UNIVERSITY OF CALIFORNIA SAN DIEGO

Polymorphic Compilation for Cross-Domain Acceleration

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science
(Computer Engineering)

by

Sean Kinzer

2023

University of California San Diego

2023

DEDICATION

To my family-Suzanne, Scott, Connor, and Marshall and especially to my wife, Kaitlin.

EPIGRAPH

I'm smart enough to know that I'm dumb.

—Richard P. Feynman

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Prof. Hadi Esmaeilzadeh for his guidance throughout my PhD. He provided opportunities for me that I would not have had anywhere else. He believed in me when I often lacked belief in myself, and transformed me into a better research as well as a better person.

I would also like to thank my committee members, Prof. Sorin Lerner, Prof. Dean Tullsen, and Prof. Yatish Turakhia for their feedback on my work, as well as valuable insights.

Throughout my PhD, I had the pleasure of working with a number of great mentors who aided in my journey. I am extremely grateful for their participation in my research as well as general guidance. I was fortunate to work with Dr. Alric Althoff at Leidos; Dr. Xialong Li, Dr. Janarbek Matai, and Dr. Liang Zhang from Qualcomm; Prof. Andrew Kahng from UCSD; Prof. Sachin Sapatnekar from University of Minnesota.

My peers in the Alternative Computing Technologies Lab (ACTLab), Dr. Byung Hoon Ahn, Dr. Soroush Ghodrati, Rohan Mahapatra, Hanyang Xu, Shu-Ting Wang, Parsa Asadi, Chris Priebe, Joon Kyung Kim, Lavanya Karthikeyan, Brahmendra Reddy Yatham, Edwin Mascarenhas, Fatemehsadat Mireshghallah and previous graduates Dr. Ahmed Taha Elthakeb, Prannoy Pilligundla, Dr. Divya Mahajan who was pivotal in her guidance, Prof. Jongse Park, and Dr. Hardik Sharma have been invaluable to me during my PhD. For most of them, doing their PhD in the United States required moving from their home country, learning a new language, writing research papers in that language, and in some cases rendered them unable to visit their families in person for the duration of graduate school. To complicate their situation even further, COVID hit and left them concerned for their loved ones abroad who may or may not have had access to quality healthcare. I mention this because I observed their perseverance and grit through all of it, as they trudged forward in an extremely competitive PhD program which presents it's own set of challenges, and did so without a single complaint or excuse. I can confidently say I would not have made it through my PhD without them as an inspiration and role models to look up to.

I was fortunate to have gained a number of friends and mentors outside of research through my graduate school experience as well, who were incredibly important in providing words of advice and moral support through the journey. In particular, I must thank Prof. Mary Boyle, who was pivotal in encouraging me to continue my PhD at a period of time when I was thinking of stopping; I never forgot her kindness and words of advice. In addition, I must thank my best friends Chris Lamb and Trevor Elwell, who spent many hours with me studying for graduate courses, and provided much needed humor and friendship throughout my PhD.

I also must thank my mother, Suzanne, my father Scott, my brothers Marshall and Connor for their encouragement and words advice throughout my PhD. I am forever grateful to have such a loving family, who would have supported whatever pursuit I may have had. In times of stress when I was on a deadline, or completing a project and had little time, their understanding and patience was invaluable to me. Without their care and support, I most certainly would not be where I am today.

I am incredibly lucky to call Kaitlin Kinzer my wife, as she has been the rock I have leaned on through my time in the PhD program. When I first decided to pursue a PhD in another city, we endured a year apart, until she made the sacrific to find a job and move to San Diego with me. She stood by me as most of my time and focus went into my research, and helped me when I was stressed, giving her unwaivering support. There was no one more pivotal in providing emotional support and guidance through this process, and made my pursuit of a PhD possible in the first place.

The material in this dissertation is based on following listed papers.

Chapter 2 is a partial reprint of the material as it appears in: S. Kinzer, JK. Kim, S.

Ghodrati, B. Yatham, A. Althoff, D. Mahajan, S. Lerner and H. Esmaeilzadeh, "PolyMath: A Computational Stack for Cross-Domain Acceleration", int *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*, February 27–March 3 2021, Seoul, South Korea. The dissertation author was the principal investigator and author of this paper.

Chapter 3 is a partial reprint of the material as it appears in S.Kinzer, S. Ghodrati, R. Mahapatra, BH. Ahn, E. Mascarenhas, X. Li, J. Matai, L. Zhang, H. Esmaeilzadeh, "Restoring the Broken Covenant Between Compilers and Deep Learning Accelerators Deep Learning Accelerators". The dissertation author was the principal investigator and author of this paper.[1]

Chapter 4 is a partial reprint of the material as it appears in: H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, AB. Kahng, JK. Kim, **S. Kinzer**, R. Mahapatra, SD. Manasi, E. Mascarenhas, S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. Yatham, and Z. Zeng, "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis", int *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, November 1–4 2021, Virtual. The dissertation author was a co-investigator and co-author of this paper.

[1]© 2021 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

VITA

2014	B. S. in Computer Science and Engineering, Santa Clara University
2021	M. S. in Computer Science, University of California San Diego
2023	Ph. D. in Computer Science, University of California San Diego

PUBLICATIONS

S.Kinzer, S. Ghodrati, R. Mahapatra, BH. Ahn, E. Mascarenhas, X. Li, J. Matai, L. Zhang, H. Esmaeilzadeh, "Restoring the Broken Covenant Between Compilers and Deep Learning Accelerators Deep Learning Accelerators"

D. Wang, J. Lou, N. Jin, E. Mascarenhas, R. Mahapatra, **S. Kinzer**, S. Ghodrati, A. Yazdanbakhsh, H. Esmaeilzadeh, N. Kim. "MESA: Microarchitecture Extensions for Spatial Architecture Generation." In International Symposium on Computer Architecture (ISCA), 2023.

H. Esmaeilzadeh, S. Ghodrati, AB. Kahng, JK. Kim, **S. Kinzer**, S. Kundu, R. Mahapatra, SD. Manasi, E. Mascarenhas, S. Sapatnekar, Z. Wang, and Z. Zeng, "Physically Accurate Learning-based Performance Prediction of Hardware-accelerated ML Algorithms", *2022 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD '22)*, September 12–13, 2022, Snowbird, Utah.

JK. Kim, BH. Ahn, **S. Kinzer**, S. Ghodrati, R. Mahapatra, B. Reddy, S. Wang, D. Kim, P. Sarikhani, B. Mahmoudi, D. Mahajan, J. Park, and H. Esmaeilzadeh, "Yin-Yang: Programming Abstractions for Cross-Domain Multi-Acceleration", *IEEE Micro, Special issue on compiling for accelerators*, August 1, 2022.

BH. Ahn, **S. Kinzer**, and H. Esmaeilzadeh, "Glimpse: mathematical embedding of hardware specification for neural compilation", *Proceedings of the 59th ACM/IEEE Design Automation Conference*, July 10–14 2022, San Francisco, California.

H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, AB. Kahng, JK. Kim, **S. Kinzer**, R. Mahapatra, SD. Manasi, E. Mascarenhas, S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. Yatham, and Z. Zeng, "VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis", *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, November 1–4 2021, Virtual.

S. Kinzer, JK. Kim, S. Ghodrati, B. Yatham, A. Althoff, D. Mahajan, S. Lerner and H. Esmaeilzadeh, "PolyMath: A Computational Stack for Cross-Domain Acceleration", *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*, February 27–March 3 2021, Seoul, South Korea.

S. Ghodrati, BH. Ahn, JK. Kim, **S. Kinzer**, B. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, NS. Kim, C. Young and H. Esmaeilzadeh "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks", *Proceedings of the 53th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 17–21 2020, Athens, Greece.

S. Ghodrati, H. Sharma, **S. Kinzer**, A. Yazdanbakhsh, K. Samadi, J. Park, NS. Kim, D. Burger and H. Esmaeilzadeh "Mixed-Signal Charge-Domain Acceleration of Deep Neural Networks through Interleaved Bit-Partitioned Arithmetic", *Proceedings of the 29th Annual IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 3–7 2020, Atlanta, Georgia.

ABSTRACT OF THE DISSERTATION

Polymorphic Compilation for Cross-Domain Acceleration

by

Sean Kinzer

Doctor of Philosophy in Computer Science
(Computer Engineering)

University of California San Diego, 2023

Professor Hadi Esmaeilzadeh, Chair

With general-purpose compute stacks struggling to meet computational demands of emerging applications, there has been a shift in industry and the research community toward domain-specific architectures. Each of these specialized architectures is designed with domain-specific properties in mind, exploiting their algorithmic structure by using specialized hardware capabilities. However, the divergence of architecture designs makes them incompatible with mature compilation platforms which were designed to target general purpose processors, and do not incorporate the domain knowledge necessary for optimizing specific classes of program. Further, the applications motivating these domain-specific architectures are not confined to a single algorithmic domain.

Existing approaches for implementing end-to-end applications leave developers with few choices. On the one hand, general purpose compilation stacks are expressive enough to implement all of the different algorithms included in this task, at the cost of diminishing performance from general purpose processors. On the other hand, domain specific architectures tend to require their own distinct compilation frameworks because of the unique architectural features as well as the need for a constrained language to restrict functionality. This means a programmer must separately implement and compile each part of their program with a different interface and compiler.

Alternatively, this dissertation sets out to provide a solution which compromises between these two choices by allowing a degree of expressiveness for a subset of connected algorithm domains to achieve performance benefits in end-to-end applications they are used in. Specifically, this work enables cross-domain acceleration, which is the acceleration of applications composed of tasks from multiple aglorithm domains using a single programming interface. In addition, this work achieves polymorphic compilation, which is defined as using the same compilation algorithms for different domain-specific architectures. Together, these accomplishments form a compute stack capable of polymorphic compilation for cross-domain acceleration. This compute stack demonstrates it's effectiveness and flexibility through VeriGOOD-ML where it is used to compile and optimize deep learning and data analytics programs to two different parameterizable accelerators with unique instruction sets.

Chapter 1

Introduction

1.1 Motivation

With general-purpose compute stacks struggling to meet the computational demands of emerging applications, there has been a shift in industry and the research community toward domain-specific architectures [33, 46]. These specialized architectures harness unique features of a target algorithm domain to achieve impressive improvements in both performance and energy efficiency. A significant amount of focus within this shift has been on designing architectures for Machine Learning (ML) [34, 22, 38, 30, 77, 76, 13, 58, 106, 23, 95, 14, 45, 21, 61, 57, 108, 12, 49, 72]. However, real-world applications are often not confined to being composed entirely of ML algorithms, and instead span multiple different algorithm domains, each with their own unique properties. As an example, Figure 1.1 demonstrates the sequence of actions required to enable self-driving cars, where first raw sensor data from cameras is ingested and processed by a Digital Signal Processing (DSP) algorithm, followed by sensor fusion of the surrounding environment, which is input to a deep learning algorithm for object detection, and lastly the percieved distance and objects in the environment are used to take an action through a control theory algorithm.

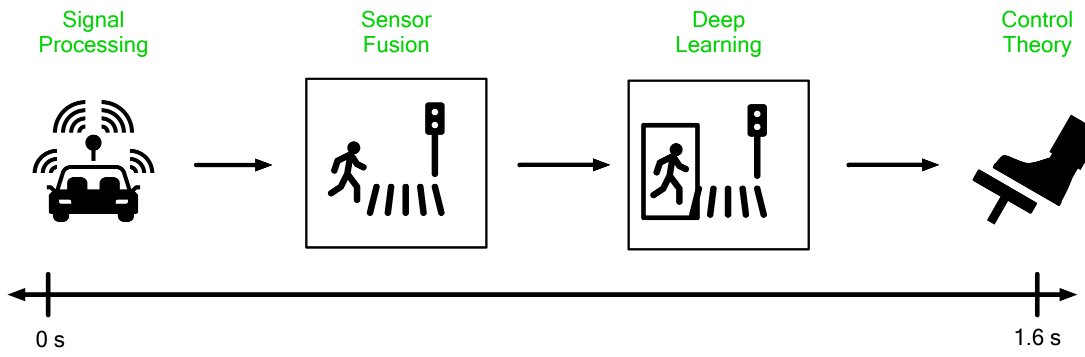


Figure 1.1: The sequence for enabling self-driving cars relies on four separate algorithm domains, which all must be completed in less than 1.6 seconds to match the average human reaction time.

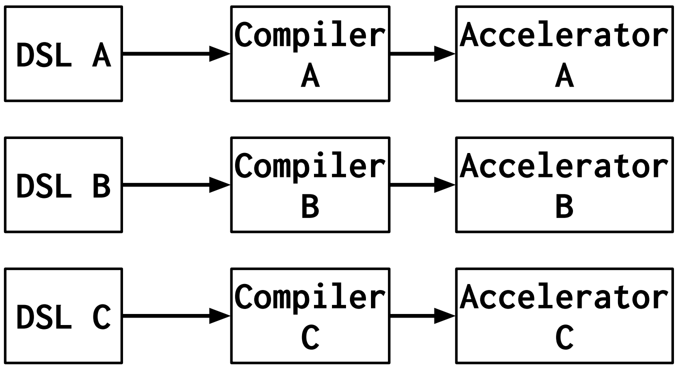
The performance of each individual algorithm within this sequence is crucial for making self-driving vehicles feasible, as the vehicle must match human response time to be usable on public roads.

To implement such an application, developers are left with the challenges of defining a program spanning multiple algorithm domains and meets the performance and energy effiency requirements from real-world conditions. As Figure 1.2 demonstrates, this challenge is complicated by existing compilation stacks, requiring developers to choose between using either (1) a general-purpose compilation stack expressive enough to implement the entire application using a single interface, but lacking performance, or (2) multiple different Domain Specific compilation stacks for each compilation target, requiring implementation of separate programs for each stack and additional code to stitch each program together, but capable of high performance and energy efficiency.

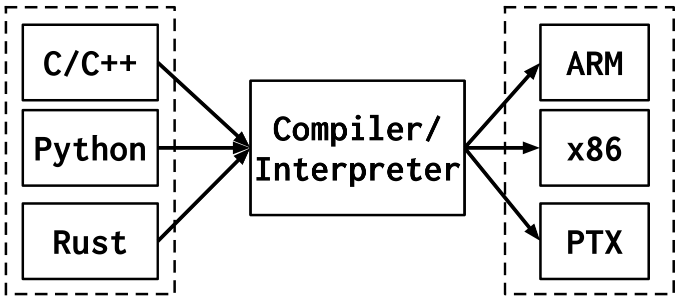
An alternate approach is to define a middle-ground which focuses on meeting the performance and energy efficiency requirements imposed by real world applications for a subset of algorithm domains, but not all domains. This dissertation sets out to realize this middle-ground through a combination of **Cross-Domain Acceleration** and **Polymorphic Compilation**.

1.2 Cross-Domain Acceleration

Cross-Domain Acceleration is defined as accelerating an application composed of tasks from multiple algorithm domains using a single interface. Therefore, achieving Cross-Domain Acceleration entails defining a Cross-Domain Language (CDL) as a unified interface for a subset of algorithm domains. Single-domain languages, referred to as Domain Specific Languages (DSLs), have grown substantially in recent years [19, 98, 79, 28, 6, 84] due to their impressive ability to simplify domain-specific program implementation. In addition, each DSL is backed by a lower-level abstraction which harnesses domain specific characteristics for both compilation to similarly specialized targets as well as optimization. Therefore, a CDL must simultaneously provide familiar syntactic constructs spanning multiple algorithm domains to simplify implementation, and also use an intermediate representation (IR)



(a) DSA Workflow.



(b) General-Purpose Processor Workflow.

Figure 1.2: Designing end-to-end applications requires developers to choose from expressive general-purpose compilation stacks or multiple, high-performance compilation stacks with unique interfaces.

capable of capturing each domain's unique characteristics for optimization. Together, a CDL and IR capable of meeting these requirements will achieve cross-domain acceleration. To this end, this dissertation explores the foundations of cross-domain acceleration by proposing a cross-domain stack composed of a CDL and IR capable of accelerating end-to-end cross-domain applications.

1.3 Polymorphic Compilation

Although enabling cross-domain acceleration addresses the issue of expressiveness across multiple algorithm domains, there is still the task of compiling an end-to-end application to one of many different specialized Domain-Specific Architectures (DSAs). One way to accomplish this task is to define separate compilation algorithms for each of the potential architecture targets; a daunting task which would likely consume a large amount of time. Instead, **Polymorphic Compilation** is defined as using the same compilation algorithm for multiple different Domain-Specific Architectures (DSAs). With DSAs being distinct from one another and possessing diverse compute granularities, resources, and limitations, using the same algorithm for each one necessitates an abstraction capable of capturing these properties to inform the compiler. In addition, DSAs tend to expose more control to the compiler and are composed of hetereogenous compute units compared to general purpose processors which rely on opaque caches and use fixed compute granularity. Therefore, a low-level IR which explicitly encodes all scheduling details, from on-chip data movement to where compute operations will be executed and in what order, is required. By introducing the architecture abstraction as a compilation input, the constraints, capabilities, and resources can be used to guide a unified compilation algorithm and achieve polymorphic compilation. This dissertation proposes such an architecture abstraction and low-level IR, and demonstrates it's ability to perform polymorphic compilation.

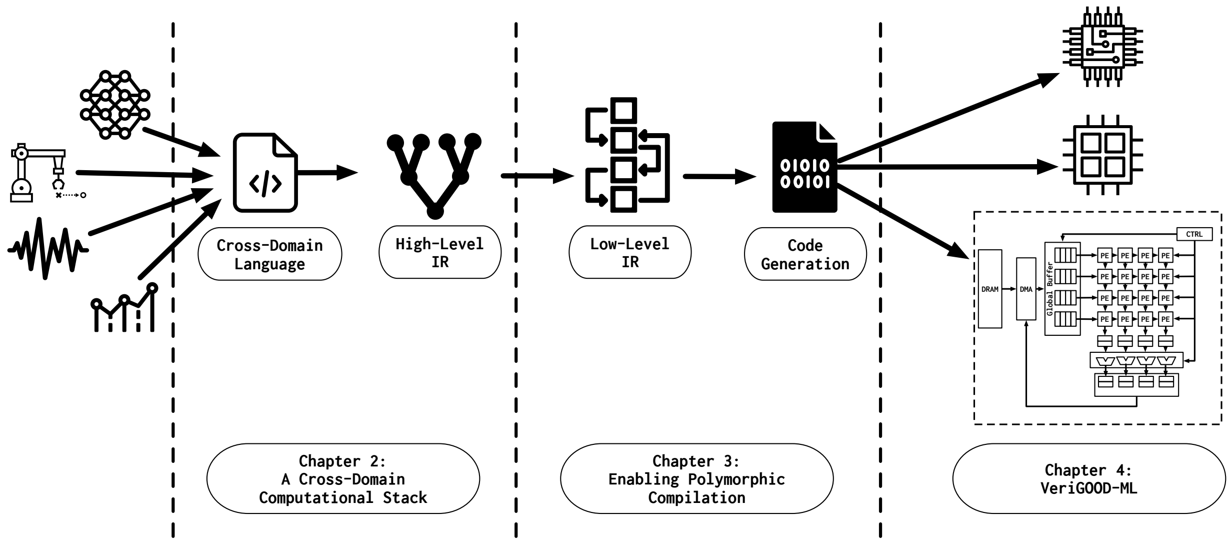


Figure 1.3: Dissertation overview.

1.4 Contributions

This section provides an overview of this dissertation, which consists of three chapters. The first two chapters delve into the details of how cross-domain acceleration and polymorphic compilation are achieved, followed by the details of a multi-year effort by a team spread across three different universities resulting in an automated methodology for generating verilog capable of executing programs compiled using the proposed polymorphic compiler. The overall workflow for this dissertation is shown in Figure 1.3.

1.4.1 A Cross-Domain Computational Stack

Domain-specific accelerators obtain performance benefits by restricting their algorithmic domain. These accelerators utilize specialized languages constrained to particular hardware, thus trading off expressiveness for high performance. The pendulum has swung from one hardware for all domains (general-purpose processors) to one hardware per individual domain. The middle-ground on this spectrum–which provides a unified computational stack across multiple, but not all, domains–is an emerging and open research challenge. This chapter sets out to explore this region and its associated tradeoff between expressiveness and performance by defining a cross-domain stack, dubbed PolyMath. This stack defines a high-level *cross-domain language (CDL)*, called PMLang, that in a modular and reusable manner encapsulates mathematical properties to be expressive across multiple domains–Robotics, Graph Analytics, Digital Signal Processing, Deep Learning, and Data Analytics. PMLang is backed by a *recursively-defined* intermediate representation allowing *simultaneous access* to all levels of operation granularity, called *sr*DFG. Accelerator-specific or domain-specific IRs commonly capture operations in the granularity that best fits a set of Domain-Specific Architectures (DSAs). In contrast, the recursive nature of the *sr*DFG enables simultaneous access to all the granularities of computation for every operation, thus forming an ideal bridge for converting to various DSA-specific IRs across multiple domains. Our stack unlocks multi-acceleration for end-to-end applications that cross the boundary of multiple domains each comprising different data and compute patterns.

Evaluations show that by using PolyMath it is possible to harness accelerators across the five domains to realize an average speedup of 3.3× over a Xeon CPU along with 18.1× reduction in energy. In comparison to Jetson Xavier and Titan XP, cross-domain acceleration offers 1.7× and 7.2× improvement in performance-per-watt, respectively. We measure the cross-domain expressiveness and performance tradeoff by comparing each benchmark against its hand-optimized implementation to achieve 83.9% and 76.8% of the optimal performance for single-domain algorithms and end-to-end applications. For the two case studies of end-to-end applications (comprising algorithms from multiple domains), results show that accelerating all kernels offers an additional 2.0× speedup over CPU, 6.1× improvement in performance-per-watt over Titan Xp, and 2.8× speedup over Jetson Xavier compared to only the one most effective single-domain kernel being accelerated. Finally, we examine the utility and expressiveness of PolyMath through a user study, which shows, on average, PolyMath requires 1.9× less time to implement algorithms from two different domains with 2.5× fewer lines of code relative to Python.

