coherence Protocol Implementation*. Tech. rep. 2015.

[62] Henry Cook, Wesley Terpstra, and Yunsup Lee. "Diplomatic Design Patterns: A TileLink Case Study". In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2017.

[63] Efabless Corp. *Caravel: A Template SoC for Google SKY130 Free Shuttles*. `https://github.com/efabless/caravel`. Accessed: 2020-11-30. 2020.

[64] Microsoft Corp. *ONNX Runtime*. 2019. URL: `http://www.onnxruntime.ai/` (visited on 05/21/2020).

[65] Standard Performance Evaluation Corporation. *SPEC CPU 2017*. `https://www.spec.org/cpu2017/`. Accessed: 2020-05-06. 2017.

[66] Ira W. Cotton and Frank S. Greatorex. "Data Structures and Techniques for Remote Computer Graphics". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I.* AFIPS '68 (Fall, part I). San Francisco, California: Association for Computing Machinery, 1968, pp. 533–544. ISBN: 9781450378994. DOI: `10.1145/1476589.1476661`. URL: `https://doi.org/10.1145/1476589.1476661`.

[67] National Research Council et al. *Frontiers in Massive Data Analysis.* National Academies Press, 2013.

[68] Leonardo Dagum and Ramesh Menon. "OpenMP: An Industry-Standard API for Shared-Memory Programming". In: *IEEE Comput. Sci. Eng.* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924. DOI: `10.1109/99.660313`. URL: `https://doi.org/10.1109/99.660313`.

[69] Abdul Dakkak et al. "Accelerating Reduction and Scan Using Tensor Core Units". In: *Proceedings of the ACM International Conference on Supercomputing.* ICS '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 46–57. ISBN: 9781450360791. DOI: `10.1145/3330345.3331057`. URL: `https://doi.org/10.1145/3330345.3331057`.

[70] Debjit Das Sarma and Ganesh Venkataramanan. "Compute and Redundancy Solution for Tesla's Full Self Driving Computer". In: *Hot Chips 31: Stanford Memorial Auditorium, Stanford, California, August 18–20, 2019.* 2019. URL: `https://www.hotchips.org/hc31/HC31_2.3_Tesla_Hotchips_ppt_Final_0817.pdf`.

[71] C. J. Date and E. F. Codd. "The Relational and Network Approaches: Comparison of the Application Programming Interfaces". In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control: Data Models: Data-Structure-Set versus Relational.* SIGFIDET '74. Ann Arbor, Michigan: Association for Computing Machinery, 1975, pp. 83–113. ISBN: 9781450374187. DOI: `10.1145/800297.811532`. URL: `https://doi.org/10.1145/800297.811532`.

[72] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51.1 (2008), pp. 107–113. DOI: `10.1145/1327452.1327492`. URL: `http://doi.acm.org/10.1145/1327452.1327492`.

[73] Robert H. Dennard et al. "Design of Ion-Implanted MOSFET's With Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: `10.1109/JSSC.1974.1050511`.

[74] Keith Diefendorff et al. "AltiVec Extension to PowerPC Accelerates Media Processing". In: *IEEE Micro* 20.2 (2000), pp. 85–95. DOI: `10.1109/40.848475`. URL: `https://doi.org/10.1109/40.848475`.

[75] Jens Domke et al. *Matrix Engines for High Performance Computing:A Paragon of Performance or Grasping at Straws?* 2020. arXiv: `2010.14373 [cs.DC]`.

[76] Jack J. Dongarra et al. "A Set of Level 3 Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* 16.1 (Mar. 1990), pp. 1–17. ISSN: 0098-3500. DOI: `10.1145/77626.79170`. URL: `https://doi.org/10.1145/77626.79170`.

[77] Jack J. Dongarra et al. "An Extended Set of FORTRAN Basic Linear Algebra Subprograms". In: *ACM Trans. Math. Softw.* 14.1 (1988), pp. 1–17. DOI: `10.1145/42288.42291`. URL: `https://doi.org/10.1145/42288.42291`.

[78] Jack J. Dongarra et al. "PLASMA: Parallel Linear Algebra Software for Multicore Using OpenMP". In: *ACM Trans. Math. Softw.* 45.2 (2019), 16:1–16:35. DOI: `10.1145/3264491`. URL: `https://doi.org/10.1145/3264491`.

[79] Jack J. Dongarra et al. "The Singular Value Decomposition: Anatomy of Optimizing an Algorithm for Extreme Scale". In: *SIAM Rev.* 60.4 (2018), pp. 808–865. DOI: `10.1137/17M1117732`. URL: `https://doi.org/10.1137/17M1117732`.

[80] Alexander Dörflinger et al. "A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations". In: *Proceedings of the 18th ACM International Conference on Computing Frontiers*. CF '21. Virtual Event, Italy: Association for Computing Machinery, 2021, pp. 12–20. ISBN: 9781450384049. DOI: `10.1145/3457388.3458657`. URL: `https://doi.org/10.1145/3457388.3458657`.

[81] Joel S. Emer and Douglas W. Clark. "A Characterization of Processor Performance in the Vax-11/780". In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. ISCA '84. New York, NY, USA: Association for Computing Machinery, 1984, pp. 301–310. ISBN: 0818605383. DOI: `10.1145/800015.808199`.

[82] Erik Engheim. *The Secret Apple M1 Coprocessor*. `https://medium.com/swlh/apples-m1-secret-coprocessor-6599492fc1e1`. Accessed: 2021-07-08. 2021.

[83] Berkeley College of Engineering. *Berkeley engineering students pull off novel chip design in a single semester*. `https://engineering.berkeley.edu/news/2021/06/berkeley-engineering-students-design-novel-chip-in-semester-long-course/`. Accessed: 2021-06-20. 2021.

[84] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 365–376. ISBN: 9781450304726. DOI: `10.1145/2000064.2000108`. URL: `https://doi.org/10.1145/2000064.2000108`.

[85] Roger Espasa and Mateo Valero. "Decoupled Vector Architectures". In: *Proceedings of the Second International Symposium on High-Performance Computer Architecture*. HPCA '96. San Jose, CA, USA: IEEE Computer Society, 1996, pp. 281–290.

[86] Farzad Farshchi, Qijing Huang, and Heechul Yun. "Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim". In: *Proccedings of the 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications, at HPCA 2019*. Washington D.C., USA, 2019.

[87] Massimiliano Fasi et al. "Numerical Behavior of NVIDIA Tensor Cores". In: *PeerJ Comput. Sci.* 7 (2021), e330. DOI: 10.7717/peerj-cs.330. URL: https://doi.org/10.7717/peerj-cs.330.

[88] Richard P. Feynman. "There's Plenty of Room at the Bottom". In: *Feynman and Computation: Exploring the Limits of Computers*. USA: Perseus Books, 1999, pp. 63–76. ISBN: 0738200573.

[89] Robert W. Floyd. "Permuting Information in Idealized Two-Level Storage". In: *Complexity of Computer Computations*. Springer, 1972, pp. 105–109.

[90] Linux Foundation. *Open Neural Network Exchange (ONNX)*. 2019. URL: https://onnx.ai/ (visited on 05/21/2020).

[91] Gianluca Frison et al. "BLASFEO: Basic Linear Algebra Subroutines for Embedded Optimization". In: *ACM Trans. Math. Softw.* 44.4 (July 2018). ISSN: 0098-3500. DOI: 10.1145/3210754. URL: https://doi.org/10.1145/3210754.

[92] Adi Fuchs and David Wentzlaff. "The Accelerator Wall: Limits of Chip Specialization". In: *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 1–14. DOI: 10.1109/HPCA.2019.00023. URL: https://doi.org/10.1109/HPCA.2019.00023.

[93] Mark Gallagher et al. "Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. Ed. by Iris Bahar et al. ACM, 2019, pp. 469–484. DOI: 10.1145/3297858.3304037. URL: https://doi.org/10.1145/3297858.3304037.