1.4.2 Enabling Polymorphic Compilation for Domain-Specific Accelerators

Deep learning accelerators are emerging as the vehicle to deal with the compute intensity of Deep Neural Networks (DNNs). The benefits of these domain-specific architectures stem from deviation from the fine-grained Von Neumman model of execution. Instead, these accelerators exploit the algorithmic structure of the application domain by matching it to specialized hardware capabilities. Four challenges make compilers for these designs different than ones targeting conventional general-purpose processors. First, more micro-architectural features and components need to be exposed, considered, and controlled by the compiler. As an example, an accelerator compute block typically expose coarser-grained operations than an ALU that performs an individual addition instruction (e.g., a systolic array that performs a whole matrix operation). Second, the on-chip storage is no longer a limited set of registers backed by a hardware-manage cache, it is usually several software-managed scratch pads with various access semantics. Third, the interconnection for on-chip data movement and off-chip loads/stores needs to be handled explicitly by the compiler and with the appropriate granularity (e.g., tile size). Last but not least, the compiler needs to match the rather coarse-grained operations (layers) of a DNN to the varying granularity of computation and storage that is supported by the hardware. These challenges call for rethinking of the compiler that has traditionally focused on generating fine-grained instruction sequences while the micro-architecture is abstracted away almost completely.

This chapter, alternatively, sets out to offer a novel abstraction for exposing the architecture structure and its varying coarser-grained capabilities through a construct dubbed Architecture Covenant Graph (ACG). This graph abstractly represents compute units, memory modules, and interconnection and their programmable capabilities. By making the architecture accessible to the compiler with the appropriate abstraction, the compilation workflow can adapt to modifications in accelerator design without necessitating redevelopment of the compiler. This is made possible by accompanying the ACG with mutable constructs, called Codelets, that initially express the functionality of the DNN operations, and are progressively transformed by the compiler to become execution mappings and schedules on the ACG. Our Covenant compiler brings these together to target significantly different deep learning accelerators without the need to be redeveloped. We compile 14 DNN layers from transformer models, deep recommender systems, and convolutional neural networks on two different architectures. The Covenant compiler achieves 93.8% of the performance of state-of-the-art compilers that use hand-tuned DNN layer implementations.

1.4.3 VeriGOOD-ML

This chapter introduces VeriGOOD-ML, an automated methodology for generating Verilog with no human in the loop, starting from a high-level description of a machine learning (ML) algorithm in a standard format such as ONNX. The Verilog RTL is then translated through a back-end design flow to GDSII, driven by a design planning approach that is well tailored to the macro-intensive nature of ML platforms. VeriGOOD-ML uses three approaches to build ML hardware: the TABLA platform uses a dataflow architecture that is well suited to non-DNN ML algorithms; the GeneSys platform, with a systolic array and a SIMD array, is optimized for implementing DNNs; and the Axiline approach synthesizes small ML algorithms by hardcoding the structure of the algorithm into hardware, thus trading off flexibility for performance and power. The overall approach explores the design space of platform configurations and Pareto-optimal-PPA back-end implementations to yield designs that represent different tradeoffs at the algorithmic level between area, power, performance, and execution time. The overall methodology, from architecture to back-end design to hardware implementation, is described in this chapter, and the results of VeriGOOD-ML are demonstrated on a set of ML benchmarks.

Chapter 2

A Cross-Domain Computational Stack

2.1 Introduction

End-to-end applications ranging from delivery drones [83] to smart speakers [5] cross multiple domains. One such application senses the environment, (1) pre-processes the sensory data, feeds it to a (2) perception module that in turn invokes a (3) decision making process to determine actions. Perception is currently reigned by Machine Learning (ML), which has attracted significant attention, but applications are not just ML. Sensory data processing relies on algorithms from Digital Signal Processing (DSP) while Control Theory and Robotics bring forth the final action that may also feed the perception module. Even though these domains work in tandem to realize an entire application, they are becoming isolated by the current push towards Domain-Specific Accelerators (DSAs). One the one hand, these accelerators tradeoff generality for performance and energy efficiency by restricting programmability to a single domain [44]. On the other hand, the traditional general-purpose computational stack cannot meet the computational demands of emerging applications [46, 33, 116]. Although, Domain-Specific Accelerators (DSA) bridge this performance gap, but make implementation an arduous task of dealing with

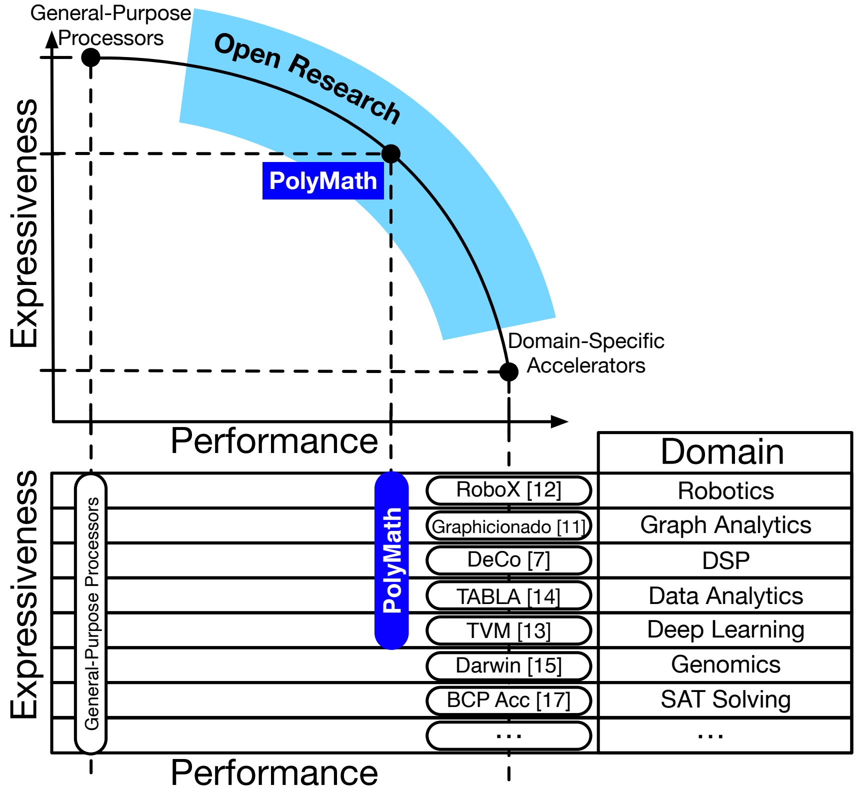


Figure 2.1: Emerging tradeoff: Expressiveness vs. Performance.

isolated programming interfaces. Thus, expressiveness is becoming limited, making the composition of an end-to-end application a major challenge for execution on accelerators. As a consequence, users seeking to create compute-intensive applications composed of algorithms from different domains must choose between either using a lower-performance, general-purpose processor or bear the burden of manually stitching together various domain-specific accelerators.

This emerging challenge creates a new tradeoff between performance and expressiveness, illustrated in Figure 2.1. On one extreme, we have General-Purpose Processors that allow expressing *all* domains at the cost of performance and/or efficiency [44]. On other extreme, are domain-specific accelerators [56, 21, 103, 40, 22, 39, 57] that can only support a *single* domain to be executed on one particular specialized architecture, thus are very performant. Even though certain DSAs offer computational stacks, composing an end-to-end application that crosses the boundary of many domains requires intimate knowledge of multiple different interfaces and various hardware accelerators to obtain high performance. Recent efforts such as, Graphicionado [43], RoboX [102], TVM [19], Tabla [77], aim to unify high-level coding within a single domain, cross-domain stacks for accelerators still remains an open challenge (Figure 2.1).

As Figure 2.1 illustrates, by addressing this challenge, PolyMath defines a new point in the Expressiveness vs. Performance design space. The ingredients of our approach are:

1. Exploiting the mathematical similarities across domains to design a modular and reusable language, PMLang, that is expressive across Robotics, Graph Analytics, DSP, Data Analytics, and Deep Learning. It offers one-to-one mapping between code and mathematical formulation while retaining modularity, thus making it familiar to both domain-experts and software engineers. PMLang offers light-weight type modifiers based on domain semantics to enable accelerators to handle on-chip and off-chip data allocation, storage, and transfer. PMLang is not an abstraction over existing domain

specific languages, rather it explores a novel dimension of designing a unified, stand-alone language across multiple domains. This unique dimension defines a class of languages that we refer to as Cross Domain Languages (CDLs). To enable cross-domain acceleration, PMLang and its associated compilation stack takes an initial step to bridge CDLs with domain-specific architectures, that are constrained to a single domain.

2. To preserve expressiveness and provide flexibility for compilation to different accelerators, we devise an intermediate representation that is a *recursively-defined* Dataflow Graph, providing *simultaneous* access to all levels of operation granularity (*sr*DFG). The compute granularity of the kernels is not uniform across different accelerators, required for cross-domain settings. Thus, we define *sr*DFG recursively in that its nodes are also *sr*DFGs. As such, *sr*DFG uniquely offers simultaneous access to various levels of computation granularity within a single program, thus enabling leveraging different accelerators. This capability enables cross-domain multi-acceleration–acceleration of a cross-domain application on different accelerators. The flexible, recursive nature of our IR shown in Figure 2.2, whose edges preserve the type modifier metadata.

3. PolyMath uses a modular compilation framework that conveniently enables creation and application of pipelined compilation passes on the *sr*DFG. To convert *sr*DFGs to executable accelerator code, PolyMath offers a graph lowering algorithm with a conversion strategy which uses metadata embedded in the *sr*DFG edges to flexibly translate the graph nodes. The lowering algorithm applies transformations which produce a new *sr*DFG made up of compute kernels at the same granularity of the target accelerator. Once lowered, the metadata associated with the *sr*DFG edges is translated to the accelerator's own IR for final binary generation through its own scheduling and mapping framework.

4. Last but not least, PolyMath will be the very first extensible, modular, and open-source computation stack to enable the community to innovate and explore the impending

challenges of cross-domain acceleration at a time when domain-specific compilation stacks, except for DNNs, are elusive. Although there are many accelerators in the literature, many of their compilation stacks are not available. By making PolyMath open-source and extensible, the community can add other domains which align with the core mathematical constructs in PMLang.

We show PolyMath's balance of expressiveness and performance by compiling twelve different algorithms across robotics, graph analytics, digital signal processing, data analytics, and deep learning. These workloads achieve an overall speedup of 3.3× over a Xeon CPU along with 18.1× reduction in energy. In comparison to Titan Xp and Jetson Xavier GPUs, cross-domain acceleration offers 7.2× and 1.7× in energy reduction. We next measure the tradeoff of cross-domain algorithm expression and find that PolyMath can achieve 83.9% of the performance of the same algorithms implemented in their native stack's language. We also study two end-to-end applications that cross multiple domains. Accelerating all the kernels offers an additional 2.0× speedup over CPU, 6.1× additional improvement in energy requirements over Titan Xp, and 2.8× speedup over Jetson Xavier vs when only one most effective kernel was accelerated. The results show that when only a part is accelerated, the slower non-accelerated kernels dictates the overall improvement, whereas, when all the algorithms in the application are accelerated Amdahl's burden reduces, and the improvement of all the domains is magnified. To evaluate the usability and expressiveness of PMLang relative to Python, we conduct a user study and found that on average PMLang required 1.9× less time to implement algorithms with 2.5× fewer lines of code. Finally, end-to-end performance of Polymath is 76.8% of the performance of two manually implemented end-to-end applications. Given the fact that PolyMath offers greater ease of programming compared to Python, the automation overhead of 23.1% (=100%-76.8%) is a fair bargain.

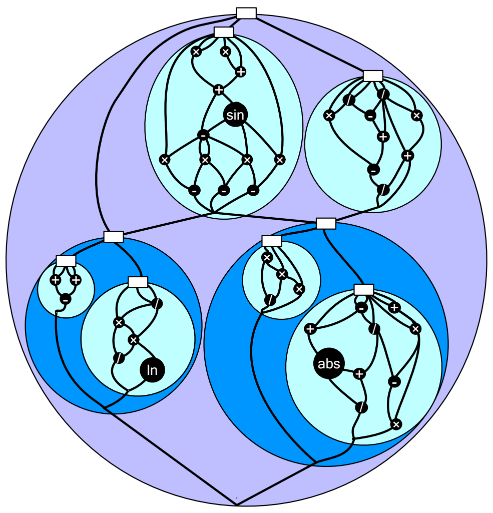


Figure 2.2: Visual representation of PolyMath's *sr*DFG.

2.2 **PMLang**: Mathematical Programming Interface

PMLang is designed to encapsulate the mathematical properties of these domains, as they are tied together by similar operations on multi-dimensional data, include minimal control-flow, and share use-cases such as cyber-physical systems. Consider the following application in its entirety: An end-to-end neuroscience application requires multiple domains to study the impact of deep brain stimulation on movement disorders and goes through the following steps: (1) convert raw electrocorticographic (ECoG) brain signals to frequency domain using fast Fourier transform (FFT); (2) apply logistic regression to classify these frequency domain signals into various biomarkers; (3) based on the classification, use model predictive control to send an optical stimulation back to the brain. This application crosses three domains, DSP, Data Analytics, and Control Theory in each iteration to generate deep brain stimulation signals.

There are numerous domain specific architectures for each of these algorithms/domains individually; however, using them for this application would require writing each part of the application in a different DSL, compiling them separately, and manually joining their executables. Instead, PMLang allows users to write their application as a *single program*, thus, eliminating the overhead of stitching together stacks to execute the program across multiple domain specific architectures.

Keeping the properties of target domains in mind, PMLang is designed to reduce the time to code a mathematical expression into a formula-based textual format, enabled by language constructs for modularity and light-weight type modifiers. Moreover for code organization and reduction in implementation time, PMLang includes reusable execution code blocks called components that perform operations on flows of data. These components encapsulate a task comprised of either other components and/or mathematical expressions which use traditional, imperative syntax to facilitate familiarity for experienced programmers. For *modularity* and *reusability*, components have distinct boundaries and arguments which are distinguished by type modifiers consisting of input, output, state, and param; each of which is associated with how the component will use the argument, shown in Table 2.1. By using type modifiers in component arguments to explicitly identify data semantics PMLang binds operations to data being operated on for accelerators to take advantage of.

The remainder of the section will delve into details of PMLang constructs through an example. For brevity, we show the PMLang program for Model Predictive Control (MPC) from Control Theory used for Robotics. MPC attempts to solve a constrained optimization problem over a finite sequence of inputs. MPC can also be used for the aforementioned brain stimulation application. We provide MPC in the context of a mobile two-wheeled robot performing trajectory tracking, shown in Figure 2.3. Here, the sensors send the current state of the robot as inputs to a model that predicts the next location, then optimizes a sequence of control signals for moving the robot to the predicted location. The program finds the optimal sequence of control signals, `ctrl_mdl`, over a finite period of time to match a reference trajectory, `pos_ref`, that specifies the position and orientation of the robot. At each point in time, the actual position and orientation of the robot is

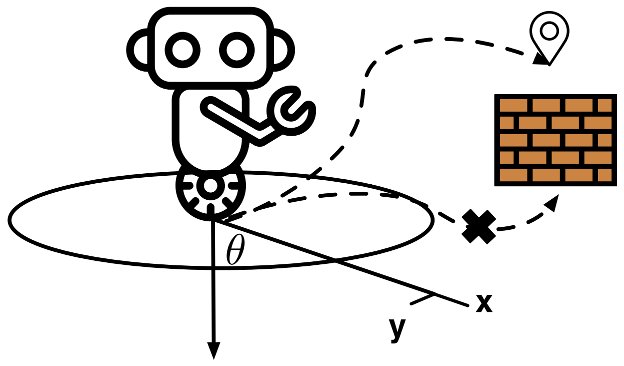


Figure 2.3: Visual of trajectory tracking for wheeled robot.

input to the algorithm, which optimizes the `ctrl_mdl` and sends the next control signal, `ctrl_sgnl`, back to the robot. This process is summed up in the following steps:

Input: `pos` $\rightarrow$ the (x,y,θ) orientation of the robot.

Output: `ctrl_sgnl`$\rightarrow$ (ν, ω) control consisting the velocity and angular velocity to be sent to the robot.

State: $u \rightarrow (v, \omega)$ linear and angular velocities across a pre-determined time horizon `h`.

Step 1: Make a prediction Using input `pos` and cost matrices `P` and `H`, predict the position and angle of the robot across horizon `h`.

Step 2: Compute the gradient of the objective function Calculate the error on the predicted position and orientation, `pos_pred`, using pre-computed gradient coefficient matrices `HQ_g` and `R_g`.

Step 3: Update the control model and send the output signal Using gradient, `g`, update the control model, and send the output control signal, `ctrl_sgnl`.

2.2.1 Components

Components form the building blocks of PMLang, and are used to delineate different parts of the program into multiple levels of execution. To delineate the access semantics for each argument of the components, PMLang uses type modifiers ((input), `output`, `state`, `param`). Using type modifiers relieves programmers from concern about the underlying

```
predict_trajectory(input float pos[a],
              input float ctrl_mdl[b],
              param float P[c][a],
              param float H[c][b],
              output float pred[c]){
  index i[0:a-1], j[0:b-1], k[0:c-1];
  pred[k] = sum[i](P[k][i]*pos[i]);
  pred[k] = pred[k] + sum[j](
      ↪ H[k][j]*ctrl_mdl[j]);
}
update_ctrl_model(input float ctrl_prev[b],
          input float g[b],
          output float ctrl_mdl[b],
          output float ctrl_sgnl[s],
          param int h){
  index i[0:b-2], j[0:s-1];
  ctrl_sgnl[j] = ctrl_mdl[h*j];
  ctrl_mdl[(h-1)*j] = 0;
  ctrl_mdl[i] = ctrl_prev[(i+1)*h] - g[(i+1)*h]
      ↪ ;
}
mvmul(input float A[m][n],
    input float B[n],
    output float C[m]){
  index i[0:n-1], j[0:m-1];
  C[j] = sum[i](A[j][i]*B[i]);
}
compute_ctrl_grad(input float pos_pred[c],
         input float ctrl_mdl[b],
         input float pos_ref[c],
         param float HQ_g[b][c], // Input Cost
      ↪ Gradient
         param float R_g[b][b], // Cost Inverse
      ↪  Hessian
         output float g[b]){
  index i[0:b-1], j[0:c-1];
  float P_g[b], H_g[b];
  err[j] = pos_ref[j] - pos_pred[j];
  mvmul(HQ_g, err, P_g);
  mvmul(R_g, ctrl_mdl, H_g);
  g[i] = P_g[i] + H_g[i];
}
main(input float pos[3],
    state float ctrl_mdl[20],
    param float pos_ref[30],
    param float P[30][3],
    param float HQ_g[20][30],
    param float H[30][20],
    param float R_g[20][20],
    output float ctrl_sgnl[2]){
  float pos_pred[30], g[20];
  index i[0:9], j[0:1];
RBT: predict_trajectory(pos, ctrl_mdl, P, H,
    ↪ pos_pred);
RBT: compute_ctrl_grad(
    ↪ pos_pred,ctrl_mdl,pos_ref,HQ_g,R_g,g);
RBT: update_ctrl_model(ctrl_mdl, g, ctrl_mdl,
    ↪ ctrl_sgnl,10);
}
```

Figure 2.4: MPC for MobileRobot trajectory tracking in PMLang.

accelerator-specific mechanisms for data exchange between different components. An `input` argument is used to feed data into the component, and is read-only; an `output` argument is used to return data from a component, and can only be written to; a `state` argument can be read or written to, and represents data that is part of the state of the component, thus is preserved across invocations/iterations; and a `param` argument is a constant that is used to parameterize the component. These type modifiers describe whether or not the data will be re-used (`state`), kept unchanged (`param`), or used once and discarded (`input`/`output`). As an example, line 1 shows that argument `pos` is an `input` to the `predict_trajectory` component. As another example, line 41 shows a `state` argument named `ctrl_mdl` which indicates that `ctrl_mdl` is used, updated, shared across invocations of `main`, which matches the MPC semantics of optimizing the control model over a series of time steps. Type modifiers also enable custom accelerators to place `input` data such as `pos` in Read-only FIFO buffers to reduce data communication overhead and hardware memory logic, or store `state` data such as `ctrl_mdl` on-chip on the accelerator for fast repeated data accesses. Using a single set of type modifiers to describe data semantics across multiple domains unifies program implementation for end-to-end applications. This is exemplified in robotics and deep learning, where one domain uses "model" and the other "weight" to describe the same data semantics, both of which are described as `state` data in PMLang.

In addition to being reusable, these components allow users to conceive their program as a collection of sub-steps at varying levels of granularity making it adaptable for compilation to different accelerators. To instantiate a component, the programmer specifies its name and arguments. An example of component instantiation is shown in line 49-51 where `predict_trajectory`, `compute_ctrl_grad`, and `update_ctrl_model` is instantiated. Each instantiation creates a copy of the component, as if it were inlined. This is in contrast to conventional languages that rely on a function call stack which is sequential in nature. Instead, inlining enables the program to be mapped to our *sr*DFG IR, which preserves opportunities for parallelism based on data flow dependencies.

2.2.2 Index Variables

PMLang is based on mathematical notations that do not use for loops and instead use indices (e.g., $\sum_{i=0}^{n-1} A_i$). To simplify programming based on formulae, PMLang uses `index` variables to concisely specify operations performed over ranges of multi-dimensional data without using explicit for loops. In its most basic form, an `index` variable represents a range of integers, specified by its lower and upper bounds. Line 48 shows two such `index` variable declarations: `i` and `j`. This approach reveals the inherent parallelism in mathematical formulae since operations expressed using this approach are naturally vectorizable without performing any loop transformations. For example, below is a PMLang statement iterating over all `j`s, each of which can be performed in parallel.

```
index j[0:s-1];
err[j] = pos_ref[j] - pos_pred[j];
```

Strided indexing. To support strided/non-sequential indexing (e.g., convolution), PolyMath also supports arithmetic operations on index variables as shown below.

```
ctrl_mdl[i] = ctrl_prev[(i+1)*h] - g[(i+1)*h];
```

Boolean conditional over indexing. Unlike a domain-specific language such as Tabla [77] that focuses solely on data analytics, PMLang allows Boolean conditionals to be applied to indices, which provides support for other domains such as graph analytics and robotics. For instance, the following computes the sum of the non-diagonal parts of the matrix `A`:

```
index i[0:N-1], j[0:M-1];
res = sum[i][j: j != i](A[i][j]);
```

Support for Boolean conditionals and non-sequential `index` variables flexibly incorporates common as well as specific-to-domain characteristics of algorithms across robotics, graph analytics, DSP, data analytics, and deep learning. These features distinguish PMLang from DSLs which either use (1) concise operations expressed through a fixed API (e.g., named functions such as "dense" in TVM [19]), or (2) simply do not support these construction since the specific target domain does not require them (e.g., TABLA [77].)