[94] Michael Gautschi et al. "Near-Threshold RISC-V Core with DSP Extensions for Scalable IoT Endpoint Devices". In: *IEEE Trans. Very Large Scale Integr. Syst.* 25.10 (2017), pp. 2700–2713. DOI: 10.1109/TVLSI.2017.2654506. URL: https://doi.org/10.1109/TVLSI.2017.2654506.

[95] Hasan Genc et al. "Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures". In: *arXiv:1911.09925 [cs.DC]* (2019). arXiv: 1911.09925 [cs.DC].

[96] Hasan Genc et al. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *Proceedings of the 58th ACM/EDAC/IEEE Design Automation Conference*. DAC '21. San Francisco, CA, USA: IEEE Press, 2021.

[97] Amir Gholami et al. "SqueezeNext: Hardware-Aware Neural Network Design". In: *arXiv preprint arXiv:1803.10615* (2018).

[98] Massimo Giordano et al. "CHIMERA: A 0.92 TOPS, 2.2 TOPS/W Edge AI Accelerator with 2 MByte On-Chip Foundry Resistive RAM for Efficient Training and Inference". In: *IEEE Symposium on VLSI Circuits, VLSI Circuits 2021, Kyoto, Japan, June 13-19, 2021.* IEEE, 2021.

[99] Gene H. Golub and Charles F. Van Loan. *Matrix computations.* Vol. 3. JHU press, 2013.

[100] Abraham Gonzalez et al. "A 16mm^2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET". In: *47th IEEE European Solid State Circuits Conference, ESSCIRC 2021, Grenoble, France, September 6-9, 2021.* Grenoble, France: IEEE, 2021.

[101] Google. *SkyWater Open Source PDK.* `https://github.com/google/skywater-pdk`. Accessed: 2021-07-08. 2020.

[102] Kazushige Goto and Robert A. van de Geijn. "Anatomy of High-Performance Matrix Multiplication". In: *ACM Trans. Math. Softw.* 34.3 (May 2008). ISSN: 0098-3500. DOI: `10.1145/1356052.1356053`. URL: `https://doi.org/10.1145/1356052.1356053`.

[103] Kazushige Goto and Robert Van De Geijn. "High-Performance Implementation of the Level-3 BLAS". In: *ACM Trans. Math. Softw.* 35.1 (July 2008). ISSN: 0098-3500. DOI: `10.1145/1377603.1377607`. URL: `https://doi.org/10.1145/1377603.1377607`.

[104] Samuel Greengard. "Will RISC-V Revolutionize Computing?" In: *Commun. ACM* 63.5 (2020), pp. 30–32. DOI: `10.1145/3386377`. URL: `https://doi.org/10.1145/3386377`.

[105] Michael Gschwind. "Workload Acceleration with The IBM POWER Vector-Scalar Architecture". In: *IBM J. Res. Dev.* 60.2-3 (2016). URL: `https://doi.org/10.1147/JRD.2016.2527418`.

[106] Gagan Gupta et al. "Kickstarting Semiconductor Innovation with Open Source Hardware". In: *Computer* 50.6 (2017), pp. 50–59.

[107] Suyog Gupta and Berkin Akin. *Accelerator-aware Neural Network Design using AutoML.* 2020. arXiv: `2003.02838 [eess.SP]`.

[108] John Gustafson and Isaac Yonemoto. "Beating Floating Point at its Own Game: Posit Arithmetic". In: *Supercomputing Frontiers and Innovations* 4.2 (2017). ISSN: 2313-8734. URL: `https://www.superfri.org/superfri/article/view/137`.

[109] Linley Gwennap. "Application-Specific Accelerators Extend Moore's Law". In: *Linley Fall Processor Conference 2020, October 20-29, 2020.* 2020. URL: `http://www.linleygroup.com/cms_builder/uploads/FPC20_October20_Session_Slides`.

[110] Azzam Haidar et al. "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed Up Mixed-Precision Iterative Refinement Solvers". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE. 2018, pp. 603–613.

[111] Azzam Haidar et al. “Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers”. In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems.* ScalA ’17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351256. DOI: 10.1145/3148226.3148237. URL: https://doi.org/10.1145/3148226.3148237.

[112] Azzam Haidar et al. “MAGMA Embedded: Towards a Dense Linear Algebra Library for Energy Efficient Extreme Computing”. In: *2015 IEEE High Performance Extreme Computing Conference, HPEC 2015, Waltham, MA, USA, September 15-17, 2015.* IEEE, 2015, pp. 1–6. DOI: 10.1109/HPEC.2015.7322444. URL: https://doi.org/10.1109/HPEC.2015.7322444.

[113] James Hamilton. *AWS Inferentia Machine Learning Processor.* https://perspectives.mvdirona.com/2018/11/aws-inferentia-machine-learning-processor/. Accessed: 2021-05-08. 2018.

[114] Mark J. Harris et al. “Physically-Based Visual Simulation on Graphics Hardware”. In: *ACM SIGGRAPH 2005 Courses.* SIGGRAPH ’05. Los Angeles, California: Association for Computing Machinery, 2005, 221–es. ISBN: 9781450378338. DOI: 10.1145/1198555.1198791. URL: https://doi.org/10.1145/1198555.1198791.

[115] Sarah L. Harris et al. “MIPSfpga: Using a Commercial MIPS Soft-Core in Computer Architecture Education”. In: *IET Circuits, Devices & Systems* 11.4 (July 2017), pp. 283–291. ISSN: 1751-858X. DOI: 10.1049/iet-cds.2016.0383. URL: https://doi.org/10.1049/iet-cds.2016.0383.

[116] Raza Hasan and Salman Mahmood. “Survey and Evaluation of Simulators Suitable for Teaching for Computer Architecture and Organization Supporting Undergraduate Students at Sir Syed University of Engineering Technology”. In: *Proceedings of 2012 UKACC International Conference on Control.* 2012, pp. 1043–1045.

[117] John Hauser. *Berkeley HardFloat.* https://github.com/ucb-bar/berkeley-hardfloat. Accessed: 2021-05-21. 2019.

[118] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* June 2016.

[119] Alexander Heinecke et al. “LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016.* Ed. by John West and Cherri M. Pancake. IEEE Computer Society, 2016, pp. 981–991. DOI: 10.1109/SC.2016.83. URL: https://doi.org/10.1109/SC.2016.83.

[120] Liao Heng. "A Scalable Unified Architecture for Neural Network Computing from Nano-Level to High Performance Computing". In: *Hot Chips 31: Stanford Memorial Auditorium, Stanford, California, August 18–20, 2019*. 2019. URL: `https://www.hotchips.org/hc31/HC31_1.11_Huawei.Davinci.HengLiao_v4.0.pdf`.

[121] John L. Hennessy and David A. Patterson. "A New Golden Age for Computer Architecture". In: *Commun. ACM* 62.2 (Jan. 2019), pp. 48–60. ISSN: 0001-0782. DOI: `10.1145/3282307`.

[122] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[123] Nicholas J. Higham, Srikara Pranesh, and Mawussi Zounon. "Squeezing a Matrix into Half Precision, with an Application to Solving Linear Systems". In: *SIAM Journal on Scientific Computing* 41.4 (2019), A2536–A2551. DOI: `10.1137/18M1229511`. eprint: `https://doi.org/10.1137/18M1229511`. URL: `https://doi.org/10.1137/18M1229511`.

[124] Nicholas Higham and Srikara Pranesh. "Exploiting Lower Precision Arithmetic in Solving Symmetric Positive Definite Linear Systems and Least Squares Problems". In: (2019).

[125] R. G. Hintz and Tate D. P. "Control Data STAR-100 Processor Design". In: *COMPCON*. IEEE. 1972, pp. 1–4.

[126] Mark Horowitz. "Computing's Energy Problem (And What We Can Do About It)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14.

[127] Andrew G. Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: `1704.04861 [cs.CV]`.

[128] Qijing Huang et al. "CoSA: Scheduling by Constrained Optimization for Spatial Accelerators". In: *Proceedings of the 48th Annual International Symposium on Computer Architecture*. ISCA '21. Valencia, Spain: Association for Computing Machinery, 2021.