2.2.3 Mathematical Operations

Index variables allow for a nearly one-to-one mapping between mathematical notation and PMLang code. PMLang offers standard mathematical operators to be used with multi-dimensional data, expressed in a single statement by using index variables. PMLang's syntax for math expression of $C_j = \sum_{i=0}^{n} A_{j,i} \times B_i$ is:

```
C[j] = sum[i](A[j][i]*B[i]);
```

Non-Linear operations. PMLang includes a set of built-in functions to be used in math expressions commonly used across the multiple target domains, including non-linear operations such as cosine/sine (DSP, robotics), gaussian (robotics, DSP, data analytics), sigmoid/ReLU (deep learning, data analytics), etc. Including non-linear operations as part of PMLang simplifies algorithm expression and allows PolyMath to leverage the performance benefits of non-linear compute units in custom accelerators.

Reduction operations. PMLang is also equipped with built-in group reduction operations such as `sum`, `prod`, `max` etc., to calculate the summation ($\sum$), product (Π), or maximum value of a sequence of numbers. These group reductions operations are converted to srDFG with two levels of granularity: (1) the outer group DFG node that (2) encapsulates the scalar inner operations (nodes). This multi-granular representation enables the compiler to map the the outer encompassing node to a dedicated unit if the accelerator harbors it. Otherwise, the inner basic nodes are mapped to individual ALUs. This crucial flexibility is a unique feature of PolyMath and enables it be cross domain and target different accelerators.

Custom reduction operations. PMLang also supports custom group reduction operations, as they are commonly used in graph analytics and DSP algorithms. Custom reduction operations can be defined in PolyMath by specifying the arithmetic for a given set of input arguments. Below is an example of the definition of the `min` reduction function and using

Table 2.1: A subset of PMLang's keywords and definitions.

Language Construct	Keyword	Description
Component	`string name`	Takes input, produces output, and reads/writes to state arguments
Domain	`RBT`, `GA`, `DSP`, `DA`, `DL`	Specifies a component's target domain
Type Modifiers	`input`	Flow of data, can be exclusively read from within a component scope
	`output`	Flow of data, can be exclusively written to within a component scope
	`param`	Constant parameter used to parameterize a component
Index Types	`state`	Flow of data, can be written to or read from within a component scope
	`index`	Specifies ranges of operations
Types	`bin`, `int`, `float`, `str`, `complex`	Data types used to for variable declarations.

it to find the minimum value for the matrix `A`:

```
reduction min(a,b) = a < b ? a : b;
res = min[i][j](A[i][j]);
```

2.2.4 Domain Annotations

PMLang uniquely targets multiple domains, each of which is eventually accelerated with a Domain-Specific Architecture. As such, PMLang offers a light-weight mechanism to specify the target domain for *only top-level* component instantiations without tying it to a specific accelerator. All of the code within a component also inherits the same domain, which alleviates the programmer from having to annotate all component instances in their program. This is done by simply adding one of the five keywords: `RBT` (Robotics), `GA` (Graph Analytics), `DSP` (Digital Signal Processing), `DA` (Data Analytics), and `DL` (Deep Learning), as demonstrated below:

```
  RBT: predict_trajectory(pos, ctrl_mdl, P, H, pos_pred);
```

2.3 Simultaneous-Recursive DataFlow Graph

Accelerator-specific or domain-specific IRs commonly capture operations in the granularity that best fits the target architecture. One of the major challenges that PolyMath faces is targeting multiple domains each of whose accelerators operate on different granularities of computation. Even within a single domain, various architectures accept

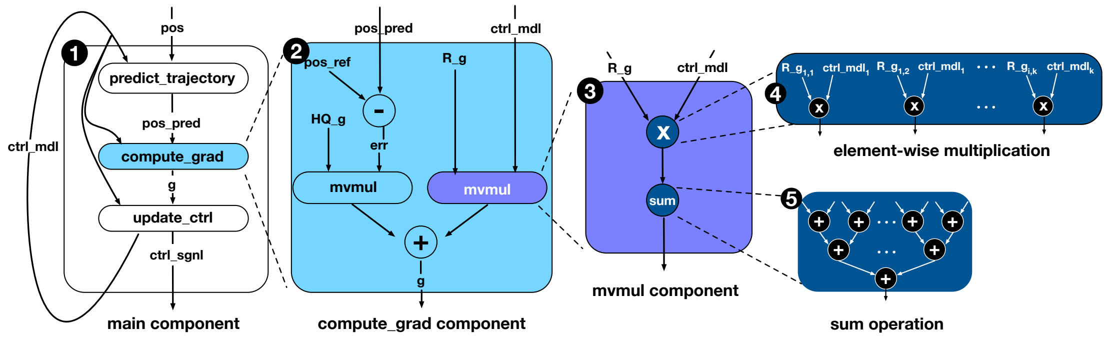


Figure 2.5: Overview of the *sr*DFG MobileRobot algorithm including zoomed-in views of its multiple levels of recursion.

computation in different granularities. To address this challenge, we designed *sr*DFG, an intermediate representation which is a *recursively-defined* dataflow graph, and provides *simultaneous* access to each level of recursion, *sr*DFG. Our *sr*DFG enables the compiler to simultaneously access all the granularities of computation for every component, thus forming the ideal bridge to convert to various accelerator-specific IR. Furthermore, *sr*DFG enable multi-acceleration for end-to-end applications that cross the boundary of multiple domains with different data and compute patterns across Robotics, Graph Analytics, DSP, Data Analytics, and Deep Learning. Next, we describe the *sr*DFG structure using Figure 2.5, a visual representation of the MobileRobot algorithm described in Section 2.2.

2.3.1 *sr*DFG Definitions

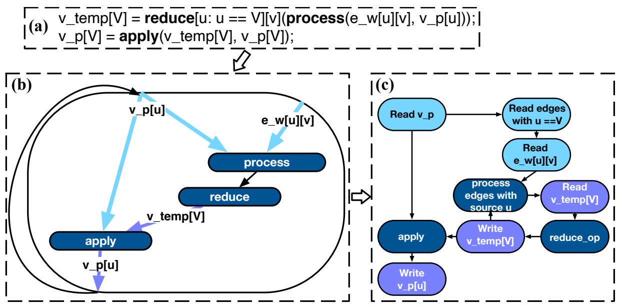
An *sr*DFG is defined as a pair, *(N,E)*, of nodes N representing PMLang operations, and edges E representing input or output operands. An *sr*DFG node $n \in N$ is a pair *(name, srdfg)* of a string representing the name of an operation, and its lower-granularity operations *sr*DFG composition. Each numbered box in Figure 2.5 represents the *srdfg* for different nodes at varying granularities within the MobileRobot algorithm. As shown in ❷: both the the subtraction operation and the `mvmul` component are nodes. An edge $e \in E$ is a tuple of source *src* and destination *dst* nodes, and the edge metadata *md*: *(src,dst,md)*. Edge metadata consists of the type, type modifier, and shape of the operand associated with the edge. For math operations, input and output edges represent operands and results, whereas adjacent edges in component instantiations represent state, input and output arguments. This is illustrated in ❶, where `pos` is the input argument, `ctrl_sgnl` is the output argument, and `ctrl_mdl` is the state argument that creates a cycle for multiple iterations. Given a node n, we denote the name and *srdfg* as *n.name* and *n.srdfg*, and similarly denote *src,dst,md* in an edge e as *e.src*, *e.dst*, and *e.md*. Lastly, the domain annotations previously described are translated to the *srdfg.domain* attribute for each *srdfg*.

2.3.2 *sr*DFG Semantics

As an example, the *sr*DFG shown in ❸ will begin operations when the data in edges `R_g` and `ctrl_mdl` are ready. Each *sr*DFG is a statically defined graph representing a single instantiation on its input values, with each component instantiation or operation getting its own *sr*DFG, which allows for computing context sensitive information. As an example, Figure 2.5 ❷ shows two unique nodes and pairs of input edges for the `mvmul` component, and as a result each instantiation of `mvmul` gets its own node and *sr*DFG. The *sr*DFG in ❷ also shows how edges propagate their metadata to the lower granularity nodes, as the shapes of `R_g` and `ctrl_mdl` determine the number of element-wise multiplication nodes in ❹. The type modifier included in edge metadata can change depending on its *sr*DFG, as shown in ❶ where the `ctrl_mdl` edge is a reusable *state*, but is an input edge for ❷.

2.3.3 Enabling Different Accelerators

Custom accelerators support unique set of operations performed on a variety of different typed and shaped inputs and outputs. To ensure flexibility, each *sr*DFG includes operations as nodes, n, as well as the more fine-grained operations to define the node, *n.dfg*. To illustrate this point, each *sr*DFG in Figure 2.5 represents a possible operation



```
v_temp[V] = reduce[u: u == V][v](process(e_w[u][v], v_p[u]));
v_p[V] = apply(v_temp[V], v_p[V]);
```

Figure 2.6: Graph Analytics algorithm compilation starting from (a) a PMLang program compiled to an (b) *sr*DFG which is lowered and converted to (b) Graphicionado[43] pipeline block IR.

supported by an accelerator. If ❷ is supported by the target accelerator, the *sr*DFG can be transformed to consist only of the operations represented by each node in ❶. If ❷ is not supported but ❸ is supported, then the operations in both ❶ and ❷ will be selected for compilation. PolyMath's base unit of lowering is a node, and if the nodes in the *sr*DFG cannot be lowered to a specific hardware because of unsupported nodes, the compilation fails for that accelerator.

Each of these accelerator operations is closely tied to the types and shapes of its operands. The *sr*DFG uses the edge metadata to specify the operand information when performing compilation of a node to an acceleration operation. For example, an accelerator might support the element-wise multiplication in ❹, but requires the number of elements being multiplied to perform the operation. Each input edge to ❸ includes the shape as part of its metadata, which allows for compilation of ❹. Domain-specific accelerators differentiate how data is stored by receiving this information from the programmer on how the variables are used. For example, the `ctrl_mdl` edge in ❶ has the `state` type modifier which causes the accelerator to store the data local to the component.

2.4 Compilation Framework

PolyMath performs compilation in three steps: (1) compilation from PMLang to an *sr*DFG; (2) lowering the *sr*DFG to the granularity of the different domains and converting it to the accelerator's IR; (3) invoking the accelerator's provided compiler to generate the final binaries. Figure 2.6 illustrates these phases for a PMLang graph analytics implementation which is compiled to an *sr*DFG, and then lowered and converted to Graphicionado's pipeline IR.

2.4.1 *sr*DFG Generation

For each PMLang program, compilation forms an Abstract Syntax Tree (AST) using syntax analysis. The program AST is then traversed and a symbol table S is created, storing information contained in each component. The component information consists of variable names and variable metadata *md* (e.g., edge type, type modifier, and shape). For each component, a DFG is formed by stitching statements together using static single assignment. Lower-level, *sr*DFG operations are formed for element-wise operations or group operations, which means edges may represent both scalar and multi-dimensional values.

After completing AST traversal generating *sr*DFGs from PMLang statements, a single *sr*DFG is generated starting with the highest level component `main`. Previously, component statements were skipped because all component *sr*DFGs were not created. When the `main` *srdfg* is traversed, a component node n_c is created using previously skipped component statements and their argument type modifiers, preserving the domain annotation as n_c.*dfg.domain* attribute. Edges adjacent to n_c are added to *srdfg* by using the type modifiers for arguments in the component signature of n_c, where type modifiers are (1) in/out edge sfor *input*/*output*, and (2) edges such that $e =$ *(src,dst,md)* where *src* $=$ *dst* for state. The *sr*DFG is repeated for each component statement recursively, creating nodes and edges to generate the lower levels of operation granularity for each component, which inherit the top-level domains.

2.4.2 Example *sr*DFG Passes

PolyMath implements a modular framework and set of APIs that enable custom, target-independent passes over the IR. These passes take an *sr*DFG as an input and produce a transformed *sr*DFG. This feature conveniently enables applying pipelines of passes on the same IR. Also, traditional passes such as constant propagation, constant folding, etc. are supported via this PolyMath pass infrastructure. We cover one such compiler pass below.

Algebraic combination. Transformation passes in PolyMath benefit from simultaneous access to all levels of operation granularity for a program. This bolsters traditional compiler passes such as algebraic simplification that are typically limited by single granularity IRs which hide opportunities for simplification. In contrast, a PolyMath pass can identify hidden algebraic simplifications which span multiple levels of granularity which would remain obscured in other flat IRs. As an example, if an *sr*DFG with a top-level matrix-vector multiplication is added to the output of another matrix-vector operation contained in another node's subgraph, the matrix vector operations can be fused together by concatenating their inputs. This transformation opportunity remains unidentified in flat IRs, but PolyMath uniquely reveals these transformation prospects by preserving a program's multi-granularity in the *sr*DFG and supporting transformations crossing granular boundaries.

2.4.3 Compilation from *sr*DFG to Accelerator IR

Compiling a *sr*DFG to a domain-specific architecture consists of (1) lowering *sr*DFG operations supported by the target accelerator (Algorithm 1) and (2) forming valid accelerator IR by translating and combining each *sr*DFG node (Algorithm 2). Algorithm 1

Algorithm 1: *sr*DFG Lowering Algorithm

```
function Lower(srdfg, $O_m$)
    let (N,E) = srdfg.subDfg
    let $O_t = O_m[srdfg.domain]$ for each $n \in N$ do
        if $n.name \notin O_t$ then
            let subDfg = Lower($n$, $O_m$)
            $srdfg \leftarrow srdfg[n \longmapsto subDfg]$
    return srdfg
```

Algorithm 2: Compilation Algorithm

```
function CompileProgram(srdfg, AccSpec)
    let $\pi_d \leftarrow \emptyset$  for  $d \in Domains$
    let (N,E) = srdfg
    for each $n \in N$ do
        let $(+_d,\ m_d) = AccSpec[n.domain]$
        let $t = m_d[n.name]$
        $\pi_d = \pi_d + t(srdfg,\ n)$
        for each $in\_edge \in n$ do
            if $(n.domain \neq in\_edge.src.domain)$ then
                $\pi_d = \pi_d + t_{load}(in\_edge,\ n)$
        for each $out\_edge \in n$ do
            if $(n.domain \neq out\_edge.dst.domain)$ then
                $\pi_d = \pi_d + t_{store}(n,\ out\_edge)$
    return $\pi_{d1}, \ldots, \pi_{dn}$
```

is a function Lower which takes as input a dataflow graph *srdfg* which is a pair *(N,E)* of nodes N and edges E defined in Section 2.3.1. PolyMath lowers *sr*DFG operations to different domains with accelerator targets that support different granularity operations. Lower uses a map, O_m, with domain names as keys, and lists of domain-specific accelerator operation names, O_t, as values to lower *sr*DFG nodes to the correct granularity.

The algorithm consists of first using the *srdfg.domain* attribute as a key to determine the correct granularity of operations for lowering, storing the set of supported operations in O_t. For each node n, if *n.name* is not included in O_t, *n.srdfg* inherits the *srdfg* domain, and lowers n in *srdfg* by replacing it with a *sr*DFG comprised of only supported operations. Replacing n in the *srdfg* consists of substituting *src* or *dst* in adjacent edges *(src,dst,md)* with a node in the *subDfg* if $src = n$ or $dst = n$. Once each $n \in N$ has been replaced with supported operations based on O_t. By preserving different levels of granularity in the *sr*DFG, the same *sr*DFG is capable of generating *dfg*'s with operations supported by a variety of custom accelerators. For instance, the hierarchy of RoBoX begins at the `System` level, followed by finer grained `Task` computations all the way down to varying operation granularities in it's macro dataflow graph, such as `Vector`, `Scalar`, and `Group` operations.

Once a *srdfg* has been lowered, it is compiled to an IR suitable for the accelerator using Algorithm 2. Accelerator IR for a domain d, denoted with π_d, is comprised of accelerator IR fragments, each of which is a basic operator and its arguments. To generate each π_d for the different targets, Algorithm 2 takes as input a lowered *sr*DFG produced by Algorithm 1, and accelerator specifications for the targets corresponding to each domain. Acceleration specifications for each domain are stored in *AccSpec*, and define how *sr*DFG nodes are translated and merged to form accelerator IR. A specification for domain d is a pair $(m_d, +_d)$ where:

- m_d is a map from operator names to a translation function for that operator. The translation functions, t, works as follows: given a *srdfg* and a node n, $t(srdfg,n)$ returns the accelerator IR fragment π_d representing the accelerator operation for n.
- $+_d$ is an operator that combines an accelerator IR π_d and an accelerator IR fragment produced by t_d.

Having defined the necessary variables, we can describe Algorithm 2. The algorithm extracts the nodes and edges in the *sr*DFG, then applies a translation function to each node, creating an accelerator IR fragment. Each IR fragment for each of the program's domains is separately accumulated into complete representations of an accelerator program IR, and are returned by the algorithm.

The most complicated part of the compilation are the translation functions, t. The translation function does two things: (1) identify the correct accelerator IR operation, and (2) assign the correct arguments for that operator. Assigning the correct arguments uses these steps:

1. Convert types to the equivalent accelerator type

2. Use edges with *input* type modifier as input arguments
3. Use edges with *output* type modifier as output arguments
4. Initialize IR variables for edges with the *state* type modifier
5. Add constants for arguments with *param* type modifier

If the accelerator IR fragment requires the shape of operation arguments, it is also included as part of the arguments to the operator or declaration operation. To ensure data is transferred between domain boundaries, load and store IR fragments are created when there are sources and destinations with different domains than a node. As a final step, accelerator provided compilers are used to create binaries from the generated IR.

Compilation flexibility. The combination of Algorithm 2 and 1 enables compilation to different types of domain-specific accelerator because of two key properties. First, simultaneous access to each level of *sr*DFG recursion allows supported accelerator operations to be translated. Unsupported *sr*DFG nodes on the particular accelerator are refined and transformed to the appropriate level of granularity through recursion which enables identification of the accelerator-supported operations. Second, the metadata stored in the *sr*DFG allows the IR generation to be parameterized based on the target accelerator. As a result, users can create different accelerator specifications for different accelerators and these same algorithms will do the appropriate mapping. Each algorithm can be instantiated for a number of different mappings without changes to the high-level algorithm.

2.5 Evaluation

Table 2.2 illustrates the difference between conventional general-purpose stacks, domain-specific stacks in the literature, and PolyMath. As shown, PolyMath represent a middle ground between domain-specificity and generality, enabling cross-domain multi-acceleration.

Table 2.2: A comparison of computational stacks.

Domain	General-Purpose Processors	Graphicionado [43]	Darwin [114]	DNNWeaver [106]	TVM [19]	TABLA [77]	RoboX [102]	DeCO [56]	BCP Acc [26]	PolyMath
Robotics	✓	✗	✗	✗	✗	✗	✓	✗	✗	✓
Graph Analytics	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓
DSP	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓
Data Analytics	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓
Deep Learning	✓	✗	✗	✓	✓	✗	✗	✗	✗	✓
Genomics	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗
SAT Solvers	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗

2.5.1 Experimental Setup

Algorithms and Datasets.

Table 2.3 shows workloads from Robotics, Graph Analytics, DSP, Data Analytics, and Deep Learning domains and the lines of code (LOC) for the PMLang implementation. Table 2.4 shows a break down of end-to-end application domains, algorithms, configurations, and PMLang LOC.

Single domain workloads. In *Robotics* domain, we have two benchmarks, MobileRobot [65] and Hexacopter [59]. Section 2.2 discusses the two-wheeled MobileRobot in detail. Hexacopter is a six-rotor micro UAV that uses motion planning and orientation control to determine trajectory. For both these workloads the physical robot and task specification is expressed in PMLang. For *Data Analytics* we have Low Rank Matrix Factorization (LRMF) and Kmeans clustering. LRMF converts a large matrix into two smaller matrices, which if taken product of, represent the original matrix. For LRMF we use two Movielens [41] datasets. Kmeans clustering partitions data into k-clusters. For one Kmeans workload cluster hand written digits with mnist [71] dataset. The second benchmark uses data from the UCI repository [73] to cluster households with similar electricity consumption. In the *Digital Signal Processing* domain we have four benchmarks, two for each Fast Fourier Transform (FFT), and Discrete Cosine Transform (DCT). The FFT implementation is

Table 2.3: Benchmarks and workloads used to evaluate PolyMath.

Domain	Benchmark	Algorithm	Config/Dataset	PMLang LOC
Robotics	**Mobile Robot**	Model Predictive Control	Trajectory Tracking, Horizon = 1024	52
	Hexacopter	Model Predictive Control	Altitude Control, Horizon = 1024	197
Graph Analytics	**Twitter Followers**	Breadth-First Search	#Vertices=61.57M, #Edges=1468.36M	14
	Wikipedia Links	Breadth-First Search	#Vertices=3.56M, #Edges=84.75M	14
	LiveJournal	Single Source Shortest Path	#Vertices=4.84M, #Edges=68.99M	14
Data Analytics	**MovieL (100k)**	Low Rank Matrix Factorization	1682 movies, 943 users; 100000 ratings	43
	MovieL (20M)	Low Rank Matrix Factorization	40110 movies, 259137 users; 244096 ratings	43
	DigitCluster	K-Means Clustering	784 features;120000 images;K=10	41
	ElecUse	K-Means Clustering	4 features; 2075259 data points; K=12	41
DSP	**FFT-8192**	Fast-Fourier Transform	1D FFT-real; 8192x1 input	12
	FFT-16384	Fast-Fourier Transform	1D FFT-real; 16834x1 input	12
	DCT-1024	Discrete Cosine Transform	1024x1024 image; 8x8 kernel, stride=8	31
	DCT-2048	Discrete Cosine Transform	2048x2048 image; 8x8 kernel, stride=8	31
Deep Learning	**ResNet-18**	Deep Neural Network	Batch Size = 1, ImageNet	117
	MobileNet	Deep Neural Network	Batch Size = 1, ImageNet	102

Table 2.4: Algorithmic composition of end-to-end applications.