[129] Andrey Ignatov et al. "AI Benchmark: All About Deep Learning on Smartphones in 2019". In: *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops 2019, Seoul, Korea (South), October 27-28, 2019*. IEEE, 2019, pp. 3617–3635. DOI: `10.1109/ICCVW.2019.00447`. URL: `https://doi.org/10.1109/ICCVW.2019.00447`.

[130] Gabriel Ilharco et al. *High Performance Natural Language Processing (EMNLP 2020)*. `http://gabrielilharco.com/publications/EMNLP_2020_Tutorial__High_Performance_NLP.pdf`. Accessed: 2021-05-24. 2020.

[131] Apple Inc. *Apple Accelerate*. `https://developer.apple.com/accelerate/`. Accessed: 2021-01-25. 2020.

[132] Apple Inc. *Apple unleashes M1*. `https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/`. Accessed: 2021-01-25. 2020.

[133] Cadence Design Systems Inc. *Cadence Cloud Portfolio*. `https://www.cadence.com/en_US/home/solutions/cadence-cloud.html`. Accessed: 2021-05-08. 2021.

[134] Cadence Design Systems Inc. *Palladium Cloud*. `https://www.cadence.com/en_US/home/solutions/cadence-cloud/palladium-cloud.html`. Accessed: 2021-05-08. 2021.

[135] Microchip Technology Inc. *PolarFire SoC*. `https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga`. Accessed: 2021-07-08. 2021.

[136] SiFive Inc. *fpga-shells*. `https://github.com/sifive/fpga-shells`. Accessed: 2020-11-30. 2019.

[137] Synopsys Inc. *Synopsys Cloud Solutions*. `https://www.synopsys.com/cloud.html`. Accessed: 2021-05-08. 2021.

[138] Synopsys Inc. *ZeBu Cloud*. `https://www.synopsys.com/verification/solutions/zebu-cloud-solution.html`. Accessed: 2021-05-08. 2021.

[139] Western Digital Inc. *RISC-V and Open Source Hardware Address New Compute Requirements*. `https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/tech-brief/tech-brief-western-digital-risc-v.pdf`. Accessed: 2021-07-08. 2021.

[140] *Intel Architecture Instruction Set Extensions and Future Features Programming Reference*. Tech. rep. 319433-040. Intel Corp., June 2020.

[141] RISC-V International. *RISC-V Educational Materials*. `https://riscv.org/educational-materials/`. Accessed: 2020-05-06. 2020.

[142] Dror Irony, Sivan Toledo, and Alexander Tiskin. "Communication Lower Bounds for Distributed-Memory Matrix Multiplication". In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026.

[143] Adam Izraelevitz et al. "Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. Irvine, California: IEEE Press, 2017, pp. 209–216.

[144] Aakash Jani. "Apple Ships Its First PC Processor". In: *Microprocessor Report* 35.1 (2021), pp. 1–2.

[145] Geonhwa Jeong et al. "RASA: Efficient Register-Aware Systolic Array Matrix Engine for CPU". In: *Proceedings of the 58th Annual Design Automation Conference, DAC 2021*. ACM, 2021.

[146] Yangqing Jia et al. "Caffe: Convolutional Architecture for Fast Feature Embedding". In: *Proceedings of the 22Nd ACM International Conference on Multimedia.* MM '14. Orlando, Florida, USA: ACM, 2014, pp. 675–678. ISBN: 978-1-4503-3063-3. DOI: `10.1145/2647868.2654889`. URL: `http://doi.acm.org/10.1145/2647868.2654889`.

[147] Hong Jia-Wei and H. T. Kung. "I/O Complexity: The Red-Blue Pebble Game". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing.* STOC '81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pp. 326–333. ISBN: 9781450373920. DOI: `10.1145/800076.802486`.

[148] Norman P. Jouppi et al. "A Domain-Specific Supercomputer for Training Deep Neural Networks". In: *Commun. ACM* 63.7 (June 2020), pp. 67–78. ISSN: 0001-0782. DOI: `10.1145/3360307`. URL: `https://doi.org/10.1145/3360307`.

[149] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: `10.1145/3079856.3080246`. URL: `http://doi.acm.org/10.1145/3079856.3080246`.

[150] Dhiraj Kalamkar et al. *A Study of BFLOAT16 for Deep Learning Training.* 2019. arXiv: `1905.12322 [cs.LG]`.

[151] Sagar Karandikar et al. "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 715–731. ISBN: 9781450371025.

[152] Sagar Karandikar et al. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018.* Ed. by Murali Annavaram, Timothy Mark Pinkston, and Babak Falsafi. IEEE Computer Society, 2018, pp. 29–42. DOI: `10.1109/ISCA.2018.00014`. URL: `https://doi.org/10.1109/ISCA.2018.00014`.

[153] Ben Keller et al. "A RISC-V Processor SoC with Integrated Power Management at Submicrosecond Timescales in 28 nm FD-SOI". In: *IEEE Journal of Solid-State Circuits* 52.7 (2017), pp. 1863–1875. DOI: `10.1109/JSSC.2017.2690859`.

[154] Jeremy Kepner et al. "Mathematical Foundations of The GraphBLAS". In: *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016.* IEEE, 2016, pp. 1–9. DOI: `10.1109/HPEC.2016.7761646`. URL: `https://doi.org/10.1109/HPEC.2016.7761646`.

[155] Brian W Kernighan and Shen Lin. "An Efficient Heuristic Procedure for Partitioning Graphs". In: *The Bell system technical journal* 49.2 (1970), pp. 291–307.

[156] Moein Khazraee et al. “Moonwalk: NRE Optimization in ASIC Clouds”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: Association for Computing Machinery, 2017, pp. 511–526. ISBN: 9781450344654. DOI: `10.1145/3037697.3037749`. URL: `https://doi.org/10.1145/3037697.3037749`.

[157] Donggyu Kim et al. “DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of Cycles”. In: *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018*. IEEE Computer Society, 2018, pp. 76–80. DOI: `10.1109/FPL.2018.00021`. URL: `https://doi.org/10.1109/FPL.2018.00021`.

[158] Olof Kindgren. “Invited Paper: A Scalable Approach to IP Management with FuseSoC”. In: *Workshop on Open Source Design Automation*. 2019.

[159] Deborah L. Knox. “Integrating Design and Simulation into a Computer Architecture Course”. In: *Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education*. ITiCSE ’97. Uppsala, Sweden: Association for Computing Machinery, 1997, pp. 42–44. ISBN: 0897919238. DOI: `10.1145/268819.268834`.

[160] H. T. Kung and Charles E Leiserson. “Systolic arrays (for VLSI)”. In: *Sparse Matrix Proceedings 1978*. Vol. 1. Society for Industrial and Applied Mathematics. 1979, pp. 256–282.

[161] Hsiang-Tsung Kung. “Why Systolic Architectures?” In: *IEEE computer* 15.1 (1982), pp. 37–46.

[162] Kiseok Kwon et al. “Co-Design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications”. In: *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*. ACM, 2018, 148:1–148:6. DOI: `10.1145/3195970.3199849`. URL: `https://doi.org/10.1145/3195970.3199849`.

[163] E. Scott Larsen and David McAllister. “Fast Matrix Multiplies Using Graphics Hardware”. In: *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. SC ’01. Denver, Colorado: Association for Computing Machinery, 2001, p. 55. ISBN: 158113293X. DOI: `10.1145/582034.582089`. URL: `https://doi.org/10.1145/582034.582089`.

[164] Luciano Lavagno, Louis Scheffer, and Grant Martin. *EDA for IC Implementation, Circuit Design, and Process Technology*. CRC press, 2006.

[165] Charles L. Lawson et al. “Basic Linear Algebra Subprograms for Fortran Usage”. In: *ACM Trans. Math. Softw.* 5.3 (1979), pp. 308–323. ISSN: 0098-3500. DOI: `10.1145/355841.355847`. URL: `https://doi.org/10.1145/355841.355847`.

[166] Jong-Hyuk Lee et al. “Pipelined CPU Design with FPGA in Teaching Computer Architecture”. In: *IEEE Transactions on Education* 55.3 (2012), pp. 341–348. DOI: `10.1109/TE.2011.2175227`. URL: `https://doi.org/10.1109/TE.2011.2175227`.