Benchmark	Algorithm	Domain	Config/Dataset	LOC
Brain Stimul	Fast-Fourier Transform (FFT)	DSP	1D FFT, 4096 Input	12
	Logistic Regression (LR)	Data Analytics	4096 features	8
	Model Predictive Control (MPC)	Robotics	Horizon = 1024	64
Option Pricing	Black-Scholes (BLKS)	Data Analytics	8192 options	10
	Logistic Regression (LR)	Data Analytics	129549 words	8

a fine-grained butterfly and bit-reversal to transform a signal to frequency domain. The DCT algorithm applies a filter kernel to an input image and is used for compression. For *Deep Learning*, we use two popular convolutional neural networks, ResNet-18 [48] and MobileNet [53] for object classification. For *Graph Analytics*, we implement and apply Breadth-First Search on two graphs, one of Twitter users and followers [67] and another of Wikipedia links [27].

End-to-end cross-domain applications. Table 2.4 shows the end-to-end applications for our case study, the different domains they comprise of, and specification of each

Table 2.5: Domains and accelerators used for evaluations.

Domain	PolyMath Accelerator	Baseline Framework
Robotics	RoBoX (ASIC)	ACADO/cuBLAS
Graph Analytics	GRAPHICIONADO (ASIC)	Intel GraphMat/Enterprise
Data Analytics	HyperStreams(FPGA) / TABLA (FPGA)	MLPack/OpenBlas/CUDA
DSP	DECO (FPGA)	FFTW3/cuFFT/NVIDIA-DCT
Deep Learning	TVM-VTA (FPGA)	TVM/Tensorflow

algorithm. The brain stimulation application,`BrainStimul`, is described in Section 2.2. The stock market application, called `OptionPricing`, predicts call option price in stock market and uses two data analytics algorithms. This application first performs sentiment analysis through logistic regression on news articles to understand market signals and then Black-Scholes to predict the price.

Optimized CPU and GPU implementations.

Table 2.5 and 2.6 shows the optimized CPU and GPU framework their specifications. The robotics's CPU implementation uses ACADO Toolkit [51] to implement optimized, self-contained C code and uses cuBLAS [90] libraries for GPUs. For graph analytics, we used Intel GraphMat [112] for CPU implementations and Enterprise [75] for GPU. The DSP workloads use C subroutine libraries [37, 93] for CPU and Nvidia implementations [91, 94] for GPU. For data analytics, we use mlpack [25], a fast and flexible C++ ML library built on top of OpenBLAS [120], NVBLAS [92], and Armadillo [104]. The Deep Learning workloads are compiled using optimized Tensorflow [6] for CPU and GPU. All of our experiments were performed on an Intel Xeon E7, Titan Xp GPU, and low-power Jetson Xavier AGX.

Domain-Specific accelerators.

Table 2.5 shows the accelerator used for each domain. To evaluate each benchmark on domain-specific accelerators, PolyMath was used to compile programs to the target accelerator IR, and the target accelerator's compiler was used to generate executable binaries.

Table 2.6: CPU, FPGA, and ASIC specifications.

	CPU	FPGA	ASIC	GPU
Chip	Xeon E-2176G	UltraScale KCU1500	RoBoX/ Graphicionado	Titan Xp/ Jetson AGX Xavier
Cores	6	-	-	3,840/ 512
Memory/ BRAM	128GB	75MB	512KB/ 64MB	12 GB/32 GB
Power	80W	35W	3.4W/7W	250W/30W
Frequency	3.7 GHz	150MHz	1GHz/ 1GHz	1.5GHz/ 1.3GHz
Logic Tables	-	1,451	-	-
Compute Units	-	5,520	256/8	-

RoBoX's [102] Macro DFG from *sr*DFG as it offers programmable ASIC for system, tasks, and penalties for control algorithms optimized using MPC. We use Graphicionado [43], an ASIC accelerator for graph analytics algorithms expressed as vertex programs, as the target for the Graph Analytics workloads. We compile FFT and DCT *sr*DFG representation to DeCO [56], a DSP block based FPGA accelerator, by translating to it's DFG, which is then compiled as executable binaries. For LRMF and kmeans we convert *sr*DFG to Tabla [7], an open source template-based FPGA accelerator for machine learning, by compiling a *sr*DFG to a DFG.We use TVM-VTA [81], a programmable deep learning FPGA accelerator, as the target for Deep Learning workloads, as it is state-of-the-art and open-source. Each DSA requires specific levels of operation granularity: single operation [77, 56], coarse DNN layers [81], and coarse time snapshots [102], enabled by each *sr*DFG's multi granularity, for mapping of kernels to the accelerator.

Multi-acceleration. For `BrainStimul`, we compile parts to DeCO [56] (FFT-4096), Tabla [77] (Logistic Regression), and RoBoX [102] accelerators. For `OptionPricing`, we execute logistic regression based sentiment analysis on Tabla [77] and Black Scholes on HyperStreams [82]. All accelerators are cascaded as a single System On Chip (SOC), comprised of memory and a host. A light-weight manager executes on the host, ensuring data dependencies between different accelerators and initiating DMA transfers between DRAM and local accelerator memory. This setting is similar to prior work [18] that also uses an array of micro-accelerators.

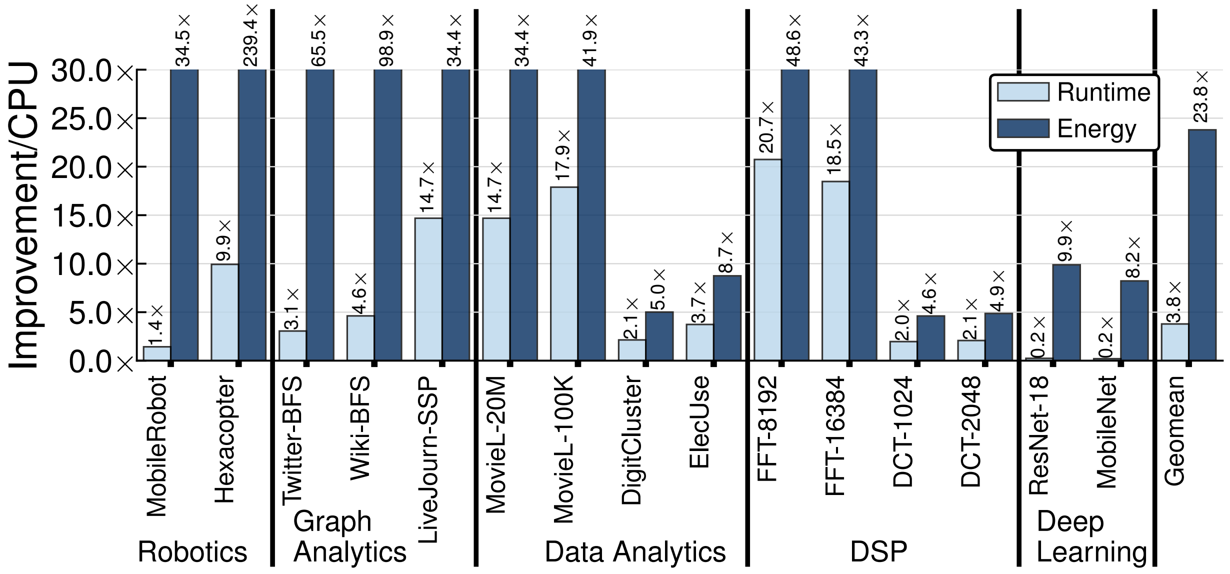


Figure 2.7: Runtime and Energy improvement of PolyMath over CPU.

2.5.2 Experimental Results

The goal of PolyMath is to facilitate the use of the wide variety of custom accelerators across end-to-end cross domain applications. It does so by abstracting away hardware level details through a versatile, extensible, and a modular stack that can maintains the required levels of kernel granularities best suited for each design. In this section we compare domain-specific accelerators executing PMLang code with optimized CPU and GPU implementations to better understand the portability of PolyMath. We then perform case studies through two end-to-end applications and observe that PolyMath allows cross domain multi-acceleration.

Performance and Energy Comparisons

Single kernel comparisons. Figure 2.7 and Figure 2.8 show the speedup of PolyMath compiled programs listed in Table 2.3 to domain specific accelerators over Xeon E-2176G CPU, Titan Xp GPU, and Jetson Xavier AGX as the baseline, respectively. On average, PolyMath translated implementations outperform Xeon E-2176G and Jetson Xavier by 3.3× and 1.2× in terms of runtime and offer 7.2× and 1.7× more Performance-per-Watt

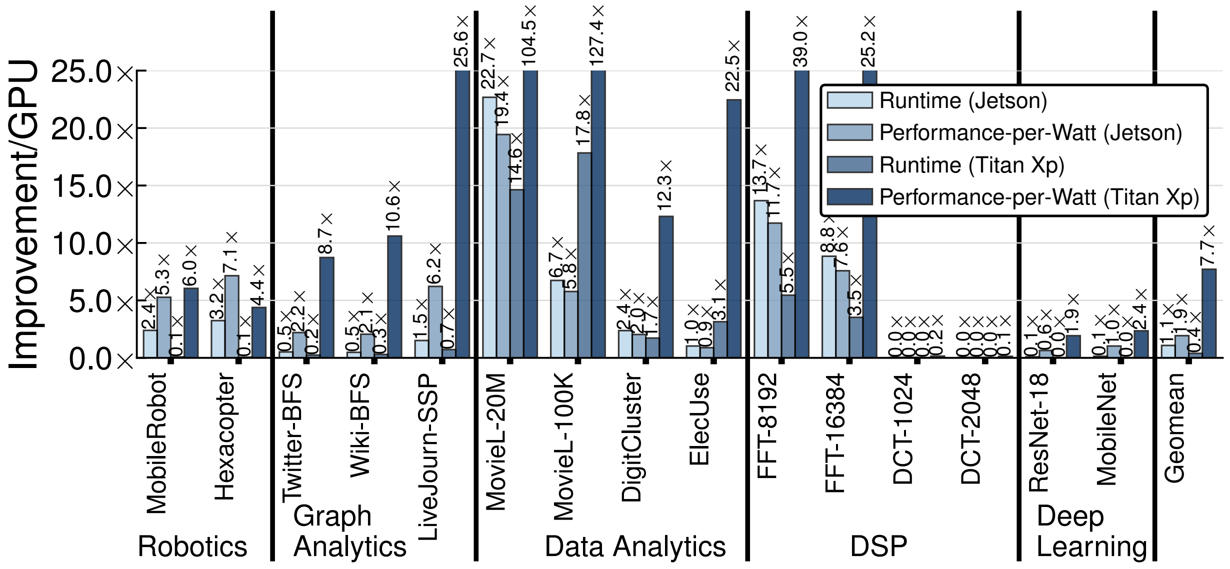


Figure 2.8: Runtime and Performance-per-Watt improvement of PolyMath over GPU.

over Titan Xp and Jetson Xavier GPUs. Smaller benchmarks such as MovieLens-100K and ElecUse are unable to fully utilize Titan XP, thus cannot obtain higher benefits in comparison to Jetson but incur higher PPW. The average still demonstrates an increase in both performance and energy in the cross-domain setting. PolyMath implementations also offer 18.1× more energy efficiency over CPUs, but due to many lower power accelerator backends only offers 40% of the GPU performance. This is especially true for discrete cosine transform and deep learning benchmarks; DCT due to its high coarse granular matrix multiplications for which DeCO a programmable FPGA accelerator is not as effective as Titan Xp and deep learning models because our backend for CNNs is VTA that is designed as a low-power accelerator but is also being compared to a high-end GPU for uniformity. Note that PolyMath does not contribute any overhead specifically for deep learning acceleration because it offers direct conversion of *sr*DFG to the TVM nodes.

Optimal performance comparison. The accelerators used for execution of PolyMath implementations also offer custom stacks that are built for their target architecture. We compare PolyMath to implementations in their native stack, which represent optimal executions to demonstrate the cross-domain overhead of our stack to native implementations.

Figure 2.9 compares the performance of PolyMath implementations with the optimal performance that can be reached by programs written by experts for each of the accelerators. The figure shows that PolyMath achieves 83.9% the optimal runtime.

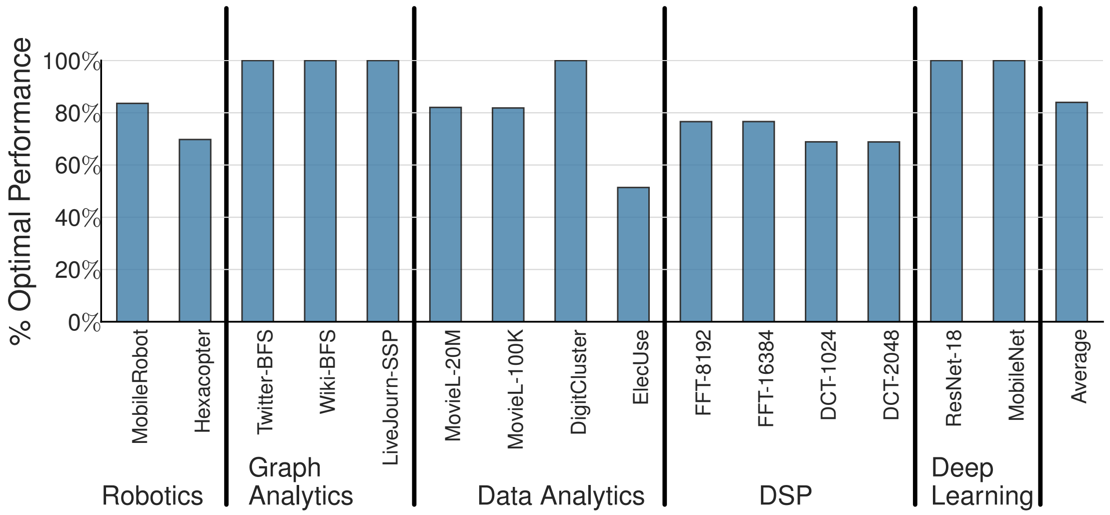
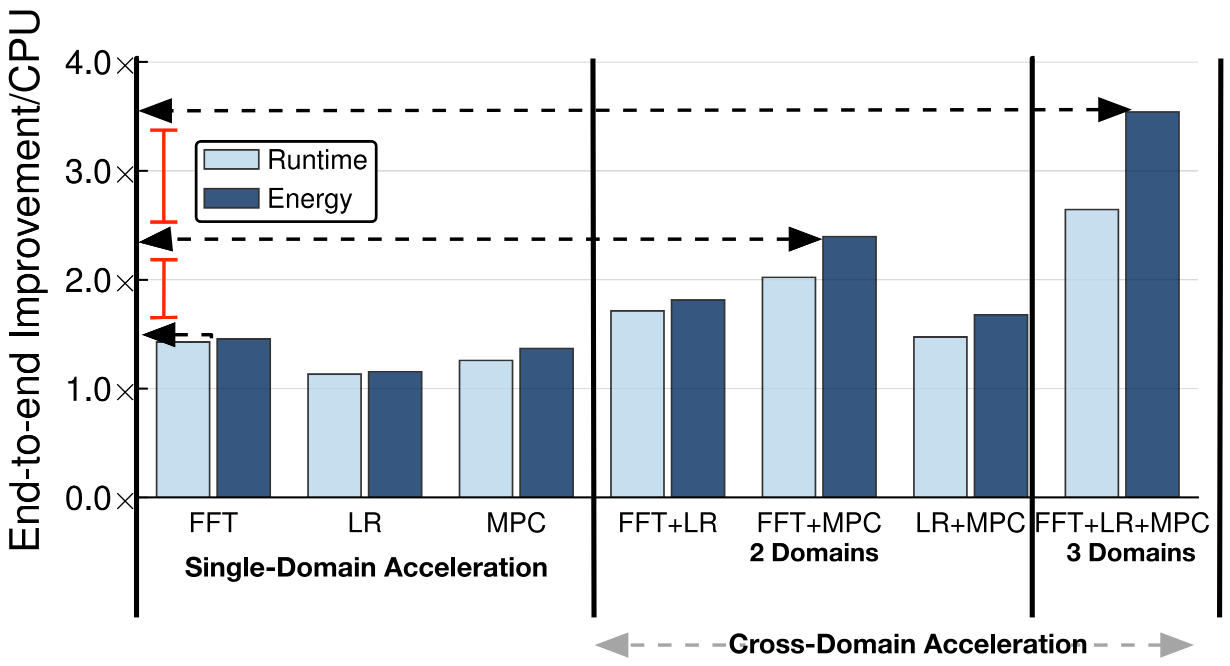


Figure 2.9: Percent of optimal runtime for PolyMath translated implementations compared to hand-tuned implementations.

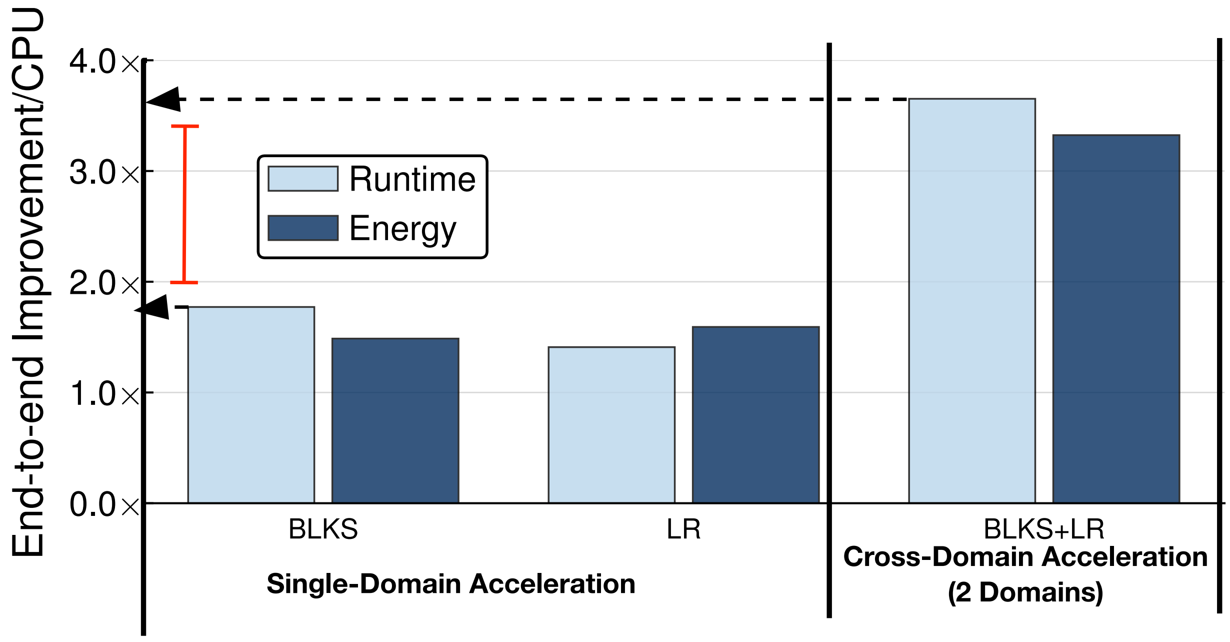
The performance of PolyMath relative to optimal implementations is dependent on the domain and the algorithm because each stack differs in the level of data semantics that can be complex to compile to from a more expressive language. For instance, accelerators specializing in Deep Learning often utilize three primary type modifiers for variables in neural network graphs: input, output, and weights–each of which can be directly mapped to type modifiers in PMLang, thus incur zero overhead. In contrast, Robotics algorithms contain unique data semantics, such as task penalties, constraints variables, time varying references, etc., which do not differentiated with PolyMath type modifiers, thus implementations do not reach optimal performance. Accelerators such as DeCO require specific topologies for their graph-based IR, i.e. balanced DFGs, because they rely on stage-based computation, which results in reduced execution time relative to PolyMath translations. In the case of Data Analytics, we see a low percentage of optimal performance for ElecUse, because the benchmark is small, which makes any extra operations included in the *sr*DFG have a more significant impact on performance relative to the optimal implementation. In contrast, DigitCluster uses the same algorithm but on a larger dataset, thus can amortize the overhead more effectively. It is important to note that the optimal performance is reached by a specialized program on each accelerator written by an expert, whereas PolyMath offers support for multiple domains constituting end-to-end applications that can be expressed as single comprehensive program, coupled with *sr*DFG to pave way for compilation to multiple accelerators.