[167] Yunsup Lee. "Decoupled Vector-Fetch Architecture with a Scalarizing Compiler". PhD thesis. EECS Department, University of California, Berkeley, May 2016. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-117.html`.

[168] Yunsup Lee and Andrew Waterman. "Managing Chip Design Complexity in the Domain-Specific SoC Era". In: *IEEE Symposium on VLSI Circuits, VLSI Circuits 2020, Honolulu, HI, USA, June 16-19, 2020*. IEEE, 2020, pp. 1–2. DOI: `10.1109/VLSICircuits18222.2020.9162812`.

[169] Yunsup Lee et al. "A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators". In: *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*. 2014, pp. 199–202. DOI: `10.1109/ESSCIRC.2014.6942056`.

[170] Yunsup Lee et al. "An Agile Approach to Building RISC-V Microprocessors". In: *IEEE Micro* 36.2 (2016), pp. 8–20. ISSN: 0272-1732. DOI: `10.1109/MM.2016.11`. URL: `https://doi.org/10.1109/MM.2016.11`.

[171] Yunsup Lee et al. "Exploring the Design Space of SPMD Divergence Management on Data-Parallel Architectures". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Cambridge, UK: IEEE Computer Society, 2014, pp. 101–113.

[172] Yunsup Lee et al. "Hwacha Preliminary Evaluation Results, Version 3.8." In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-264* (2015).

[173] Yunsup Lee et al. "The Hwacha Microarchitecture Manual, Version 3.8." In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-263* (2015).

[174] Yunsup Lee et al. "The Hwacha Vector-Fetch Architecture Manual, Version 3.8". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-262* (2015).

[175] Charles E. Leiserson et al. "There's Plenty of Room At The Top: What Will Drive Computer Performance After Moore's law?" In: *Science* 368.6495 (2020). ISSN: 0036-8075. DOI: `10.1126/science.aam9744`. eprint: `https://science.sciencemag.org/content/368/6495/eaam9744.full.pdf`. URL: `https://science.sciencemag.org/content/368/6495/eaam9744`.

[176] Jure Leskovec et al. "Stanford Network Analysis Project". In: *ht tp://snap. stanford. edu* (2010).

[177] Xiaoye S. Li et al. "Design, Implementation and Testing of Extended and Mixed Precision BLAS". In: *ACM Trans. Math. Softw.* 28.2 (2002), pp. 152–205. DOI: `10.1145/567806.567808`. URL: `https://doi.org/10.1145/567806.567808`.

[178] *Libgloss, a free board support package (BSP)*. URL: `https://www.gnu.org/software/dejagnu/manual/Libgloss.html` (visited on 04/09/2020).

[179] Chris Lomont. “Introduction to Intel Advanced Vector Extensions”. In: *Intel white paper* 23 (2011).

[180] Ryan Lund. “Design and Application of a Co-Simulation Framework for Chisel”. MA thesis. EECS Department, University of California, Berkeley, May 2021. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-133.html`.

[181] Yirong Lv et al. “CounterMiner: Mining Big Performance Data from Hardware Counters”. In: *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 613–626. DOI: `10.1109/MICRO.2018.00056`. URL: `https://doi.org/10.1109/MICRO.2018.00056`.

[182] Martin Maas, Krste Asanović, and John Kubiatowicz. “A Hardware Accelerator for Tracing Garbage Collection”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. Los Angeles, California: IEEE Press, 2018, pp. 138–151. ISBN: 9781538659847. DOI: `10.1109/ISCA.2018.00022`. URL: `https://doi.org/10.1109/ISCA.2018.00022`.

[183] Albert Magyar et al. “Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes”. In: *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*. Ed. by David Z. Pan. ACM, 2019, pp. 1–8. DOI: `10.1109/ICCAD45719.2019.8942087`. URL: `https://doi.org/10.1109/ICCAD45719.2019.8942087`.

[184] Phillip R. Malone. *GOOGLE LLC. vs. ORACLE AMERICA INC. BRIEF AMICI CURIAE OF 83 COMPUTER SCIENTISTS IN SUPPORT OF PETITIONER*. `https://www.supremecourt.gov/DocketPDF/18/18-956/128391/20200113145027664_18-956%20Google%20v%20Oracle%20Computer%20Scientists%20Merits%20Amicus%20FOR%20FILING.pdf`. Number 18-956. Accessed: 2021-01-25. 2020.

[185] Svetlin A. Manavski and Giorgio Valle. “CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment”. In: *BMC bioinformatics* 9.2 (2008), pp. 1–9.

[186] Paolo Mantovani et al. “Agile SoC Development with Open ESP : Invited Paper”. In: *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*. IEEE, 2020, 96:1–96:9. DOI: `10.1145/3400302.3415753`. URL: `https://doi.org/10.1145/3400302.3415753`.

[187] Howard Mao. “Hardware Acceleration for Memory to Memory Copies”. MA thesis. EECS Department, University of California, Berkeley, Jan. 2017. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-2.html`.

[188] Howard Mao, Randy H. Katz, and Krste Asanović. *Hardware Acceleration for Memory to Memory Copies*. Tech. rep. Technical Report UCB/EECS-2017-2, EECS Department, University of California, Berkeley, 2017.

[189] Eitan Medina. “Habana Labs Approach to Scaling AI Training”. In: *Hot Chips 31: Stanford Memorial Auditorium, Stanford, California, August 18–20, 2019.* 2019. URL: `https://www.hotchips.org/hc31/HC31_1.14_HabanaLabs.Eitan_Medina.v9.pdf`.

[190] Mahim Mishra et al. “Tartan: Evaluating Spatial Computation for Whole Program Execution”. In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS XII. San Jose, California, USA: Association for Computing Machinery, 2006, pp. 163–174. ISBN: 1595934510. DOI: `10.1145/1168857.1168878`. URL: `https://doi.org/10.1145/1168857.1168878`.

[191] Gaurav Mitra et al. “Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms”. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013.* IEEE, 2013, pp. 1107–1116. DOI: `10.1109/IPDPSW.2013.207`. URL: `https://doi.org/10.1109/IPDPSW.2013.207`.

[192] Kenneth Moreland and Edward Angel. “The FFT on a GPU”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware.* HWWS ’03. San Diego, California: Eurographics Association, 2003, pp. 112–119. ISBN: 1581137397.

[193] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI Applications”. In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018.* Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 561–577. URL: `https://www.usenix.org/conference/osdi18/presentation/nishihara`.

[194] Tipp Moseley, Neil Vachharajani, and William Jalby. “Hardware Performance Monitoring for the Rest of Us: A Position and Survey”. In: *Network and Parallel Computing - 8th IFIP International Conference, NPC 2011, Changsha, China, October 21-23, 2011. Proceedings.* Ed. by Erik R. Altman and Weisong Shi. Vol. 6985. Lecture Notes in Computer Science. Springer, 2011, pp. 293–312. DOI: `10.1007/978-3-642-24403-2\_23`. URL: `https://doi.org/10.1007/978-3-642-24403-2%5C_23`.

[195] Dorit Naishlos. “Autovectorization in GCC”. In: *Proceedings of the 2004 GCC Developers Summit.* 2004, pp. 105–118.

[196] John A. Nestor. “Teaching Computer Organization with HDLs: An Incremental Approach”. In: *2005 International Conference on Microelectronics Systems Education, MSE 2005, Anaheim, CA, USA, June 12-13, 2005.* IEEE Computer Society, 2005, pp. 77–78. DOI: `10.1109/MSE.2005.51`. URL: `https://doi.org/10.1109/MSE.2005.51`.

[197] Rishiyur S. Nikhil. “Bluespec System Verilog: Efficient, Correct RTL From High Level Specifications”. In: *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), 23-25 June 2004, San Diego, California, USA, Proceedings*. IEEE Computer Society, 2004, pp. 69–70. DOI: `10.1109/MEMCOD.2004.1459818`. URL: `https://doi.org/10.1109/MEMCOD.2004.1459818`.

[198] Borivoje Nikolic, Elad Alon, and Krste Asanovic. “Generating the Next Wave of Custom Silicon”. In: *44th IEEE European Solid State Circuits Conference, ESSCIRC 2018, Dresden, Germany, September 3-6, 2018*. IEEE, 2018, pp. 6–11. DOI: `10.1109/ESSCIRC.2018.8494310`. URL: `https://doi.org/10.1109/ESSCIRC.2018.8494310`.

[199] Bosko Nikolić et al. “A Survey and Evaluation of Simulators Suitable for Teaching Courses in Computer Architecture and Organization”. In: *IEEE Transactions on Education* 52.4 (Nov. 2009), pp. 449–458. ISSN: 0018-9359. DOI: `10.1109/TE.2008.930097`.

[200] Nod.ai. *Comparing Apple’s M1 matmul performance – AMX2 vs NEON*. `https://nod.ai/comparing-apple-m1-with-amx2-m1-with-neon/`. Accessed: 2021-07-08. 2021.

[201] Thomas Norrie et al. “The Design Process for Google’s Training Chips: TPUv2 and TPUv3”. In: *IEEE Micro* 41.2 (2021), pp. 56–63. DOI: `10.1109/MM.2021.3058217`. URL: `https://doi.org/10.1109/MM.2021.3058217`.

[202] Tony Nowatzki et al. “Architectural Simulators Considered Harmful”. In: *IEEE Micro* 35.6 (2015), pp. 4–12. DOI: `10.1109/MM.2015.74`. URL: `https://doi.org/10.1109/MM.2015.74`.

[203] Dorit Nuzman, Ira Rosen, and Ayal Zaks. “Auto-Vectorization of Interleaved Data for SIMD”. In: *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*. Ed. by Michael I. Schwartzbach and Thomas Ball. ACM, 2006, pp. 132–143. DOI: `10.1145/1133981.1133997`. URL: `https://doi.org/10.1145/1133981.1133997`.

[204] NVIDIA. *White paper: NVIDIA TESLA V100 GPU ARCHITECTURE*. Aug. 2017. URL: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`.

[205] Lawrence Page et al. *The PageRank Citation Ranking: Bringing Order to The Web.* Tech. rep. Stanford InfoLab, 1999.

[206] David Parkins. “The World’s Most Valuable Resource Is No Longer Oil, But Data”. In: *The economist* 6 (2017).

[207] David Lorge Parnas. “On the Criteria To Be Used in Decomposing Systems into Modules”. In: *Commun. ACM* 15.12 (1972), pp. 1053–1058. ISSN: 0001-0782. DOI: `10.1145/361598.361623`. URL: `https://doi.org/10.1145/361598.361623`.

[208] Atish Patra. *Successful KVM RISC-V bring up on FPGA (Rocket core with H extension).* https://lore.kernel.org/linux-riscv/CAOnJCULY7PTcyZ4dOMiC6YBW3_ZdN3tVC4jg_wAJdJs_YeCnMQ@mail.gmail.com/. Accessed: 2021-05-08. 2021.

[209] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12 (2011), pp. 2825–2830. URL: http://dl.acm.org/citation.cfm?id=2078195.

[210] Mark S. Peercy et al. "Interactive Multi-Pass Programmable Shading". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '00. USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 425–432. ISBN: 1581132085. DOI: 10.1145/344779.344976. URL: https://doi.org/10.1145/344779.344976.

[211] Nathan Pemberton and Alon Amid. "FireMarshal: Making HW/SW Co-Design Reproducible and Reliable". In: *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* ISPASS '21. Stony Brook, NY, USA: IEEE, 2021, pp. 299–309. ISBN: 9781728186436. DOI: 10.1109/ISPASS51385.2021.00052.

[212] Daniel Petrisko et al. "BlackParrot: An Agile Open-Source RISC-V Multicore for Accelerator SoCs". In: *IEEE Micro* 40.4 (2020), pp. 93–102. DOI: 10.1109/MM.2020.2996145. URL: https://doi.org/10.1109/MM.2020.2996145.

[213] Raghu Prabhakar et al. "Plasticine: A Reconfigurable Architecture For Parallel Patterns". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 389–402. ISBN: 9781450348928. DOI: 10.1145/3079856.3080256. URL: https://doi.org/10.1145/3079856.3080256.

[214] Pranav Prakash. "End-to-end Model Inference and Training on Gemmini". MA thesis. EECS Department, University of California, Berkeley, May 2021. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-37.html.

[215] P. W. C. Prasad et al. "Using Simulators for Teaching Computer Organization and Architecture". In: *Computer Applications in Engineering Education* 24.2 (2016), pp. 215–224. DOI: 10.1002/cae.21699. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cae.21699.

[216] Guofeng Qin et al. "Design and Performance Analysis on Static and Dynamic Pipelined CPU in Course Experiment of Computer Architecture". In: *2018 13th International Conference on Computer Science Education (ICCSE).* 2018, pp. 1–6.

[217] Vijay Janapa Reddi et al. "MLPerf Inference Benchmark". In: *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020.* IEEE, 2020, pp. 446–459. DOI: 10.1109/ISCA45697.2020.00045. URL: https://doi.org/10.1109/ISCA45697.2020.00045.

[218] Daniel Richins et al. “Amdahl’s Law in Big Data Analytics: Alive and Kicking in TPCx-BB (BigBench)”. In: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018.* IEEE Computer Society, 2018, pp. 630–642. DOI: 10.1109/HPCA.2018.00060. URL: https://doi.org/10.1109/HPCA.2018.00060.

[219] Matthew Rocklin. “Dask: Parallel Computation with Blocked Algorithms and Task Scheduling”. In: *Proceedings of the 14th python in science conference.* Citeseer. 2015, pp. 130–136.

[220] Bita Rouhani et al. “Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point”. In: *NeurIPS 2020.* ACM. Nov. 2020. URL: https://www.microsoft.com/en-us/research/publication/pushing-the-limits-of-narrow-precision-inferencing-at-cloud-scale-with-microsoft-floating-point/.

[221] Cindy Rubio-González et al. “Precimonious: Tuning Assistant for Floating-Point Precision”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC’13, Denver, CO, USA - November 17 - 21, 2013.* Ed. by William Gropp and Satoshi Matsuoka. ACM, 2013, 27:1–27:12. DOI: 10.1145/2503210.2503296. URL: https://doi.org/10.1145/2503210.2503296.

[222] Martin Rumpf and Robert Strzodka. “Nonlinear Diffusion in Graphics Hardware”. In: *3rd Joint Eurographics - IEEE TCVG Symposium on Visualization, VisSym 2001, Ascona, Switzerland, May 28-30, 2001.* Ed. by David S. Ebert, Jean M. Favre, and Ronald Peikert. EGVISSYM’01. Eurographics Association, 2001, pp. 75–84. DOI: 10.1007/978-3-7091-6215-6_9. URL: https://doi.org/10.1007/978-3-7091-6215-6%5C_9.

[223] Richard M. Russell. “The CRAY-1 Computer System”. In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. ISSN: 0001-0782. DOI: 10.1145/359327.359336. URL: https://doi.org/10.1145/359327.359336.

[224] Bruno Sá, José Martins, and Sandro Pinto. “A First Look at RISC-V Virtualization from an Embedded Systems Perspective”. In: *CoRR* abs/2103.14951 (2021). arXiv: 2103.14951. URL: https://arxiv.org/abs/2103.14951.

[225] Alberto L. Sangiovanni-Vincentelli. “The Tides of EDA”. In: *IEEE Design Test of Computers* 20.6 (2003), pp. 59–75. DOI: 10.1109/MDT.2003.1246165. URL: https://doi.org/10.1109/MDT.2003.1246165.

[226] Pasquale Davide Schiavone et al. “Slow and Steady Wins The Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications”. In: *27th International Symposium on Power and Timing Modeling, Optimization and Simulation, PATMOS 2017, Thessaloniki, Greece, September 25-27, 2017.* IEEE, 2017, pp. 1–8. DOI: 10.1109/PATMOS.2017.8106976. URL: https://doi.org/10.1109/PATMOS.2017.8106976.