End-to-End Application Case Study

PolyMath offers means to express cross domain applications as a single program which can be compiled to multiple accelerators pertaining to each of these domains. Figure 2.10 shows the runtime and energy improvement of the end to end applications in comparison to CPU. Figure 2.11 illustrates the runtime and performance-per-Watt improvement in comparison to Titan Xp and Jetson. Figure 2.10a and Figure 2.11a shows these results for `BrainStimul` and Figure 2.10b and Figure 2.11b for `OptionPricing` application. In these graphs we provide entire application improvement for all possible acceleration combinations, from one domain algorithm accelerated to cross-domain where all algorithms are accelerated. Each end-to-end result incorporates data communication overheads from data transfer between hardware. Stand-alone kernel acceleration, as shown in , can offer very high speedups. However, when these kernels are incorporated within a more comprehensive application, those speedups do not manifest in the entire application because the non-accelerated kernel becomes a bottleneck. For instance, the gap between the highest benefit obtained from the best single-domain acceleration and cross-domain end-to-end acceleration, is 1.85× for `BrainStimul` and 2.06× for `OptionPricing` (Figure 2.10). Every kernel that is added for acceleration not only benefits itself from specialized execution but also reduces the Amdahl's burden and magnifies other accelerated component's impact. The benefits



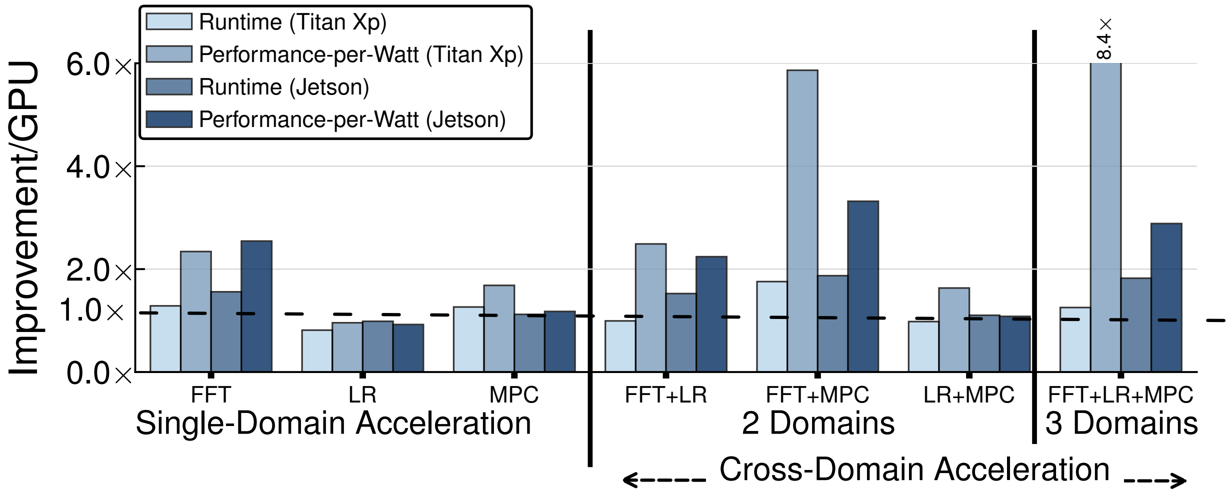
(a) `BrainStimul` brain stimulation application



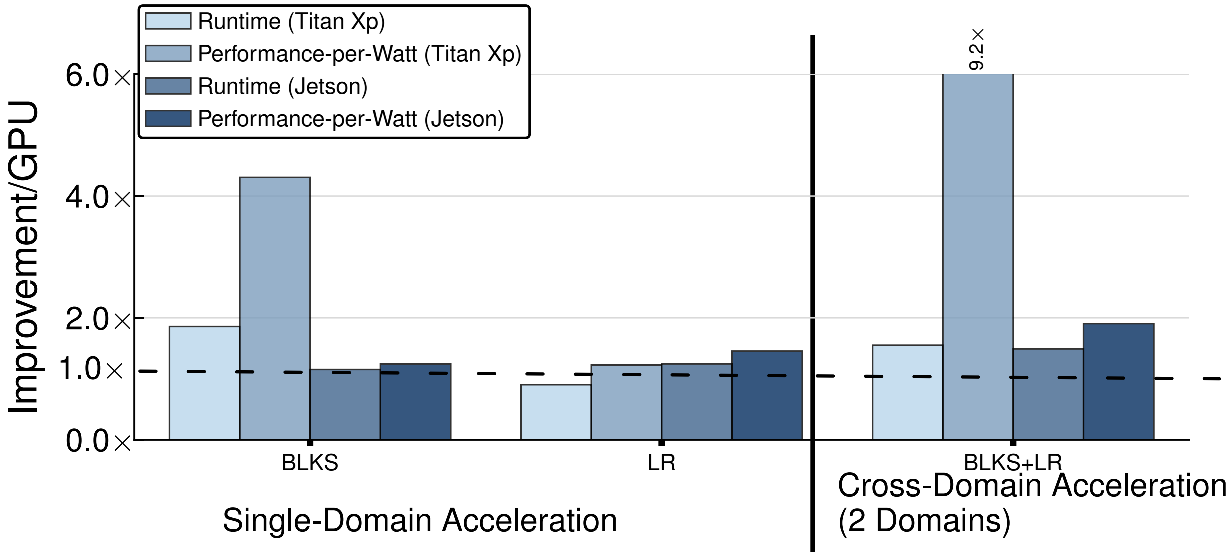
(b) `OptionPricing` finance application

Figure 2.10: Runtime and energy improvement over CPU of end-to-end applications for different combinations of accelerated domains.

of individual kernel acceleration are present despite end-to-end communication runtime overheads of 23.4% and 17.0% and energy overheads of 21.8% and 12.4% for `BrainStimul` and `OptionPricing`, respectively.



(a) `BrainStimul` brain stimulation application



(b) `OptionPricing` finance application

Figure 2.11: Runtime and Performance-per-Watt improvement over GPU for combinations of accelerated domains for end-to-end applications.

Figure 2.11a and 2.11b show the `BrainStimul` and `OptionPricing` results for both Titan Xp and Jetson. Figure 2.11a shows that PolyMath offers 1.2× runtime improvement over Titan Xp compared to 1.8× over Jetson. In contrast, PolyMath improves performance-per-watt by 8.3× over Titan Xp compared to 2.8× for Jetson due to its lower power consumption. As Figure 2.11b shows, the `OptionPricing` benchmark underutilizes the Titan Xp, only offering 1.5× and 9.2× improvement in performance and performance-per-watt compared to 1.4× and 1.9× over Jetson. This is caused by the difference in levels of coarse parallelism in the algorithms. Overall, the PolyMath implementation of `OptionPricing` still outperforms both GPUs for both runtime and performance-per-watt.

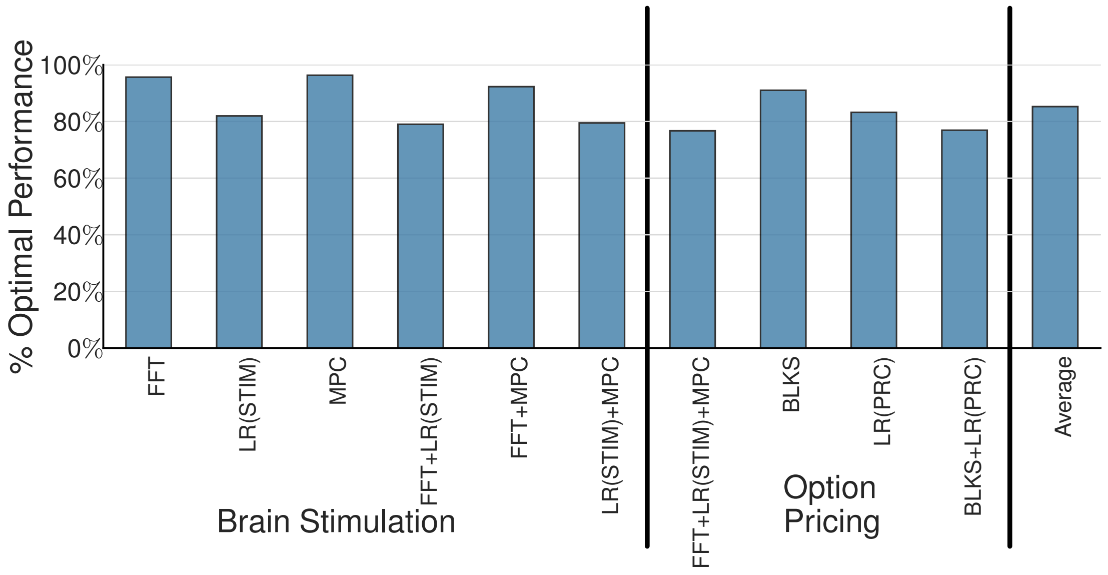


Figure 2.12: Percent of optimal performance for `BrainStim` and `OptionPricing` compared to hand-tuned implementations.

Lastly, Figure 2.12 shows end-to-end Polymath implementations achieve 76.7% optimal performance for `BrainStim` and 76.9% for `OptionPricing` compared to entirely manual implementations of each application. Given the fact that PolyMath offers greater ease of programming compared to Python (Figure 2.13), the automation overhead of 23.1% is a fair tradeoff.

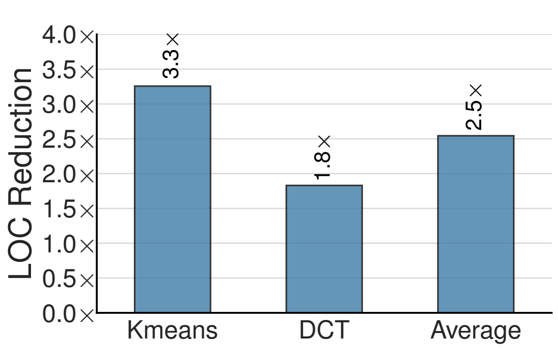
User Study

To determine the usability of PMLang, we conducted a user study with 20 programmers who are either professional software engineers or PhD students in computer science.

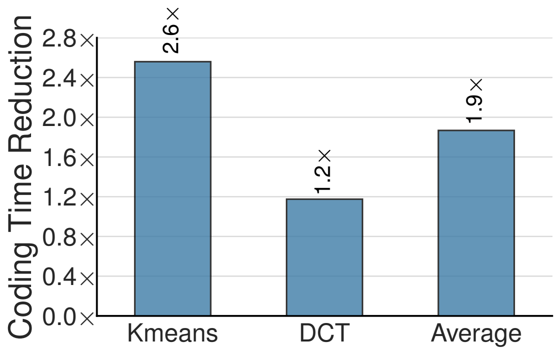
The goal of the user study is to measure the expressiveness of PMLang by comparing it to Python, an intuitive programming language which is commonly used in three of the focus domains of PolyMath: DSP, Data Analytics, and Deep Learning. The study tasked each user in the study with implementing either a DSP or Analytics algorithm in Python or PMLang. The users were divided into four groups; Kmeans in Python, Kmeans in PMLang, DCT in Python, and DCT in PMLang. For fairness and ease in expression of tensor algebraic operations, we allow the users to import Python modules such numpy [31]. Each participant in the user study has varying levels of expertise (from beginner to proficient) in the target domains, and is proficient in Python. Every participant went through the following process:

1. Participants were introduced to PMLang with a short, six-minute video which walked through the language and small examples.
2. To avoid any algorithm knowledge bias, participants were randomly assigned either the DSP or ML algorithm to implement in either Python and PMLang.
3. To minimize variation in algorithm understanding, users were not allowed to begin their first implementation before having read and confirmed their understanding of the algorithm.
4. We timed participants during their implementations and measured their Lines of Code (LOC) after completion.

Results. Figure 2.13 compares the LOC between Python and PMLang as a ratio of Python LOC to PMLang LOC in (a), and the implementation time of Python and PMLang as a ratio of Python implementation time to PMLang implementation time in (b). The results show that PMLang required 2.5× fewer lines of code on average and 1.9× less implementation time on average. The Kmeans implementation averaged 3.3× fewer LOC, whereas for DCT the average reduction of LOC is 1.8×. In general, the Kmeans algorithm is more verbose than DCT and on average required 47.6% more lines of Python code than DCT. Because



(a) Lines of Code Reduction



(b) Coding Time Reduction

Figure 2.13: Reduction in Lines of Code (LOC) and coding time with PMLang over Python for Kmeans and DCT.

there were more lines of code included in Kmeans, there was more opportunity to reduce multi-line operations to a single PMLang statement, which explains the difference in LOC reduction.

The greater complexity of Kmeans appears to have an effect in the speedup of implementation time as well, where the average speedup for Kmeans was 2.6× and the average speedup for DCT was 1.2×. Part of this speedup can be attributed to typing more LOC in PMLang, but it is also a result of being able to directly translate mathematical notation to the equivalent PMLang statement. These results indicate that the more complicated the mathematical program is, the more the programmer will benefit from implementing the program in PMLang. Further, PMLang is expressive enough for programmers unfamiliar with the language to write algebraic expressions more efficiently than they would write the same expressions in a language they are familiar with, Python.

2.6 Related Work

DSLs for custom architectures. There are various domain-specific languages designed to facilitate the use of hardware accelerators. These languages are mostly designed for a single domain [36, 17, 98, 111] or like Spatial [63], they focus on conveniently expressing lower level hardware-centric information. Another language, Halide [98], allows expression of image processing pipelines and contains constructs for filter-based algorithms. Lime [15] focuses on high-level synthesis from Java, thereby enabling execution of Java programs on both FPGAs and CPUs. Instead, PolyMath offers a *Cross-Domain Language (CDL)* and compute stack to explore the emerging tradeoff between expressiveness and performance while leveraging currently isolated, domain-specific accelerators.

Mathematical and scientific computing environments. There are numerous scientific and numerical programming environments [78, 16, 110, 97], and frameworks [31, 96]. PolyMath uniquely provides a 1-to-1 mapping of mathematical expressions to its statements and leverages the natural parallelism in formulas without any explicit annotations for vectorization In contrast, MATLAB [78], Julia [16], or R, require manual effort from the user to identify the parallelism across different computations, vectorize its code, and determine column/row arrangements for matrix operations. Moreover, these languages do not delineate between the semantics of data in their programs and do not offer a multi-granular representation, as offered by PolyMath, to enable usage of various domain-specific accelerators.

Intermediate Representations. A number of intermediate representations [69, 74] provide abstractions to enable program analysis using virtual resources. Both LLVM and JVM operate at the granularity of a single CPU instruction, which is highly inefficient for domain-specific architectures. Some works [64] have adapted LLVM to guarantee independence between parallel operation threads by using a dataflow graph structure intended for heterogeneous platforms. A number of other works [19, 80, 100, 115, 101, 70] focus on the domain of machine learning and have implemented an end-to-end approach for optimization on heterogeneous platforms after performing optimizations from a high-level language. These works supported limited algorithm domains [19, 80, 100, 115, 101], and rely on C/C++ or other general-purpose programming languages [70, 64], requiring the programmer to express complex mathematical expressions in unintuitive ways. MLIR [70] is a hierarchical, high level IR, but is general-purpose and as such is on the end of the expressiveness curve (Figure 2.1). Whereas PolyMath is restricted by its mathematical language (PMLang) to a limited set of domains, falling in the middle of the spectrum of expressiveness. Furthermore, MLIR does not have any compilation stack to support variety of accelerators from different domains as PolyMath does and is practically demonstrated in the evaluations. Tiramisu [28] introduces a scheduling language with novel commands to explicitly manage the complexities that arise when targeting multicores, GPUs, and distributed machines. Tiramisu offers an IR based on the polyhedral model to allow fine-grained optimization. As such, Tiramisu can serve as a potential backend for PolyMath that deals with the higher-level complexity of expressing cross-domain application and not low-level fine-grained optimization.

Acceleration frameworks and toolchains. TensorFlow [6] is an end-to-end open source platform for expressing ML algorithms in Python. Deep learning accelerators (e.g., TPU [57]) leverage Tensorflow. Similarly, a variety of deep learning frameworks [23, 35, 106] allow users to run their DNNs on FPGA based hardware designs. Full stack solutions such as TABLA [77] and RoBoX [102] support classical supervised machine learning and model predictive control in robotics, respectively. Other toolchains [99, 10, 9, 109] aim to simplify running deep neural networks on hardware accelerators by performing design space exploration to find the best configuration for their particular design. These solutions, however, are bound to their own custom architectures for particular platforms (FPGAs or ASICs). In contrast, the *sr*DFG offers a flexible hook that can be translated to these toolchains and frameworks as well as to future accelerator designs and platforms. The cross-domain nature of PolyMath that supports Robotics, Graph Analytics, DSP, Data Analytics and Deep learning sets it apart from these domain-specific stacks.

2.7 Conclusion

As domain-specific accelerators are becoming prevalent, there is an emerging tradeoff between expressiveness and performance. This paradigm–a pendulum swing from general-purpose processing to the opposite direction–creates implicit programming silos between different domain. This paper set out to explore the region between these extremes and explore the new expressiveness-performance tradeoff. To that end, we defined a *cross-domain* computational stack, PolyMath, that bridges the expressiveness gap between multiple domains, Robotics, Graph Analytics, DSP, Deep Learning, and Data Analytics. The results from user study and performance evaluations showed that PolyMath strikes an effective balance between expressiveness and performance while enabling cross-domain multi-acceleration. It is time to look beyond the timely, yet temporary, success of domain-specific accelerators and devise a future that enables end-to-end applications. The current approach towards acceleration excludes significant opportunities by restricting the domain. To harness these untapped opportunities, a new paradigm needs to emerge that breaks the boundaries of domains, but also preserves the benefits of domain-specificity. PolyMath takes the initial step in breaking this new ground.

2.8 Acknowledgement

Chapter 2 is a partial reprint of the material as it appears in: S. Kinzer, JK. Kim, S. Ghodrati, B. Yatham, A. Althoff, D. Mahajan, S. Lerner and H. Esmaeilzadeh, “PolyMath: A Computational Stack for Cross-Domain Acceleration”, int *2021 IEEE 27th International Symposium on High Performance Computer Architecture (HPCA)*, February 27–March 3 2021, Seoul, South Korea. The dissertation author was the principal investigator and author of this paper.

Chapter 3

Enabling Polymorphic Compilation for Domain-Specific Accelerators

3.1 Introduction

Deep Learning has taken the IT industry by a storm and it is set to penetrate various disciplines and markets from healthcare [8] and social networking [86] to gaming [117] and entertainment [29]. However, its success is predicated upon the availability of responsive execution platforms as DNNs require massive computations [32, 47]. In fact, they have become the driving use-case for the development and adoption of domain-specific accelerators [57, 50]. These new architectures require state-of-the-art and highly optimized compilers prior to even delivering the expected performance and efficiency gains.

Four challenges make compilers for these designs different than ones targeting conventional general-purpose processors. First, these architectures no longer adhere [76, 106, 21] to the long-held abstraction of fine-grained Instruction-Set Architectures (ISAs) and Von Neumann model [119]. Therefore, more micro-architectural features and components need to be exposed, considered, and controlled by the compiler. For instance, an accelerator compute block typically exposes coarser-grained operations than an ALU that performs an individual addition instruction (e.g., a systolic array performs a whole matrix operation). Second, the on-chip storage is no longer a limited set of registers backed by a hardware-managed cache, it is usually several software-managed scratch pads with various access semantics. Third, the interconnection for on-chip data movement and off-chip loads/stores needs to be handled explicitly by compiler, with the appropriate granularity (e.g., tile size). Finally, the compiler needs to match the rather coarse-grained operations (layers) of a DNN to the varying granularity of computation and storage, supported by the hardware.

To address these challenges, one option is to take a software-centric approach [69, 19] by restricting architectures to a standardized ISA that makes the compiler reusable. However, this approach limits the architectural innovations, offering orders of magnitude benefits through novel, specialized execution semantics. Another option is to take a hardware-centric approach [106, 101] that demands re-implementing new compiler stacks and optimization infrastructure for each accelerator.

Alternatively, this paper takes on these challenges and sets out to simultaneously enable the reuse of the compiler while reducing constraints on the architecture. To achieve these conflicting objectives, we propose a compilation framework that integrates a novel architecture abstraction, dubbed the Architecture Covenant Graph (ACG), in its workflow. Traditional ISAs focus on what fine-grained instructions an architecture can perform, which typically operate with a register file and an opaque caching system. In contrast, ACG is defined to capture accelerator structure as a graph consisting of compute units, on-chip/off-chip memory components, and interconnect; each of which contains operational capabilities as attributes.

To leverage this abstraction, we also devise the Codelets construct which is combined with the ACG to enable our Covenant compiler to target varying types of DNN accelerators. While the ACG abstracts the architecture, the Codelets represent the DNN operations and are gradually transformed into accelerator execution schedules by the Covenant compiler. Each Codelet represents DNN layers as sequences of operation on input variables to produce output variables. During compilation, Codelets are transformed into schedules by mapping operands to ACG memory locations, and assigning operations to ACG compute nodes capable of execution. Once operands and operations are mapped to ACG nodes, the dependence between operations and their operands is translated to explicit data transfer operations over the ACG interconnect.

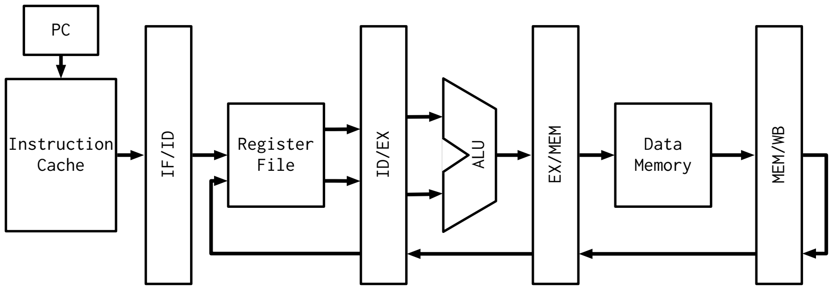
While a number of inspiring works have achieved multi-target compilation and scheduling support [19, 101, 6], the requirements for efficiently scheduling and generating code for new targets can be prohibitive. For scheduling to new targets, frameworks such as TVM [19] use flexible, target-agnostic scheduling directives to optimize DNN kernels, but each DNN operator schedule requires hand-tuning by architectural experts. As an alternative to manually scheduling, FlexTensor [123] and then Ansor [122] proposed novel search algorithms capable of identifying optimal schedules using stochastic search and performance measurements, but are inflexible to scheduling on new and unique architectures. Our approach provides the opportunity to adapt these scheduling techniques to new targets and further prune the space of transformations by coalescing architectural characteristics into the schedule. For code generation, both TVM as well as Glow [101] intentionally exclude architectural details because they rely on LLVM [69] as a backend, which is not designed for accelerators. Instead, we provide a malleable technique for code generation which is particularly important for architectures ordinarily using intrinsics which cause powerful instructions to be treated as black boxes by compilers. To support additional accelerators as compilation targets, these frameworks require creation of custom compiler backends and hand-tuned schedule templates.