[227] Colin Schmidt. “Extending Temporal-Vector Microarchitectures for Two-Dimensional Computations”. PhD thesis. EECS Department, University of California, Berkeley, 2021.

[228] Colin Schmidt and Albert Ou. “Hwacha: A Data-Parallel RISC-V Extension and Implementation”. In: *Proceedings of the Inaugural RISC-V Summit*. RISC-V Foundation. 2018.

[229] Colin Schmidt et al. “4.3 An Eight-Core 1.44GHz RISC-V Vector Machine in 16nm FinFET”. In: *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. Vol. 64. 2021, pp. 58–60. DOI: 10.1109/ISSCC42613.2021.9365789.

[230] Colin Schmidt et al. “Programmable Fine-Grained Power Management and System Analysis of RISC-V Vector Processors in 28nm FD-SOI”. In: *IEEE Solid-State Circuits Letters* 3 (2020), pp. 210–213. ISSN: 2573-9603. DOI: 10.1109/LSSC.2020.3010295.

[231] Robert Schreiber and Charles Van Loan. “A Storage-Efficient *WY* Representation for Products of Householder Transformations”. In: *SIAM Journal on Scientific and Statistical Computing* 10.1 (1989), pp. 53–57. DOI: 10.1137/0910005. eprint: https://doi.org/10.1137/0910005. URL: https://doi.org/10.1137/0910005.

[232] Naresh Sehgal, John M. Acken, and Sohum Sohoni. “Is the EDA Industry Ready for Cloud Computing?” In: *IETE Technical Review* 33.4 (2016), pp. 345–356. DOI: 10.1080/02564602.2015.1099056.

[233] Amazon Web Services. *AWS Inferentia*. https://aws.amazon.com/machine-learning/inferentia/. Accessed: 2021-05-08. 2021.

[234] Ofer Shacham et al. “Rethinking Digital Design: Why Design Must Change”. In: *IEEE Micro* 30.6 (2010), pp. 9–24. DOI: 10.1109/MM.2010.81. URL: https://doi.org/10.1109/MM.2010.81.

[235] Mohamed Shalan and Tim Edwards. “Building OpenLANE: A 130nm Openroad-Based Tapeout-Proven Flow”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD ’20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415735. URL: https://doi.org/10.1145/3400302.3415735.

[236] Yakun Sophia Shao. “Design and Modeling of Specialized Architectures”. PhD thesis. Harvard University, 2016.

[237] Yakun Sophia Shao et al. “Co-designing Accelerators and SoC Interfaces Using GEM5-Aladdin”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–12.

[238] Dongjoo Shin et al. “DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks”. In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. 2017, pp. 240–241. DOI: 10.1109/ISSCC.2017.7870350.

[239] Gil Shomron and Uri C. Weiser. “Non-Blocking Simultaneous Multithreading: Embracing the Resiliency of Deep Neural Networks”. In: *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 2020, pp. 256–269. DOI: `10.1109/MICRO50266.2020.00032`. URL: `https://doi.org/10.1109/MICRO50266.2020.00032`.

[240] Siemens. *Catapult High-Level Synthesis and Verification*. `https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis`. Accessed: 2021-07-08. 2021.

[241] Frans Sijstermans. “The NVIDIA Deep Learning Accelerator”. In: *Hot Chips 30: The Flint Center for the Performing Arts, Cupertino, California, August 19–21, 2018*. 2018. URL: `https://www.hotchips.org/hc30/2conf/2.08_NVidia_DLA_Nvidia_DLA_HotChips_10Aug18.pdf`.

[242] James E. Smith. “Decoupled Access/Execute Computer Architectures”. In: *Proceedings of the 9th Annual Symposium on Computer Architecture*. ISCA ’82. Austin, TX, USA: IEEE Computer Society Press, 1982, pp. 112–119.

[243] Marc Snir et al. *MPI–the Complete Reference: the MPI core*. Vol. 1. MIT press, 1998.

[244] Sandro Neves Soares and Flávio Rech Wagner. “Design Space Exploration of Embedded Processors in Computer Architecture Education using T D-Bench”. In: *Proceedings. Frontiers in Education. 36th Annual Conference*. 2006, pp. 19–24.

[245] Sandro Neves Soares and Flávio Rech Wagner. “T D-Bench—Innovative Combined Support for Education and Research in Computer Architecture and Embedded Systems”. In: *IEEE Transactions on Education* 54.4 (2011), pp. 675–682.

[246] Jinook Song et al. “An 11.5TOPS/W 1024-MAC Butterfly Structure Dual-Core Sparsity-Aware Neural Processing Unit in 8nm Flagship Mobile SoC”. In: *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 2019, pp. 130–132. DOI: `10.1109/ISSCC.2019.8662476`.

[247] William Starke and Brian Thompto. “IBM’s POWER10 Processor”. In: *Hot Chips 32: August 16–18, 2020*. 2020. URL: `https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_Server_Processors_IBM_Starke_POWER10_v33.pdf`.

[248] Nigel Stephens et al. “The ARM Scalable Vector Extension”. In: *IEEE Micro* 37.2 (2017), pp. 26–39. DOI: `10.1109/MM.2017.35`. URL: `https://doi.org/10.1109/MM.2017.35`.

[249] Narayanan Sundaram et al. “GraphMat: High Performance Graph Analytics Made Productive”. In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1214–1225. ISSN: 2150-8097. DOI: `10.14778/2809974.2809983`. URL: `https://doi.org/10.14778/2809974.2809983`.

[250] Synopsys. *Lynx Design System - A Comprehensive Design Automation Environment*. `https://www.synopsys.com/implementation-and-signoff/lynx-design-system.html`. Accessed: 2021-01-10. 2021.

[251] Thierry Tambe et al. “AdaptivFloat: A Floating-point based Data Type for Resilient Deep Learning Inference”. In: *CoRR* abs/1909.13271 (2019). arXiv: 1909.13271. URL: http://arxiv.org/abs/1909.13271.

[252] *TASKING Lapack Performance Libraries For Infineon AURIX MCUs*. Tech. rep. Altium LLC (TASKING Brand), July 2018.

[253] Sanket Tavarageri et al. “PolyDL: Polyhedral Optimizations for Creation of High-Performance DL Primitives”. In: *ACM Trans. Archit. Code Optim.* 18.1 (Jan. 2021). ISSN: 1544-3566. DOI: 10.1145/3433103. URL: https://doi.org/10.1145/3433103.

[254] Michael Bedford Taylor. “Basejump STL: Systemverilog Needs a Standard Template Library for Hardware Design”. In: *Proceedings of the 55th Annual Design Automation Conference*. DAC ’18. San Francisco, California: ACM, 2018, 73:1–73:6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3199848. URL: http://doi.acm.org/10.1145/3195970.3199848.

[255] Michael Bedford Taylor. “Your Agile Open Source HW Stinks (Because It Is Not a System)”. In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–6.

[256] SkyWater Technology. *First Google-Sponsored MPW Shuttle Launched at SkyWater with 40 Open Source Community Submitted Designs*. https://www.skywatertechnology.com/press-releases/first-google-sponsored-mpw-shuttle-launched-at-skywater-with-40-open-source-community-submitted-designs/. Accessed: 2021-07-08. 2021.

[257] Neil Thompson and Svenja Spanuth. “The Decline of Computers As A General Purpose Technology: Why Deep Learning And The End of Moore’s Law Are Fragmenting Computing”. In: *Available at SSRN 3287769* (2018).

[258] Sivan Toledo. “Locality of Reference in LU Decomposition with Partial Pivoting”. In: *SIAM Journal on Matrix Analysis and Applications* 18.4 (1997), pp. 1065–1081.

[259] Stanimire Tomov et al. “MAGMA Users’ Guide”. In: *ICL, UTK (November 2009)* (2011).

[260] Konrad Trifunovic et al. “Polyhedral-Model Guided Loop-Nest Auto-Vectorization”. In: *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*. IEEE Computer Society, 2009, pp. 327–337. DOI: 10.1109/PACT.2009.18. URL: https://doi.org/10.1109/PACT.2009.18.

[261] *University of California, Berkeley uses AWS Educate and Amazon FPGA Instances in Undergraduate Computer Architecture Course*. https://aws.amazon.com/blogs/publicsector/university-of-berkeley-uses-aws-educate-for-amazon-fpga-accelerator-development-and-deployment-in-the-cloud/. Accessed: 2020-05-06.

[262] Amin Vahdat. *The past, present and future of custom compute at Google.* `https://cloud.google.com/blog/topics/systems/the-past-present-and-future-of-custom-compute-at-google`. Accessed: 2021-04-29. 2021.

[263] Field G. Van Zee and Tyler M. Smith. "Implementing High-Performance Complex Matrix Multiplication via the 3m and 4m Methods". In: *ACM Trans. Math. Softw.* 44.1 (July 2017). ISSN: 0098-3500. DOI: `10.1145/3086466`. URL: `https://doi.org/10.1145/3086466`.

[264] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Transactions on Mathematical Software* 41.3 (June 2015), 14:1–14:33. URL: `http://doi.acm.org/10.1145/2764454`.

[265] Field G. Van Zee et al. "The BLIS Framework: Experiments in Portability". In: *ACM Transactions on Mathematical Software* 42.2 (June 2016), 12:1–12:19. URL: `http://doi.acm.org/10.1145/2755561`.

[266] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon et al. 2017, pp. 5998–6008. URL: `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

[267] Pauli Virtanen et al. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17 (2020), pp. 261–272. DOI: `10.1038/s41592-019-0686-2`.

[268] Stefan Wallentowitz. "RISC-V in Practical Education of Computer Architecture". In: *Proceeding of the RISC-V Summit 2019.* 2019.

[269] Edward Wang et al. "A Methodology for Reusable Physical Design". In: *Twenty First International Symposium on Quality Electronic Design, 2020. Proceedings.* Mar. 2020.

[270] Shibo Wang and Pankaj Kanwar. *BFloat16: The Secret to High Performance on Cloud TPUs.* `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`. Accessed: 2021-05-24. 2019.

[271] Yu Wang, Gu-Yeon Wei, and David Brooks. "A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms". In: *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020.* Ed. by Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze. mlsys.org, 2020. URL: `https://proceedings.mlsys.org/book/289.pdf`.

[272] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213.* Tech. rep. RISC-V Foundation, Dec. 2019.

[273] Jian Weng et al. "UNIT: Unifying Tensorized Instruction Compilation". In: CGO '08 (2021).

[274] R. Clinton Whaley and Jack J. Dongarra. "Automatically Tuned Linear Algebra Software". In: *Proceedings of the ACM/IEEE Conference on Supercomputing, SC 1998, November 7-13, 1998, Orlando, FL, USA*. IEEE Computer Society, 1998, p. 38. DOI: 10.1109/SC.1998.10004. URL: https://doi.org/10.1109/SC.1998.10004.

[275] Paul N. Whatmough et al. "CHIPKIT: An Agile, Reusable Open-Source Framework for Rapid Test Chip Development". In: *IEEE Micro* 40.4 (2020), pp. 32–40. DOI: 10.1109/MM.2020.2995809. URL: https://doi.org/10.1109/MM.2020.2995809.

[276] Nathan Whitehead. *Whitepaper: Linear Algebra Package for Qualcomm Snapdragon Math Libraries*. Tech. rep. Qualcomm Technologies, Inc., Apr. 2017.

[277] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[278] C. Wolf. *Yosys Open SYnthesis Suite - write_firrtl - write design to a FIRRTL file*. 2018. URL: http://www.clifford.at/yosys/cmd_write_firrtl.html (visited on 04/14/2020).

[279] John Charles Wright et al. "A Dual-Core RISC-V Vector Processor with On-Chip Fine-Grain Power Management in 28-nm FD-SOI". In: *IEEE Trans. Very Large Scale Integr. Syst.* 28.12 (2020), pp. 2721–2725. DOI: 10.1109/TVLSI.2020.3030243. URL: https://doi.org/10.1109/TVLSI.2020.3030243.

[280] Bichen Wu. "Efficient Deep Neural Networks". PhD thesis. EECS Department, University of California, Berkeley, Aug. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-120.html.

[281] Carole-Jean Wu et al. "Machine Learning at Facebook: Understanding Inference at the Edge". In: *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 331–344. DOI: 10.1109/HPCA.2019.00048. URL: https://doi.org/10.1109/HPCA.2019.00048.

[282] Kun Yang et al. "High Performance Monte Carlo Simulation of Ising Model on TPU Clusters". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356149. URL: https://doi.org/10.1145/3295500.3356149.

[283] Marco Zagha and Guy E. Blelloch. "Radix Sort for Vector Multiprocessors". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pp. 712–721. ISBN: 0-89791-459-7. DOI: 10.1145/125826.126164. URL: http://doi.acm.org/10.1145/125826.126164.

[284] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". In: *Commun. ACM* 59.11 (2016), pp. 56–65. DOI: 10.1145/2934664. URL: http://doi.acm.org/10.1145/2934664.

[285] Florian Zaruba and Luca Benini. "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2926114. URL: https://doi.org/10.1109/TVLSI.2019.2926114.

[286] Shaoshuai Zhang, Elaheh Baharlouei, and Panruo Wu. "High Accuracy Matrix Computations on Neural Engines: A Study of QR Factorization and Its Applications". In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '20. Stockholm, Sweden: Association for Computing Machinery, 2020, pp. 17–28. ISBN: 9781450370523. DOI: 10.1145/3369583.3392685. URL: https://doi.org/10.1145/3369583.3392685.

[287] Shaoshuai Zhang, Ruchi Shah, and Panruo Wu. "TensorSVM: Accelerating Kernel Machines with Tensor Engine". In: *Proceedings of the 34th ACM International Conference on Supercomputing*. ICS '20. Barcelona, Spain: Association for Computing Machinery, 2020. ISBN: 9781450379830. DOI: 10.1145/3392717.3392770. URL: https://doi.org/10.1145/3392717.3392770.

[288] Jerry Zhao et al. "Sonicboom: The 3rd Generation Berkeley Out-Of-Order Machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2020.

[289] Haishan Zhu et al. "Kelp: QoS for Accelerated Machine Learning Systems". In: *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 172–184. DOI: 10.1109/HPCA.2019.00036. URL: https://doi.org/10.1109/HPCA.2019.00036.

[290] Brian Zimmer et al. "A RISC-V Vector Processor with Simultaneous-Switching Switched-Capacitor DC–DC Converters in 28 nm FDSOI". In: *IEEE Journal of Solid-State Circuits* 51.4 (2016), pp. 930–942. DOI: 10.1109/JSSC.2016.2519386.

Appendix A

Observations on the RISC-V Vector Standards Process

The RISC-V standard vector extension proposal has been under development by the RISC-V Foundation Vector Extension Task Group since 2016 [19]. Its protracted gestation has involved at least two complete rewrites of the specification, during which a radically "cleaner" design (in terms of ISA intuitiveness, extensibility, and generalizability), initially attractive from a research perspective, transitioned into a more outwardly complex architecture to adapt to implementation realities.

Initial proposals for the RISC-V standard vector extension were closely influenced by past vector processor designs at UC Berkeley, including the Hwacha vector accelerator, and especially the concept of reconfigurable vector register files. An important feature is the ability to aggregate vector registers to extend the hardware vector length. These previous vector processor designs considered primarily temporal vector implementations, with small-scale multi-lane spatial implementations as experimental enhancements composing together multiple temporal lane implementations. While the vector ISA which were developed together with these vector processor implementations were intended to support flexibility across both temporal and spatial implementations, their primary evaluation (including in the case of Hwacha) used single-lane or dual-lane implementations.