Our Covenant compiler is intended for an orthogonal purpose: ***automatically*** *scheduling and generating code for accelerators without a unified, LLVM-like backend* *by integrating an architecture abstraction into the compiler. This is one of the main contributions of the work, in addition to the* ***ACG*** *and* ***Codelet*** *constructs which enable* ***Covenant*** *to target varying deep learning accelerators.*

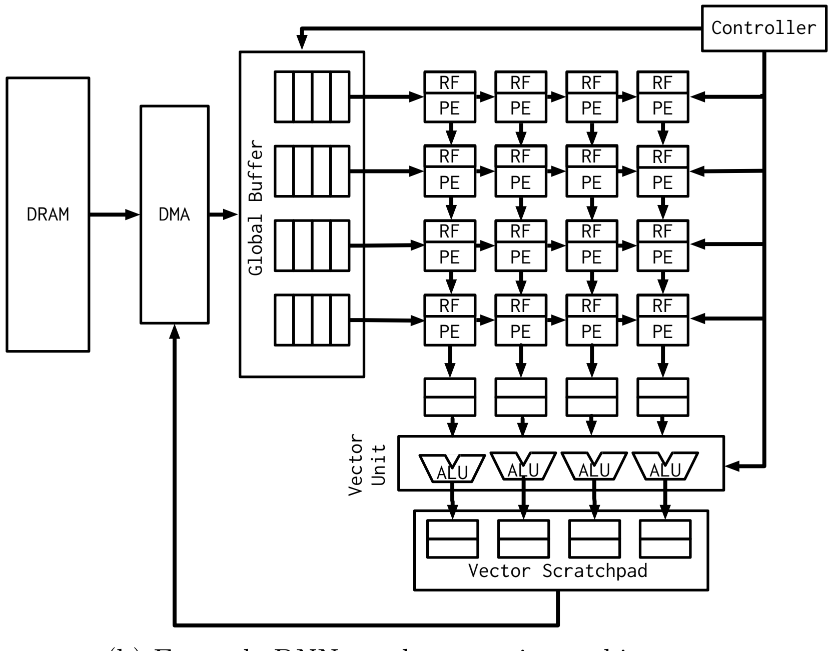
To demonstrate the flexibility of the Covenant compiler, we implement ACGs for Qualcomm© Hexagon™ HVX [1] DSP [24] and an open-source DNN accelerator [106]. For both architectures, we compile 14 different DNN layers across a combination of transformer networks, neural recommender systems, and convolutional neural networks and measure their performance. When targeting Hexagon, our automated approach achieves 93.8% of the performance of TVM's hand-scheduled templates that rely on manually constructed intrinsic. Compared to manually-implemented DNN layers in Qualcomm's nnlib which include hand-written assembly kernels, we achieve 31.3% improved performance. Besides Hexagon, we target an open-source DNN accelerator [106] that shows the flexibility of the Covenant compiler to target an entirely different architecture. The Covenant compiler achieves 182× performance improvement using the DNN accelerator compared to a CPU baseline. Finally, we illustrate the feasibility of implementing optimizations using the Covenant compiler by combining different optimization passes and achieve 128.6× speedup compared to unoptimized code on Hexagon. These results show the flexibility of the Covenant compiler for automating scheduling and code generation for accelerators while maintaining high-performance by integrating architecture characteristics through the ACG and Codelets.

3.2 The Missing Link: An Abstraction for Micro-Architecture Specification

General-purpose processors are based on the von Neumann model of computing, which is a sequential fine-grained instruction execution model. Hence, compilation for these processors is made possible by exposing the Instruction Set Architecture (ISA), through which the micro-architecture is completely abstracted away. However, rapidly emerging



(a) Pipeline processor microarchitecture.



(b) Example DNN accelerator microarchitecture.

Figure 3.1: Comparison of microarchitectures for general purpose processors and DNN accelerators.

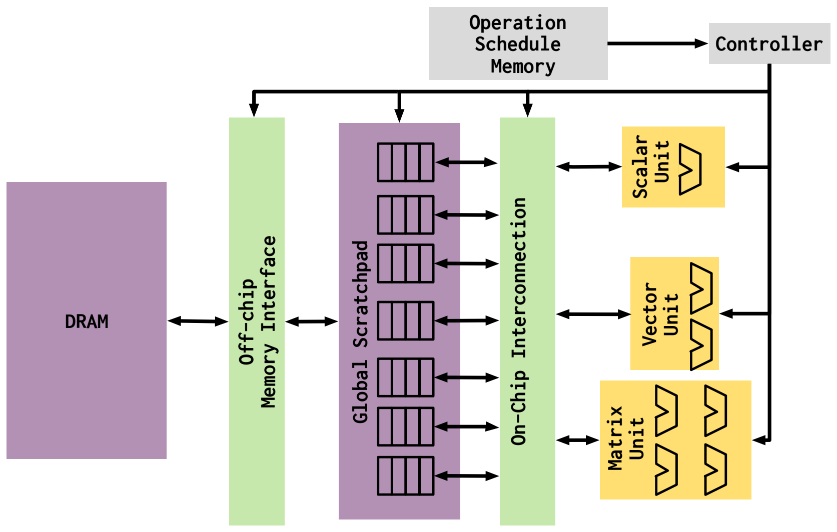
[1]Qualcomm Hexagon HVX is a product of Qualcomm Technologies, Inc. and/or its subsidiaries.

DNN accelerators tend to use other models of computing, such as systolic in the case of Google's TPU [57, 88] and dataflow in the case of Microsoft's Brainwave [23, 35]. These DNN accelerators typically consist of one or more arrays of Processing Elements, that can only perform simple arithmetic operations in parallel, as shown by the example in Figure 3.1b. Typically these PEs are connected to one another as well as on-chip memory through software-managed interconnection and memory hierarchy. As such, compilation for these novel architectures requires exposing more of the microarchitectural details. In contrast, general-purpose processors use a pipeline to enable a number of ALUs to carry out instructions, as illustrated in Figure 3.1a. They are also connected to the memory through a hardware-managed cache. The fundamental differences in the compute model and the organization of the architecture and microarchitecture between DNN accelerators and general-purpose processor clearly demonstrates the need for a new abstraction for compilation. However, exposing every detail makes compiler design an adhoc practice for each specific microarchitecture that is not reusable. Instead, DNN accelerator abstractions are required to enable a reusable compilation workflow for different types of DNN accelerator microarchitecture. The following section details such an abstraction, called the Architecture Covenant Graph (ACG).

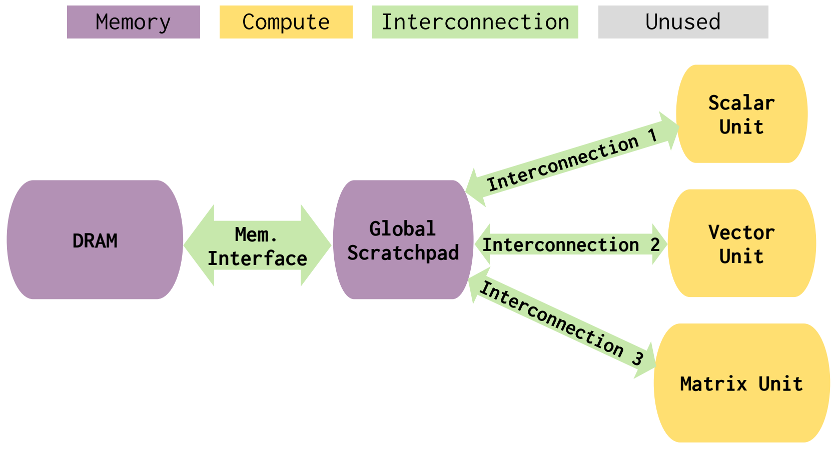
3.2.1 Architecture Covenant Graphs

We describe ACG and it's design rationale by using a running example of a generic DNN accelerator microarchitecture and the corresponding ACG in Figure 3.2. Figure 3.2a visualizes the microarchitecture for an example DNN accelerator, including its off-chip memory and software-managed, on-chip memory in purple, programmable interconnection in green, and three functional units with unique capabilities in yellow.

To capture the data movement properties on DNN accelerator microarchitectures such as these with programmable interconnection and different types of functional unit for



(a) Example DNN accelerator architecture.



(b) Architecture Covenant Graph (ACG) example.

Figure 3.2: Example DNN accelerator architecture and its ACG.

mapping operations to, the ACG is modeled as directed graph as shown in Figure 3.2b. Each ACG is comprised of vertices representing *programmable memory* and compute components, and unidirectional or bidirectional edges connecting each component. The edges represent the programmable interconnection between the on-chip/off-chip memory and compute components. Edge direction is required for enabling a reusable compilation workflow, as it informs the scheduler of valid paths for moving data, such as `DRAM` to `Global Scratchpad`, and `Global Scratchpad` to one of the functional units in Figure 3.2a. In this example, for each of the interconnections, data can be read and written to and from each of the functional units (`Scalar Unit`, `Vector Unit`, `Matrix Unit`) and the `Global Scratchpad`, as well as between `DRAM` and the `Global Scratchpad`. In some other cases multiple on-chip scratchpads are used for different purposes, with some scratchpads being restricted to sending data to functional units and unable to receive data, in which case the edge between them would be unidirectional. This is unlike traditional memory and caches in general-purpose processors, which are passive and generally do not execute instructions to send or receive data. Instead, the processor core is the active party that loads or stores data to these passive structures. In contrast, the compiler for a DNN accelerator often needs to generate instructions for memory components since they are active elements. Figure 3.2a also includes three separate programmable functional units capable of executing separate operations in parallel: a `Matrix Unit`, a `Vector Unit`, and a `Scalar Unit`. By using a directed graph, the compiler is capable of identifying opportunities for parallelizing operations across multiple functional units by selecting graph nodes which support the operation and have a common memory node predecessor.

However, scheduling the data movement also requires validation that the size of data being transferred is able to fit on the intermediate storage nodes such as `Global Scratchpad` in Figure 3.2a because there is no hardware-controlled data caching mechanisms. To distinguish between the attributes necessary for computation versus memory, ACG uses compute nodes shown in yellow and memory nodes shown in purple, each of which have distinct sets of attributes for informing the compilation process. In addition, lower-level architecture components shown in gray in Figure 3.2a such as the `Controller` for sending control signals to other components and `Operation Schedule Memory` for storing operations are not included in the ACG. *With the primary goal being machine code generation, the ACG excludes components such as these and other low-level details because they are not programmable, and do not provide relevant information to the compiler.*

Lastly, the unique properties across different DNN accelerator microarchitectures and even across their functional units binds them closely to the binary codes necessary for execution. As an example, the `Matrix Unit` in Figure 3.2a uses dataflow execution to perform matrix multiplication, only requiring data availability from the scratchpad to execute instead of relying on an explicit matrix multiplication binary code. In addition, making data available may require a sequence of binary codes for separately sending each input data to the functional unit rather than a single, dedicated code. Therefore, the ACG specifies binary code for a DNN accelerator as mnemonics without tying them to a specific computation model or set of execution semantics. This allows the code generation implementation to be reused across different architectures by because sequences of mnemonics can be defined for a finite set of operations which are delineated by the ACG nodes and edges.

Below, the specification used for mnemonics is detailed, in addition to the different attributes of compute nodes, memory nodes, and edges, included in the ACG.

Memory

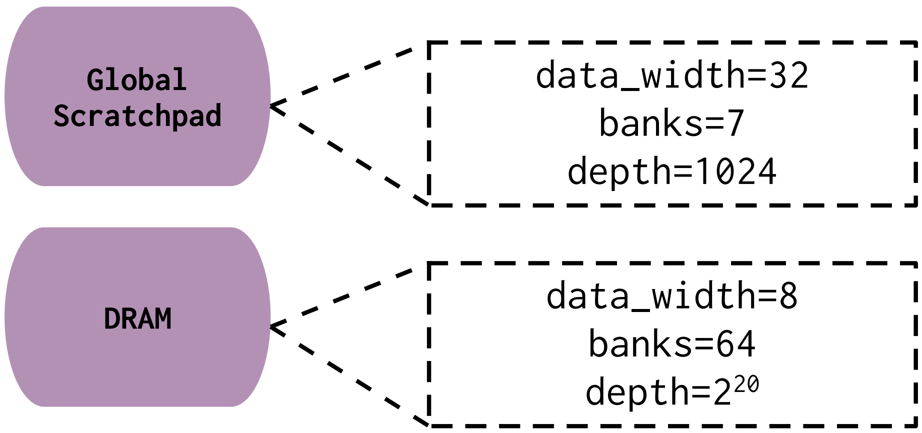


Figure 3.3: ACG storage nodes and their capabilities.

Software-controlled memory such as `Global Scratchpad` in Figure 3.3 allows the compiler greater control over data reuse, but also require explicit mnemonics for operations such as off-chip data transfers. To ensure valid memory accesses during execution, the access semantics and capacity of the memory needs to be known to the compiler so that memory request addresses are properly aligned. As shown in Figure 3.3, each `memory` node includes attributes defining their access semantics, such as the `data_width` for specifying the smallest unit of accessible data in bits is 32. The `data_width` is particularly important for DNN accelerators supporting mixed precision operations, because certain functional units might support 16-bit operations but read data from a memory component storing each 16-bit operand with a 32-bit `data_width`. In this case, the compiler must ensure that 16-bit operands are stored in 32-bit chunks rather than packed together, and the increased memory consumption for the operation is accounted for.

In addition, `memory` nodes use the `banks` attribute to denote the number of banks in a memory component, as it is common for on-chip memory to include varying number of banks for reading and writing multiple data in parallel to/from coarse grained functional units such as the `Vector Unit` or `Matrix Unit` shown in Figure 3.2b. Each bank is capable of sending `data_width` bits of data at a time, which means `data_width` $\times$ `banks` determines the size of an addressable element in the memory component. When selecting the sizes of on-chip data to be stored and operated on, the compiler must use this information to ensure the size is correctly aligned in memory by requiring data chunks are divisible by the size of an addressable element. As an example, the `Global Scratchpad` has $32 \times 7 = 224$ bit entries, which must be taken into account when generating mnemonics requiring address calculation based on immediate values.

Finally, compilers can exploit large on-chip scratchpads for data reuse by partitioning operands into chunks called tiles which are stored on-chip and operated on together. To validate tile selection, the compiler must ensure all being stored at once is within the capacity of the on-chip memory being used. For the `Global Scratchpad`, the capacity can be calculated by multiplying the `depth` attribute by the addressable element size: $224 \times 1024 = 229,376$ bits, or 28,672 bytes.

Interconnection

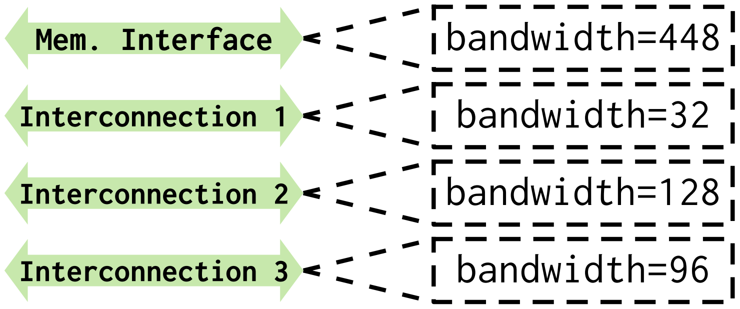


Figure 3.4: ACG interconnection examples

When it comes to generating code for transferring data on and off a DNN accelerator, a single binary code is often insufficient due to the limitations imposed by the interconnection between on and off-chip memory. For instance, `DRAM` in Figure 3.2a is connected to `Global Scratchpad` through a bidirectional `Off-Chip Memory Interface` interconnection. This link constrains the amount of data in bits transferred at a time, or may allow for more than one unit of `Global Scratchpad` data to be moved in a given cycle. In the running example, a directed edge called `Mem. Interface` represents these types of interconnection which represent the supported programmable communication capabilities. The directed ACG edges use the `bandwidth` attribute to define the amount of data in bits capable of being transmitted in a single operation as shown in Figure 3.4. This information is crucial during compilation, as DNN accelerators provide more flexible data transfer capabilities allowing variable-sized data transfers between on and off-chip memory. Furthermore, the bandwidth determines the number of memory requests the compiler needs to generate for this specific edge to load a tile of data.

In addition, the `Interconnection` is capable of sending data to multiple parallel programmable functional units, with unique data processing properties, therefore requiring different bandwidths. To distinguish between the different data transmission properties between a single interconnection and different DNN accelerator components, the ACG includes several `Interconnection` edges with unique bandwidths.

This is particularly important when making scheduling decisions, because a coarse-grained operation could be mapped to multiple parallel functional units with hardware-controlled synchronization, but the interconnection between on-chip storage and certain functional units may require multiple data transfer operations for sending the necessary operand data.

Compute

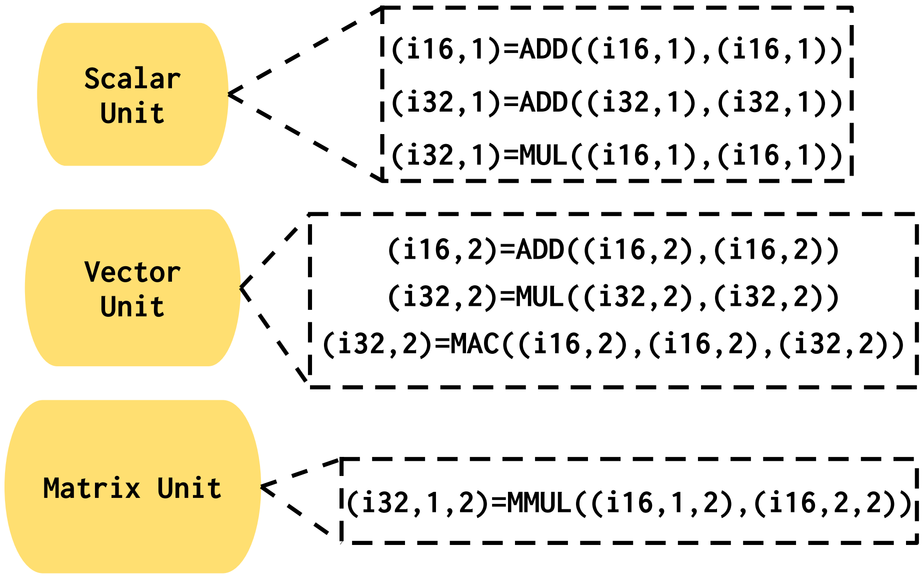

Scalar Unit
(i16,1)=ADD((i16,1),(i16,1))
(i32,1)=ADD((i32,1),(i32,1))
(i32,1)=MUL((i16,1),(i16,1))
Vector Unit
(i16,2)=ADD((i16,2),(i16,2))
(i32,2)=MUL((i32,2),(i32,2))
(i32,2)=MAC((i16,2),(i16,2),(i32,2))
Matrix Unit
(i32,1,2)=MMUL((i16,1,2),(i16,2,2))

Figure 3.5: ACG compute nodes and their capabilities.

DNN accelerators provide unique opportunities for mapping coarse-grained operations to a variety of compute resources, as shown in Figure 3.2a, which includes a 2×2 `Matrix Unit`, 2-wide `Vector Unit`, and `Scalar Unit`. The ACG represents programmable functional units as compute nodes, using an attribute called `capabilities` to describe the coarse-grained functionality supported by the corresponding architecture component. Figure 3.5 demonstrates capabilities for each compute in Figure 3.2b, with each compute node supporting varying granularity, datatype, and number of operations. Capabilities encapsulate opportunities for parallelism and type-specific operations in the compute nodes. They are defined by an operation name and an ordered list of datatype and element size pairs for each input/output operand associated with the operation. A subset of the supported operations are defined in Table 3.1. For example, the `Vector Unit` supports the `ADD` operation, taking two input operands with two, 16-bit integer elements and generates two 16-bit integer output elements. The sizes and datatypes are included in the operand specification because the specialized compute units in DNN accelerators are capable of performing different operations in parallel on varying kinds of operand datatypes and sizes. By defining capabilities this way, the compiler can identify which functional units can

Table 3.1: Subset of supported capabilities and their definitions.

Type	Name	Description
Unary	`RELU`	Rectified Linear Unit function.
	`SIGMOID`	Logistic sigmoid.
	`TANH`	Hyperbolic tangent function.
Binary	`ADD/SUB`	Element-wise addition and subtraction.
	`MUL/DIV`	Element-wise multiplication and division.
	`MAX/MIN`	Element-wise maximum/minimum.
	`MMUL`	Matrix-matrix multiplication.
Ternary	`MAC`	Multiply-accumulate.
	`GEMM`	General Matrix Multiply.

execute parts of the DNN layer in parallel by matching the operation name and data type to the functional unit capability, and then breaking the coarse grained DNN operation into the same size chunks. To demonstrate this, consider an element-wise addition operation specified as: `(i16,3)=ADD((i16,3),(i16,3))`. The compiler can decompose this operation into a scalar addition on the `Scalar Unit` and a vector addition on the `Vector Unit`, as both compute nodes support 16-bit integer addition at different granularities. To ensure the full range of layer mappings are exposed to the compiler, capabilities defined for a compute do not require one-to-one mappings between capability primitive and a functional unit's mnemonic. As an example, the `Vector Unit` might not directly support a multiply-accumulate (`MAC`) operation using a single mnemonic, but it can be defined as a capability by breaking it into separate multiply-add mnemonics.

Mnemonics

Thus far, the ACG has described the structure and programmability of a DNN accelerator, but the mnemonics which can be composed to carry out the data movement and operations represented in the ACG must also be defined to generate executable binaries. In contrast to general-purpose processors which use instructions and assume a von Neumann compute model, different DNN accelerators depend on different compute models with unique machine code semantics. Thus, machine codes for a DNN accelerator are defined as mnemonics stored as an ACG attribute for generating sequences of mnemonic code. Each individual mnemonic is defined with customizeable attributes for analysis/optimization, and an ordered list of named fields with fixed bitwidth, which can represent either a constant number or an enumerated set of values. As an example, a mnemonic with the `ADD` id is defined above and includes 4 fields, where `src1,src2` and `dst` are constant fields representing the starting addresses in scratchpad, and `target` is an enumerated value field which can be set to one of `SCALAR` or `VECTOR` depending on the functional unit to be executed on. By generically defining mnemonics in this manner, they can be used for different types of DNN accelerators without binding the mnemonics to certain execution semantics.

3.3 Codelets

To flexibly enable DNN compilation to domain-specific architectures, a programming abstraction must capture both the semantics of an operation, and the relevant microarchitecture components it is tied to. In addition, a construct for enumerating the different types of macro-mnemonics required for code generation must be designed. Covenant uses compute kernel abstractions called Codelets which are complimentary to the ACG to enable compilation. Codelets are defined prior to compilation as a sequence of operations on

$$mnemonic \in Mnemonic ::= \textbf{mnemonic}\ name(opcode)\{field^*, attr^*\}$$

(a) Mnemonic definition syntax.

```
# Example: ADD #3,#0,#1, VECTOR
mnemonic ADD(3) {
  ifield("SRC1_ADDR",8),
  ifield("SRC2_ADDR",8),
  ifield("DST_ADDR",8),
  efield("TGT", 1, ["SCALAR","VECTOR"])
}
```

(b) Example of `ADD` mnemonic definition.

Figure 3.6: Example of a mnemonic definition.

parametric-shaped operands called *surrogates* which represent DNN layers. Initially, the operations do not include architecture-specific details, which enables their portability across different architectures. However, during Covenant compilation each Codelet is gradually transformed to define the sequence and mapping of operations based on an ACG. Codelets are declared using a DNN layer name, and are composed of `compute`, `transfer`, and `loop` operations which represent operations on tensors, movement of data, and repetition of operations. As an example, an add Codelet can be defined as shown in Figure 3.7a. To integrate ACG information into the compiler, Codelet operations rely on different types of surrogate variables to encompass both data attributes (e.g., datatype, shape) and ACG location throughout execution.

3.3.1 Surrogate Variables

The process for generating valid sequences and mappings of operations on data is inherently tied to accelerator attributes. As such, surrogate variables in Codelets encode shape information, datatype and ACG location:

```
x=inp([dim1,...,dimN],dtype,loc);
```

Data movement is tracked by requiring surrogate variables to be associated with a single

```
cdlt add {
  N=param();
  a=inp([N],null,null);
  b=inp([N],null,null);
  c=out([N],null,null);
  loop n(N) {
    c[n]=compute(null,"ADD",a[n],b[n]);
  }
}
```

(a) Initial Codelet.

```
cdlt add {
  # Size and datatype are set
  a=inp([12],i16,null);
  b=inp([12],i16,null);
  c=out([12],i16,null);
  loop n(12) { # Number of iterations is set
    c[n]=compute(null,"ADD",a[n],b[n]);
  }
}
```

(b) Codelet mapped to a DNN layer.

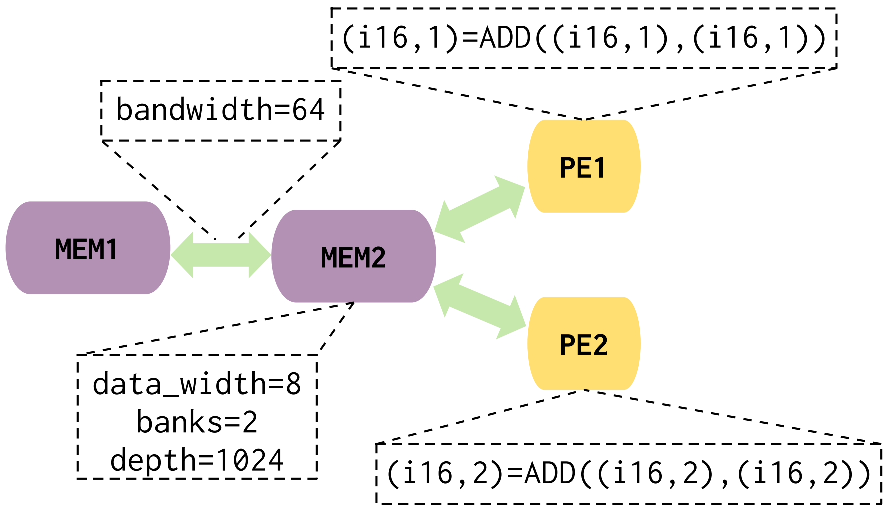
Figure 3.7: Example of a add Codelet.

ACG location, defined by the `loc` attribute. Using single location surrogates has the added benefit of distinguishing between a DNN layer input and a tile generated from operand data because they will each be represented by different variables with a similarly different shape and layout in memory. To further simplify Codelet compilation, different types of surrogate variable with unique semantics are used. For instance, Codelets are defined with the basic assumption that it will receive a certain number of inputs and generate outputs, defined as `inp` and `out` type surrogates. In addition, each DNN layer performs similar sets of operations on different shaped tensors, and sometimes use parameters to define how the layer is executed, both of which are represented as `param` surrogates. Prior to compilation, other unset fields such as the location and datatype are set to `null` to indicate they have not been assigned. When a `relu` layer is mapped to a Codelet, each `param` surrogate is replaced with with the corresponding layer-specific value, which results in known input/output sizes and operation sizes as shown in Figure 3.7b The datatype is also set during the layer mapping, and is assumed to be provided in the DNN layer specification.

Once a Codelet has been mapped to a DNN layer instance, the Covenant compiler applies ACG attributes to transform the Codelet. To begin, the compiler assumes that `inp` and `out` surrogates are stored on the highest level of the ACG memory hierarchy, identified as the memory node with the longest path to each functional unit. Once the operand surrogates are mapped, computations are mapped to compute nodes in the ACG and data movement operation to and from the target compute node are added. The last surrogate type, `local`s, represent data stored on the intermediate nodes on the path from an `inp` location to compute node location or compute node location to an `out` location. Each `local` surrogate is created as a result of `transfer` and `compute` operations, and their attributes are inferred based on the source operation as we will discuss below.

3.3.2 Codelet Operations

DNN accelerators provide diverse sets of programmable compute and memory resources for enabling more parallelism when it comes to computation and additional opportunity for data reuse due to programmable memory. In contrast to von Neumann architectures which use uniform memory accesses and sequential computation models, scheduling for DNN accelerators adds an extra layer of difficulty by having to keep track of where data is stored and where computation is occurring. Codelets address these complexities using three categories of operations to represent DNN operations: `loop`, `transfer` and `compute`. Each operation operations type has a fixed set of attributes which determine layer-specific and architecture-specific properties required for scheduling and code generation. To demonstrate how the Covenant compiler uses these constructs to handle the added complexity we will reference the example shown in Figure 3.8. In the example, the add Codelet shown in Figure 3.8b is targeting the ACG in Figure 3.8a, and



(a) Example ACG use for compilation.

```
cdlt add {
  a=inp([12],i16,"MEM1");
  b=inp([12],i16,"MEM1");
  c=out([12],i16,"MEM1");
  c1=transfer(i16(0),"MEM2",[2]);
  loop n(0,6,2) {
   # Using only 25% of bandwidth!
   a1=transfer(a[n],"MEM2",[2]);
    # Much more available memory!
   b1=transfer(b[n],"MEM2",[2]);
   # Unused PE!
   c1[0]=compute("PE2","ADD",a1,b1);
   transfer(c1,c[n],[2]);
  }
}
```

(b) Codelet prior to being fully scheduled.

```
cdlt add {
  a=inp([12],i16,"MEM1");
  b=inp([12],i16,"MEM1");
  c=out([12],i16,"MEM1");
  c1=transfer(i16(0),"MEM2",[6]);
  loop n(2,stride=6) {
    # Tile loops, maximum bandwidth use
   a1=transfer(a[n],"MEM2",[6]);
   b1=transfer(b[n],"MEM2",[6]);
   loop n1(2,stride=3) {
     # Split operations between PE1 and PE2, in parallel
     c1[n1]=compute("PE2","ADD",a1[n1],b1[n1]);
     c1[n1+2]=compute("PE1","ADD",a1[n1+2],b1[n1+2]);
   }
    transfer(c1,c[n],[6]);
  }
}
```

(c) Scheduled Codelet

Figure 3.8: Example Codelet scheduling using an ACG.

the the starting location of inp and out surrogates is set to MEM1.

Compute operations: To accommodate the variation in operations supported by different compute units, the compute operations are defined using the coarse-grained capabilities described in Section 3.2.1, and operate on tensor operands. Every compute operation has is defined with the ACG compute node it is mapped to, it's capability name, and the surrogate operands and their offsets:

```
c[i]=compute(loc,capability, op1[off1], op2[off2],...,opN[off3]);
```

To specify tensor offsets, compute operands can be indexed using loop operations, which can be converted to address offsets for programmable memory by combining the size of the surrogate and the addressing information for it's ACG location.

The ADD compute operation in Figure 3.8b can be mapped to either PE1 or PE2, as both include the supported capability but with different granularities. The compiler automatically determines the mapping by selecting the ACG node capable of performing the most operations at a time, PE2 in this case, because it can do two element-wise additions at a time. Once selected, the Covenant compiler updates the location field in the compute operation with the target compute node.

Transfer operations After mapping each compute operations in the Codelet, the compiler orchestrates data movement across programmable memory by adding explicit transfer operation to the Codelet. transfer operations are used in Codelets to represent data movements across a DNN accelerator, explicitly codifying scheduling of data locations as required by domain-specific compilers. In Figure 3.8b, this can be accomplished by first finding the shortest path between MEM1 and PE2 and adding transfer operations for each operand and each edge. transfer operations are specified with a source, destination, and the transfer size in number of source elements in each dimension of the source operation. The semantics of a transfer operation can differ depending on the type of source and destination used, which accommodates the different operations required by programmable memory:

```
dst=transfer(src[i],"MEM1", [n]); # Move data to MEM1
dst=transfer(i16(0),"MEM1", [n]); # Allocate new memory at MEM1
transfer(src[i],dst[i], [n]); # Overwrite data stored in dst
```

In the first two examples, the destination is specified by an ACG node name, which tells the compiler that memory needs to be allocated at that location and a new `local` surrogate is generated. When the source of allocation transfers is an operand with an index offset, the compiler will generate a `local` surrogate with `n` elements as its size, the same datatype as the `src`, at `MEM1`. Alternatively, new memory can be allocated for reuse if the source operand is a constant value which includes its type and size. In cases where the destination is an operand such as `dst`, memory at `dst` location will be overwritten and no additional surrogates will be generated.

To determine the needed transfer type for Figure 3.8b, the Covenant compiler adds `transfer` operations for each of the edges between the source operand location and compute target. Specifically, the compiler generates new memory allocation operations necessary for storing the outputs of `ADD` on `MEM1` as shown in Figure 3.8c. The compiler also generates `transfer` operations for each `inp` to the intermediate memory nodes. Lastly, the compiler must send the on-chip results stored in `MEM2` to `MEM1`, which will write data to the location of `c`. Once inserted, each `transfer` operation can be combined to calculate the cumulative size of the data for each memory node at different points in the Codelet by tracking the `transfer` operations and their sizes, as well as operand datatypes.

Loop operations: Lastly, to represent repeated operations, addressing offsets, and operations execution amounts, `loop` operations with the same semantic meaning as “for” loops are used. To satisfy the need for operand address offsets, `loop` operations can be used as indices for operands in both `transfer` and `compute` operations, which is commonly used in DNN operation descriptions to specify which tensor elements are being operated on. Each `loop` operation is used for specifying DNN layer semantics, and therefore does not include attributes relating to the ACG. `loop` operations are created using a variable name, lower and upper iteration bounds, the stride, and opening a scope for execution using curly braces:

```
loop i(0,6,2) { ... } # Iterate from 0 to 6, stride=2
```

To use `loops` as indices for surrogates, `loop` name is put inside brackets alongside a surrogate to represent an address offset. A key component of compilation is tiling, and `loops` offer a familiar construct to apply tiling transformations, as loop splitting is a commonly technique for tiling on general purpose processors. When tiling a Codelet, `loops` are split into groups according to the number of transfers required to send data from it's source to the compute destination. Splitting a `loop` operations consists of factoring the number of iterations into an outer `loop` operations with a step size corresponding to how large a tile will be, and an inner `loop` operations which has a range equivalent to the outer loops step size.

3.3.3 Macro-Mnemonics

For DNN accelerators, generating valid mnemonics is conditioned on which functional unit is being used because the same operation can generate different mnemonics depending on the compute unit. The Covenant compiler ensures valid code generation by combining operations types, operand types, and their ACG node attributes to select pre-defined functions for generating sequences of mnemonics called macro-mnemonics. These macro-mnemonics use the Codelet operation type it is matched with, the ACG node(s) it is associated with, and the containing Codelet as contextual input to define mnemonics generation. Each mnemonic is generated by populating it's fields with either statically determined values or by using attributes of Codelet operations.

3.4 Enabling Optimization

Compiler optimizations for DNNs have been shown to enable significant performance improvements when targeting CPU and GPU [10, 122]. However, state-of-the-art, stochastic optimization techniques which rely on performance measurements to guide the algorithm cannot be applied to domain-specific architectures without the ability to generate executable code. When targeting domain-specific architectures, optimizations have the potential to offer even greater benefits due to their tendency to provide more compute and memory resources with greater programmability.

The Covenant compiler is intended to be a community driven project which improves as a crowd-sourced effort. Therefore, the initial goal is to provide a framework which *enables* new and existing optimization algorithms to be constructed and benefit from the use of the ACG rather than introducing new optimizations. Below, we discuss how existing optimizations can be transformed by integrating architectural details into the algorithm.

Algorithm 3: Codelet Tiling Validation

```
function ValidTiling(codelet, ACG)
    let $V \leftarrow \emptyset$ // Valid tilings
    let $f_i$ =loop iteration factors for $\texttt{loop}_i \in codelet$
    let $P$ =factor permutations $\in f_i$
    for each $p \in P$ do
        let $constraint\_sat = True$
        // Keep track of data stored on each ACG storage node
        let $storage[s] = 0$  for each storage node $s \in ACG$
        for each $t$ =transfer $\in codelet$ do
            let $p\_t = \{factor \in p | factor \in t.offsets\}$
            let $xfer\_size = t.operand.dtype.bits \times \Pi(p\_t)$
            $storage[t.dst] += xfer\_size$ // Update $t.dst$ storage
            if $(xfer\_size \mod t.src.data\_width) \neq 0$ or $storage[t.dst] > t.dst.capacity$
              then
                $constraint\_sat = False$
                break
        if $constraint\_sat == True$ then
            $V = V + p$
    return $V$
```

Codelet optimization passes are defined as functions which take an individual Codelet and the ACG as arguments, and return the transformed Codelet. Providing the

ACG as an argument allows for retrieval of certain characteristics embedded in the ACG because Codelet operations only contain the ACG node names as attributes. The attributes embedded in ACG nodes bolster common optimizations used in traditional compilers which might otherwise be applied using a heuristic.

Tiling Validation is one example of commonly used optimizations is loop tiling, where loops are grouped into smaller blocks of operations on tiles of data to increase the data locality as previously discussed. In contrast to other frameworks, tiling is built into Covenant scheduling algorithm rather than being an optimization pass, although further optimization of tile selection can be implemented as a Codelet optimization. Here, we show how tiling validation is performed in the Covenant compiler, and can be extended to search-based optimization passes. When tiling operations for targeting general purpose processors, loop ranges can be split using almost any permutation of numbers which are factors of the loop iterations because memory is typically hardware-controlled which prevents invalid memory requests. In contrast, domain-specific architectures often provide programmable memory where certain tiling permutations will lead to invalid programs instead of slower programs. As shown in Figure 3, Covenant validates tiling by first collecting all valid factors of the Codelet `loop` ranges in f_i, and then generating all unique combinations of those factors in P. The first concern for tile validation is sufficient memory space to store each tile, which is a map of memory ACG nodes to data sizes,$storage$, is initialized to 0 to track total storage for the permutation p. Each factor in p represents a possible stride for a `loop` operation, and each `transfer` operation uses `loop` operation as index offsets. This allows the `transfer` size to be computed using the product of `loop` strides and the datatype size of the `transfer` operand, $t.operand.dtype.bits$. Once the $transfer\_size$ is computed, it is added to the destination memory node in t. The tiling can be validated by first checking if the $transfer\_size$ is divisible by the data width of the source memory node to ensure addressability, and then the capacity of destination memory node is verified.

If these constraints are satisfied, the tiling permutation is validated and is added to a set of possible tilings for final scheduling.

Loop Unrolling Loop unrolling is another common optimization, used to reduce the impact of loop branching as well as memory overheads by transferring more data in a loop body and unrolling computations for the transferred data. Using the ACG, opportunities for loop unrolling can be identified by iterating over `transfer` operations, and checking the bandwidth of the edge connecting the source and destination ACG nodes. If the `transfer` size is less than the edge bandwidth, more data can be transferred in a single operation if the destination ACG node does not reach maximum capacity.

Parallelization A central focus of domain-specific architectures for DNNs is providing as many opportunities for parallelism as possible. Taking advantage of the parallelism in such architectures is not always trivial, especially when heterogeneous compute cores are available with varying capabilities. However, the ACG simplifies parallelism identification through the capability attributes in compute nodes, which can be combined to form the equivalent operation and therefore be performed in parallel. As an example, Figure 3.9a demonstrates an ReLU operation on two 25-element tensors targeting an ACG composed of two compute nodes: a "SIMD" capable of performing four ReLU operations at a time, and a processing engine ("PE") capable of a scalar ReLU. The two tensors do not factor perfectly into the SIMD, which demonstrates a common difficulty when trying to identify parallelization. One solution to this problem is to introduce additional operations which pad zeros to each of the tensors so that they can be tiled correctly. Instead, the ACG can be used to identify other compute units, namely "PE", capable of being combined with the SIMD to form tiles of parallel operations.

Mnemonic Packing For micro-architectures using Very Long Instruction Words (VLIW), multiple instructions can be performed in parallel by "packing" them together. In these architectures, compiler needs to identify independent instructions and pack them to increase

```
cdlt relu {
  a = inp("DRAM",[25],i32);
  c = inp("DRAM", [25], i32);
  loop i(25) {
    c[i]=compute("PE","RELU",a[i]);
  }
}
```

(a) Pre-scheduled operations

```
cdlt relu {
  a = inp("DRAM",[25],i32);
  c = out("DRAM", [25], i32);
  loop i(25,stride=5) {
    # SIMD: RELU((i32,4),(i32,4)), PE: RELU((i32,1),(i32,1))
    c[i]=compute("SIMD","RELU",a[i]);
    c[i+4]=compute("PE","RELU",a[i+4]);
  }
}
```

(b) Parallelized operations

Figure 3.9: Parallelization Identification Using an ACG.

utilization. With Codelet operation being coarsely defined to represent multiple mnemonics, forming mnemonic packets is performed during code generation as an optimization. To form mnemonic packets, ACG resource availability as well as mnemonic dependencies need to be identified. To enable packing, the ACG node executing each mnemonic is identified to determine the resources consumed by a VLIW packet and integrated into the packing algorithm. For dependency analysis, the `field` attributes in mnemonics can be annotated with read and write semantics to identify sequences of independent mnemonics. Using both of these mnemonic attributes allows packet formation by iterating over `mnemonics` for a Codelet and creating a packet with a single mnemonic occupying the `tgt` resource. Then, independent mnemonics capable of execution in the current packet, determined by the consumed ACG resources and available VLIW slots, can be hoisted into the current packet.

Table 3.2: DNN Layer Benchmarks.

Model	Layer	N	IH/ IW	OH/ OW	KH/ KW	IC/ OC	# Heads
BERT-LG	GEMM1	384	1	1	1	1024/4096	-
	GEMM2	384	1	1	1	4096/1024	-
	ATN1-GEMM	384	1	1	1	1024/64	16
	ATN2-GEMM	384	1	1	1	64/384	16
	ATN3-GEMM	384	1	1	1	384/64	16
	ATN4-GEMM	384	1	1	1	1024/1024	1
DLRM	FC1	1	1	1	1	745/367	-
	FC2	1	1	1	1	367/512	-
	FC3	1	1	1	1	512/256	-
	FC4	1	1	1	1	256/1	-
InceptionV3	FC1	1	1	1	1	2048/1000	-
	CONV1	1	299	149	3	3/32	-
MobileNetV3	CONV1	1	224	112	3	3/16	-
	CONV2	1	112	112	3	16/64	-
ResNet50	FC1	1	1	1	1	512/1000	-
	CONV1	1	224	112	7	3/64	-
	CONV2	1	224	56	3	64/64	-

3.5 Evaluation

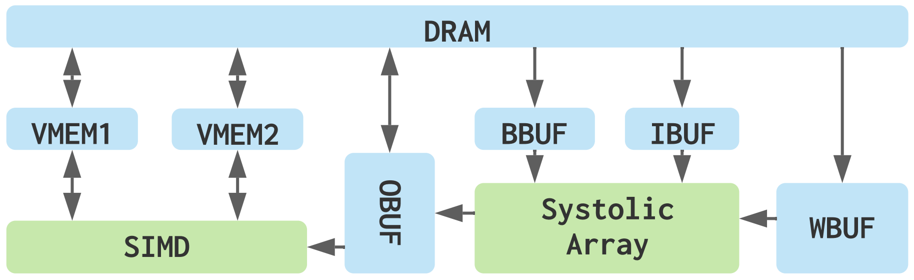
3.5.1 Experimental Setup

Benchmarks. To evaluate covenant, we use a comprehensive set of benchmarks from various classes of DNNs including image classification (InceptionV3 [113], ResNet-50 [48]), object detection (MobileNetV3 [52]), natural language processing (BERT-Large [32]), and neural recommendation systems (DLRM [85]). For image classification and object detection networks we choose convolutional and fully-connected layers that make up the majority of these networks. For BERT-Large, we benchmark the GEMM layers and the self-attention layer of an encoder block. Finally, for DLRM, we benchmark its Multi-Layer Perceptron (MLP) fully-connected layers. Table 3.2 lists all the DNN layer benchmarks with their layer dimensions. N shows the sequence length for language models and the batch size for other DNNs. IW/IH and OW/OH show the input/output width/height dimensions of the layers, while KW/KH parameters specify the weight kernel dimensions. Note that for FC/GEMM layers, these dimensions are equal to one. Finally, IC/OC column show the number of input/output channels for the DNN layers. We use INT8 precision for inputs/weights and INT32 precision for outputs of layers.