The first widely available version of the RISC-V vector specification, v0.4, was designed to be fully polymorphic. Statically encoded parameters such as datatypes and data layout are instead converted into dynamic state in the form of control registers (named `vtypes`), effectively generalizing the concept of vector length indirection into a more reconfigurable vector unit. Polymorphic instructions occupy minimal encoding space since each fundamental operation requires only a single instruction, in contrast to non-polymorphic instruction encodings which require unique opcodes for each combination of source and destination data types. Furthermore, polymorphic designs are extremely amenable to flexible customization and extension, since data properties are not encoded in instructions, but rather in control state, which can be more easily customized and extended for individual implementations.

The flexibility and aestheticism associated with a fully polymorphic design come at a cost. First, a polymorphic design is not formally well-defined. This ambiguity may be beneficial under relaxed constraints, but challenging under compliance constraints. In order for a polymorphic-design to be well-defined for compliance purposes, an elaborate and complex cross-product of state-spaces must be defined, which impacts the complexity of underlying implementations. This was further complicated by the desire for mixed data type computations, meaning each instruction could have up to 4 different data types as inputs and outputs. Some of these combinations were illegal or undesired and so this complexity would have to be represented in hardware. Second, the additional control registers required to store configuration information increase the state of the vector ISA well beyond the vector register file and vector length register. At instruction decode, an implementation would have to read the configuration state of all registers involved and determine whether this operation is legal, as well as which functional unit would be required for the computation. In addition, the minimum required configuration state is extremely prohibitive for the simplest designs implementing the vector extension.

This configuration state overhead was a major concern, and therefore a major re-write of the vector extension proposal focused on two primary simplifications: reducing the amount of configuration state, and restricting the space of register layouts. This was done by converting some of the configuration state into encoded instructions, and making the `vtypes` state optional. Version 0.5 of the vector extension proposal proposed a series of pre-set configurations for "fast configuration" which could be encoded in a single control register. By providing a subset of "likely" configurations, this version of the specification allowed implementations to support only a pruned configuration state space that would be most frequently used. While complex implementations could opt to provide additional support for the complete configuration state space using the optional `vtypes` control registers, simple designs could limit the optimization of their design to the much smaller pre-set configurations with significantly smaller control state. This revision kept much of the flexibility of the polymorphic model such that complex and custom implementations would have the option for extensibility, while optimizing for the "most likely" configurations in order to simplify the design and enable low-overhead support for simple implementations.

The next major revision of the RISC-V vector extension proposal (v0.7) was primarily a result of a point of contention regarding compatibility with software and hardware implementations that are currently optimized for packed-SIMD vectorization. Notably, the polymorphic proposal did not expose spatial register width, and abstracted wide vector computations in terms of vector elements rather than bits. Specifically, mixed-precision computation under several different implementation design points became a topic of concern, as the vector element abstraction was preventing mixed-precision computation using packed-SIMD techniques which bit-pack wide physical registers with multiple lower-precision vectors. While this implementation style may not be characteristic of deeply-pipelined vector processors for high performance computing, as a rule of thumb RISC-V traditionally tries to enable/optimize for the simplest possible design, and therefore there was a high desire to accommodate this requirement. Hence, this version of the vector extension proposal abandoned the layout-agnostic element abstractions, and exposed the bit layout of the vector registers.

This change had rippling implications on all parts of the vector extension proposal. Once the bit layout has been exposed, support for variable length vectors and reconfigurability of the vector register file had to be exposed to software as well. As such, long vector register groups were now composed of groups of smaller physical vector registers, associated with specific register identifiers. A single instruction can operate on a full vector register group, as if it were a single vector register. While the sizes of the physical vector registers could change across implementations (setting the `VLEN` machine parameter), software portability would be guaranteed only for cases in which the value of the vector register grouping parameter (`LMUL`) times the standard element width (`SEW`) divided by the physical register size (`VLEN`) remained the same ratio (`LMUL*SEW / VLEN`). Version 0.7 was the last fundamental re-design of the vector extension proposal. Subsequent versions added additional refinements and enhancements based an implementation and software development experimentation.

Table A.1 summarizes the main differences between the major three generations of the RISC-V standard vector extension proposals. The RISC-V vector extension proposal was not used for the work in this dissertation due to its volatility throughout its development process, but demonstrates a small sample of the wide ranging concerns required to address the spectrum of implementation and application design points for data-parallel processors, and the need to support customizability within this space.

Table A.1: RISC-V vector extension proposal draft version comparison table.

	v0.1-v0.4	v0.5	v0.6-1.0
Reconfiguration granularity (individual vector widths)	Each vector register has different element width	Groups of vector registers have different element widths (limited to 4x range)	All vector registers have the same width
Scalar handling	Scalar “shape” configuration for each vector register	First element of each vector register can be treated as a scalar. Scalar bit in instruction encoding.	Use the standard x, f scalar registers, with additional vector-scalar operation encodings.
Minimum architectural state	vtypes (5 bits * 32 reg.) + vstart (XLEN bits) + vxsat (1 bit) + vxrm (2 bits) + vfflags (5 bits) + vfrm (3 bits) + vl (XLEN bits) + VS in Mstatus (2 bits)	vcfg (12 bits) + SEW (3 bits) + vstart (XLEN bits) + vxsat (1 bits) + vxrm (2 bits) + vfflags (5 bits) + vfrm (3 bits) + vl (XLEN bits) + VS in Mstatus(2 bits)	LMUL (2 bits) + SEW (3 bits) + vtype (6 bits) + vstart (XLEN bits) + vxsat (1 bit) + vxrm (2 bits) + vfflags (5 bits) + vfrm (3 bits) + vl (XLEN bits) + VS in Mstatus (2 bits)

Mixed-precision computation	Supported using polymorphic instructions based on register type (`vtype`)	Supported using polymorphic instructions based on register type (`vtype`) + fixed-type mixed precision instructions	Explicit widening and/or narrowing instructions (limited to 2x widening/narrowing in each direction)
Instruction polymorphism (without vector unit reconfig.)	Full	Partial	Minimal
Predication	Two explicit predicate mask registers (`vp0`, `vp1`). Predication can be disabled in the vtype register. 1-bit `vp` field in the instruction encoding (enabling the possibility of negated/inverse mask interpretation)	LSBs of `v1` elements are interpreted as a predicate bit mask. 2-bit `vm` field in the instruction encoding (enabling the possibility of negated/inverse mask interpretation)	LSBs of `v0` elements are interpreted as a predicate bit mask. 1-bit `vm` filed in the instruction encoding (no possibility of negated/inverse mask interpretation)
Reductions	Reductions supported via `vslide` instructions	Explicit reduction instructions (`vredop`). Write results to vector reg.	Explicit reduction instructions (`vredop`). Write results to vector reg.
Handling illegal configs	Illegal instruction trap upon bad configuration instruction.	WARL (machine had to check that configs were legal). Take trap on first non-config vector instruction.	`vill` CSR bit is set when there is an illegal configuration. Execution attempt will result in an illegal instruction exception
Encoding Space	Less than 1 major opcode space	1 major opcode space	Between 1-2 major opcode spaces
Extensibility (non-standard types/operations)	• 32-bit encoding: – Shape extensions – Representation extensions – Polymorphic instruction overloading. • 64-bit extension with additional encoding.	• 32-bit encoding: – Shape extensions – Representation extensions – Limited polymorphic instruction overloads. • 64-bit extension with additional encoding.	64-bit extension with additional encoding
Portability (correctness, exposed machine parameters)	Minimal VLEN of 4 elements	N/A	VLEN, ELEN, SLEN exposed. SEW / LMUL ratio must be maintained to ensure software portability.

Portability (platform-level constraints)	Vector unit supports only the platform-level extensions that are defined on the scalar ISA version (e.g., floating point).	Vector unit supports only the platform-level extensions that are defined on the scalar ISA version (e.g., floating point).	Vector unit can support platform-level extensions that are not defined on the scalar ISA version (e.g., floating point). Broader selection of platform-level extensions.
Tail zeroing	Undisturbed tail	Zero tail	Undisturbed tail
Reconfiguration zeroing	Upon the configuration of an individual register, zero all registers with a higher index identifier (i.e., if v5 was reconfigured, v5-v31 would be zeroed, while v0-v4 would be undisturbed)	Zero all vector registers upon any reconfiguration	Undisturbed vector register state upon any reconfiguration