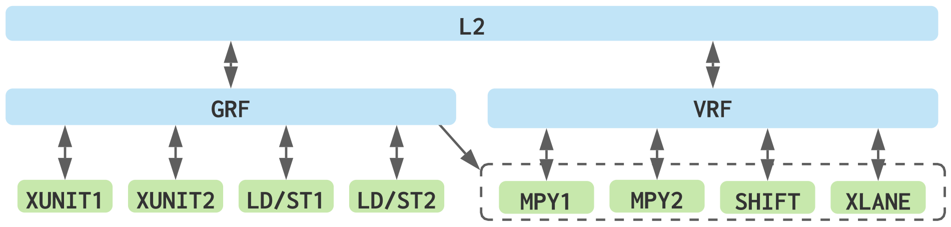
Target Architectures

To demonstrate the flexibility of Covenant for multi-target compilation, we use two distinct architectures: Qualcomm© Hexagon™ HVX [1] DSP [24] and an open-source DNN accelerator [106]. For each architecture, we use the ACG DSL for Covenant compilation.

DNNWeaver. DNNWeaver is a parameterizable DNN architecture which consists of two main compute components: (1) a systolic array connected to several on-chip buffers that is capable of executing various-sized convolution and GEMM layers, and (2) a SIMD vector processing array connected to two vector scratchpad memories that supports the remainder of layers (e.g. pooling, activation, normalization, etc.) As shown in Figure 3.10a, the systolic array is connected to four separate on-chip buffers by unidirectional edges, where it reads input activation data, model weights, and bias data from IBUF, WBUF, and BBUF buffers, respectively, and writes output to OBUF buffer. Additionally, the SIMD array is connected to OBUF with a unidirectional edge to consume its data, while is also connected



(a) DNNWeaver ACG



(b) Hexagon ACG.

Figure 3.10: Visualization of ACGs for DNNWeaver and Hexagon. Blue nodes are memory and green nodes are compute.

[1]Qualcomm Hexagon HVX is a product of Qualcomm Technologies, Inc. and/or its subsidiaries.

Table 3.3: A Subset DNNWeaver and Hexagon ACG Attributes.

Architecture	ACG Node	Example Attributes
DNNWeaver	Systolic Array	(i32,64)=GEMM((i8,64),(i8,64,64),(i32,64))
	SIMD	(i32,64)=ADD/SUB((i32,64),(i32,64))
		(i32,64)=SIGMOID/TANH((i32,64))
	VMEM1/2	data_width=32; banks=64; depth=2048
	IBUF	data_width=8; banks=64; depth=2048
	WBUF	data_width=8; banks=4096; depth=4096
	OBUF	data_width=32; banks=64; depth=2048
	BBUF	data_width=32; banks=64; depth=1024
	DRAM	data_width=8; banks=1; depth=32 billion
Hexagon	CORE	(u8,8)=ADD((u8,8),(u8,8))
		(i32,1)=ADD((i32,1),(i32,1))
		(i32,1)=MAC((u8,4),(u8,4),(i32,1))
		(i32,1)=MUL((i32,1),(i32,1))
	HVX	(i32,32)=ADD/SUB((i32,32),(i32,32))
		(i32,32)=MVMUL((u8,32,4),(u8,4))
		(i32,32)=GEMM((u8,32,4),(u8,4),(i32,32))
		(u32,32)=GEMM((u8,32,4),(u8,4),(u32,32))
	GRF	data_width=32;banks=4;depth=32
	VRF	data_width=1024;banks=32;depth=32
	L2	data_width=8;banks=32;depth=1024

with bidirectional edges to two scratchpad memories (VMEM1/2) to read/write vectors during computations. Table 3.3 lists a subset of attributes for DNNWeaver ACG nodes.

Hexagon. Qualcomm© Hexagon™ HVX [1] is a Digital Signal Processor (DSP) created by Qualcomm Technologies, which uses VLIW instructions and includes vector extensions. Figure 3.10b illustrates the ACG of HVX. As shown, HVX incorporates a scalar core that supports a diverse set of scalar instructions (Add, Mul, MAC, Max, etc.) and uses a General Register File (GRF) for operand read/write. In addition to the scalar core, Hexagon includes an additional SIMD processor for vector instructions, with 32 lanes each capable of performing a range of four 8-bit operations to a single 32-bit operation per lane, called Hexagon Vector Extensions (HVX). As opposed to DNNWeaver where all the data transactions between DRAM and on-chip buffers are governed explicitly by the instructions, Hexagon HVX is similar to typical general-purpose processors and incorporates hardware-managed caching mechanisms for loading/storing from/to DRAM, which is why DRAM is not included in the ACG.

Performance Measurements and Comparisons

Baseline frameworks. We compare the performance of our proposed Covenant compilation framework to two other frameworks: nnlib and TVM. For all three comparison points we use the Hexagon DSP as the target architecture and evaluate the performance of the compiled benchmark DNN layers. For benchmark baselines, we use optimized PyTorch [96] implementations on an Intel Xeon E7 CPU. nnlib [55] is a framework developed by Qualcomm for offloading DNN operations to Hexagon, comprising a set of hand-tuned C code and assembly kernels for DNN layers. TVM [19] is a compilation stack that supports a variety of general-purpose architectures as well as its own custom accelerator, VTA [81]. To compile to Hexagon DSP using TVM, we used hand-tuned schedules and manually defined intrinsics developed by Qualcomm experts, which generate optimized LLVM code for Hexagon.

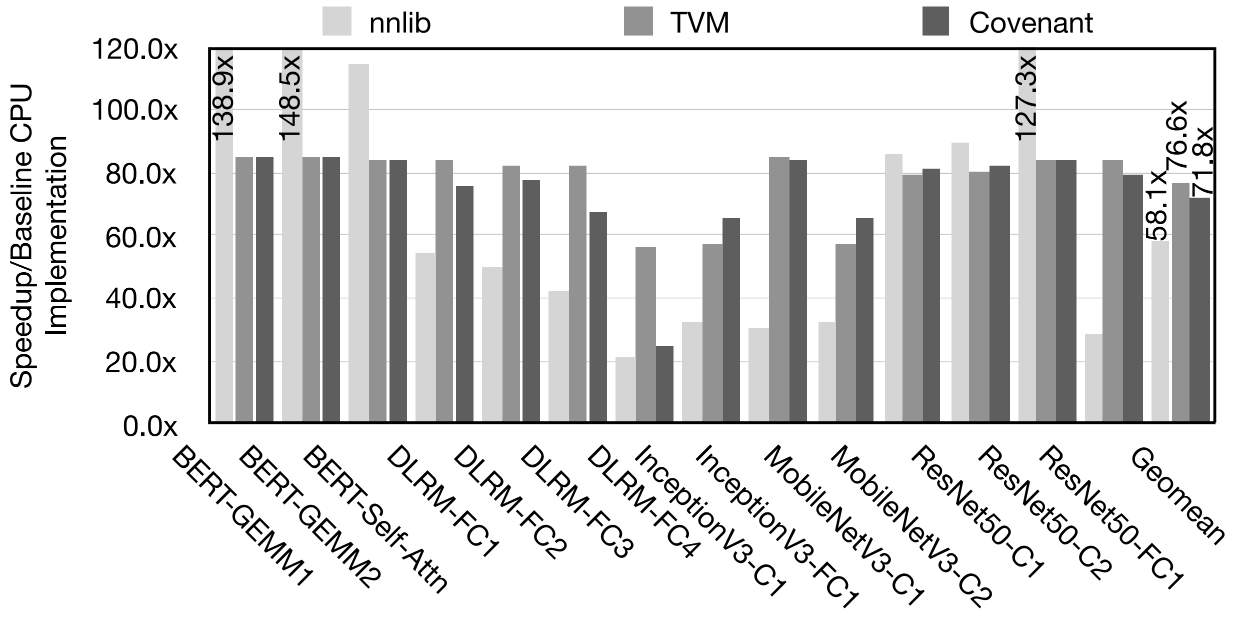


Figure 3.11: Performance comparison of various frameworks.

Performance measurements. To measure the performance of the codes compiled by Covenant, nnlib, and TVM targeting Hexagon DSP, we use the built-in cycle-accurate Hexagon SDK simulator developed by Qualcomm experts. To assure a fair comparison, we include the device execution time, which manifests the actual runtime of the DNN layers on the target hardware, for all the comparison points and use that without considering host execution overheads. To evaluate the capability of the Covenant framework in targeting multiple architectures, we also use DNNWeaver, an open-source DNN accelerator [106]. To measure the runtime of the Covenant compiled code on DNNWeaver, we used its open-sourced cycle-accurate simulator [2]. To verify the correctness of the compiled codes for all the frameworks and target architectures, we compare the outputs generated by the simulators with the software implementation of the DNN layers in PyTorch.

3.5.2 Results

Framework Comparison

Figure 3.11 shows the speedup enabled by the three compilation frameworks targeting Hexagon DSP, compared to a baseline CPU implementation. Across all benchmarks, Covenant provides an average of 31.3% improvement compared to nnlib's hand-tuned kernels. Covenant also achieves 93.8% of TVM's performance on average. As Figure 3.11 shows, all three frameworks perform better on larger layers having more operations. This results from the compounding parallelization optimizations across more loop iterations. Among all benchmarks, BERT-GEMM1 and BERT-GEMM2 layers see the maximum performance gains, as the larger number of computations in these layers provide highest code optimization opportunities. Relative to TVM, the DLRM-FC4 has a smaller speedup in Covenant because it includes a branch instruction for the single-iteration OC loop, whereas TVM generated code avoids this overhead. With regard to nnlib, the improvements are more significant for larger layers, due to inclusion of hand-tuned tensor transformations allowing more MAC operations per cycle. However, these transformations can be detrimental for smaller layers (e.g., DLRM) and convolutional layers where the total size of reduction dimensions is smaller because the transformations cannot maximally utilized, and the overhead is magnified. Lastly, TVM is also able to achieve consistent speedups across each benchmark, similar to Covenant, with the added advantage of LLVM optimization passes. As a result, TVM manages to achieve high performance for even small benchmarks such as DLRM-FC4, but does not attain the significant speedups of nnlib which required specialized tensor transformations.

Optimization Results

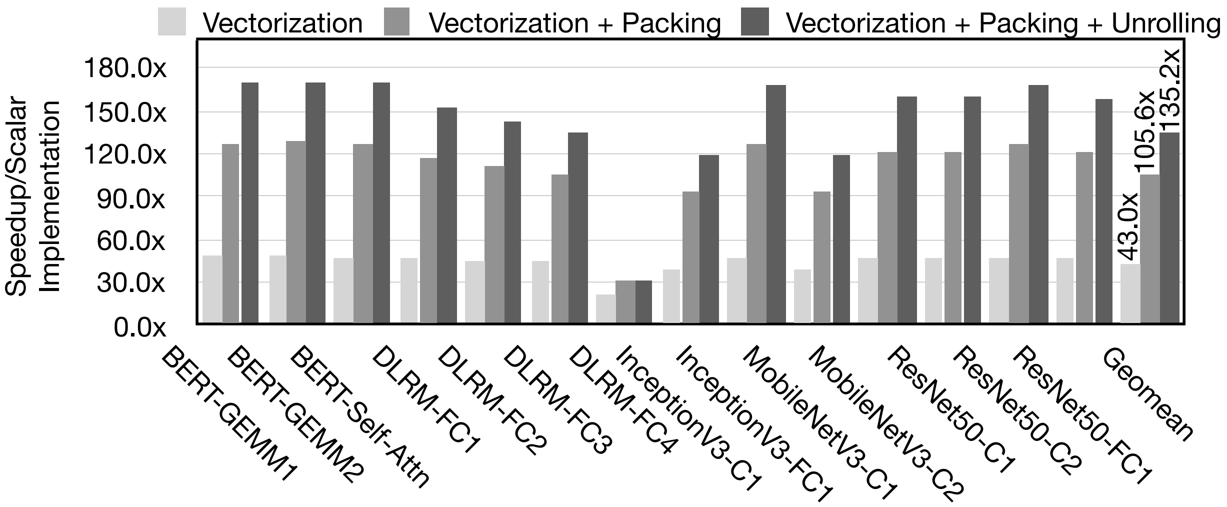


Figure 3.12: Performance improvements based on code optimizations implemented in Covenant.

We evaluate the effectiveness of three Codelet optimizations when targeting Qualcomm© Hexagon™ HVX[1] DSP [24]. Figure 3.12 shows the benefits across the benchmark DNNs enabled by the optimizations. The baseline is vanilla Covenant implementations for the DNN layers. We first use Vectorization based on the parallelization techniques described in Section 3.4. We then enable Mnemonic Packing, as described in Section 3.4, on top of Vectorization. Finally, we add the third optimization, Loop Unrolling as discussed in Section 3.4. As the figure 3.12 shows, Vectorization is the most effective technique. This is

[1]Qualcomm Hexagon HVX is a product of Qualcomm Technologies, Inc. and/or its subsidiaries.

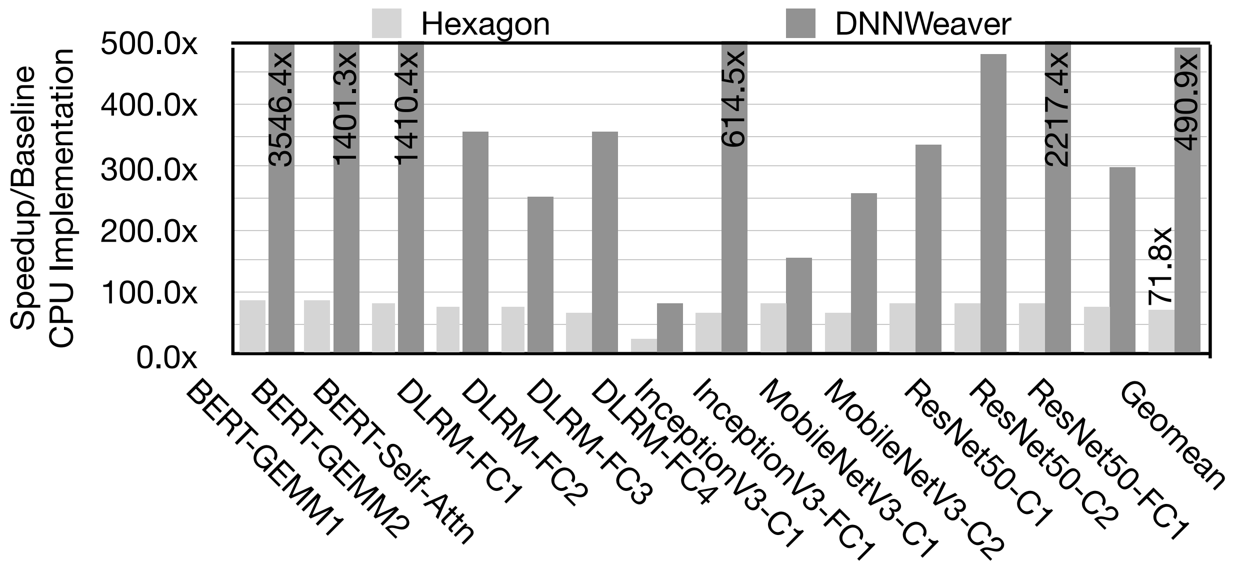


Figure 3.13: Performance results of evaluated hardware, while using Covenant for compilation.

a due to massive data-level parallelism available in both DNN layers, as well as Hexagon's Vector Extensions. Among the benchmarks, DLRM-FC4 sees the least improvement due to its relatively smaller matrix dimensions. On average across all DNN layer benchmarks, Vectorization achieves 43.0× speedup compared to the baseline CPU implementation. Mnemonic Packing leverages the mnemonic level parallelism opportunities in compiled DNN mnemonics to utilize the four available instruction slots in Hexagon DSP VLIW architecture. On average, it brings about an additional 2.4× performance improvements. Finally, Loop Unrolling is enabled to facilitate efficient memory accesses, which provides a 1.3× extra performance improvements, on average.

Multi-Target Compilation

To demonstrate the flexibility in targeting various hardware architectures, we use Covenant to compile to two different styles of architectures. Hexagon DSP is a more general-purpose-style architecture that supports a wide range of operations. On the other hand, DNNWeaver is a domain-specific specialized DNN accelerator with a systolic array architecture that only supports DNN execution. Figure 3.13 shows the performance of these two hardwares compared to the baseline CPU implementation. On average, Hexagon brings 71.8× speedup over baseline CPU, while DNNWeaver provides 490.9× performance improvements, both using Covenant for compilation. The higher speedups offered by DNNWeaver are due to two reasons: 1) DNNWeaver harbors 32× more number of compute resources compared to Hexagon and 2) it utilizes a systolic array architecture which is specialized for vector-matrix multiplications, as opposed to SIMD architecture of Hexagon HVX. Across all benchmarks, DNNWeaver performance improvements are more pronounced for larger DNN layers, as they require large matrix multiplications, suitable for systolic array architectures.

3.6 Related Work

With the growing interest in DNN accelerators, creating efficient and flexible compilers for them is increasingly vital. This work fundamentally differs from prior works in that it integrates a novel accelerator architecture abstraction (ACG) into the compilation stack through Codelets construct. These two enables seamless reuse of the same compiler across various accelerators. Below, we discuss the most related works.

Compiler Infrastructure for DNN Accelerators. MLIR [70] and Glow [101] seek to enable compilation for different targets by offering multiple levels of IR. However, they fall short of code generation due to not offering a mechanism to describe the target hardware. Tensorflow's XLA [6] is another framework that uses a high-level graph IR for compilation to general-purpose processors and domain-specific Google's TPUs. Similarly, XLA is a set of optimizations on a specialized IR that is a representation of the DNN and does not concern itself with abstractions for the hardware (i.e., ACG and Codelets).

Architecture Abstractions for Scheduling. A prior work has leveraged architecture abstractions for scheduling on spatial architectures by modeling them as directed graphs [89].

This work is focused solely on scheduling methodology and does not deal with code generation, whereas Covenant comes with a complete compilation stack that leverages Codelets to facilitate use of scheduling techniques for code generation.

Architecture Abstractions for Hardware Generation. A number of prior works have used DSLs to incorporate architecture features into algorithm specification for the purposes of hardware generation [87, 68, 63]. LLHD [105] uses MLIR [70] to simplify hardware design and generation by defining an architecture description language. Covenant fundamentally differs from these prior works because it aims to leverage architecture abstractions to compile to various existing hardware as opposed to generating new hardware.

Low-level IRs for DNN Scheduling. Halide [98] and it's extensions [118] introduced the idea of distinguishing between computation and schedule to compile image processing pipelines, and include schedule transformations for common optimizations. TVM [19] takes inspiration from Halide and uses tensor expressions combined with additional scheduling operations such as tensorization to optimize code generation. Schedules for tensor expressions in TVM's IR support arbitrary transformations regardless of the target backend, but can be constrained with manual construction of a valid schedule templates for each tensor expression to constrain code generation [20]. Tensor Comprehensions [115] and PlaidML [121] automate the scheduling process using tensor-based IRs, yet lack flexibility for scheduling to new hardware. These works do not propose or integrate an architecture abstraction into the compiler. Moreover, in contrast to these IRs, the Covenant compiler performs scheduling by integrating architectural details into Codelets, enabling scheduling algorithms be reused across DNN operations and different targets.

Schedulers for DNN Operations. Another body of works have focused solely on scheduling for different architectures. FlexTensor [123] and Ansor [122] automate the scheduling process by extending TVM's code generation backend. However, they cannot perform scheduling for the accelerators without a pre-existing compiler and runtime environment. Fireiron [42] is a scheduling language for targeting to only GPUs that explicitly incorporates data movement into schedule definitions. CoSA [54] is a scheduling framework that incorporates hardware features into a mixed-integer programming algorithm to form constraints on schedules, without support for code generation. In contrast, Covenant compiler leverages the combination of ACGs and Codelets to provide a uniform and automated compilation framework with code generation backend for targeting to various DNN accelerators.

3.7 Conclusion

DNN accelerators are introducing a new age of compiler design requiring alternative constructs and abstractions. This paper defines two such building blocks, ACG and Codelets. The ACG is an architecture abstraction which makes various components of the accelerator and their connectivity accessible to the compiler. The ACG is integrated into the Covenant compiler through the Codelet construct which represents mutable operations on DNNs, and is progressively transformed into execution mappings and schedules on the ACG. The encouraging empirical results show this work is an effective step towards developing compilers, targeting different accelerators.

3.8 Acknowledgement

Chapter 3 is a partial reprint of the material as it appears in S.Kinzer, S. Ghodrati, R. Mahapatra, BH. Ahn, E. Mascarenhas, X. Li, J. Matai, L. Zhang, H. Esmaeilzadeh, "Restoring the Broken Covenant Between Compilers and Deep Learning Accelerators Deep Learning Accelerators". The dissertation author was the principal investigator and author of this paper.[1]

[1]© 2021 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Chapter 4

VeriGOOD-ML

4.1 Introduction

Recent advances in machine learning (ML) algorithms have seen a proliferation of new ML algorithms and architectures, as well as new work on ML accelerators. However, the design of these accelerators requires intense manual designer effort and is time-consuming. There is considerable recent interest in real-time machine learning (RTML), where data is sent to an ML accelerator chiplet through fast interfaces [60] and processed on the chiplet in real time, with applications ranging from ML tasks in autonomous vehicles (e.g., obstacle detection, collision avoidance, path planning) to next-generation wireless networks (e.g., resource sharing in virtualized radio access networks, channel estimation, channel decoding, RF fingerprinting). These applications are best supported by building an ability for rapid translation from an ML algorithm to a hardware implementation.

VeriGOOD-ML is an open-source project [4] that automatically compiles a high-level description of an ML algorithm (in a standard ML format such as ONNX) to a register-transfer level (RTL) Verilog implementation with no human in the loop. The RTL is then taken through synplace-and-route, resulting in a silicon implementation.

The entire design flow, from architecture design to physical implementation, is guided by models for performance, power, and area (PPA), working in conjunction with architectural simulation. This enables the designer to perform cross-layer optimizations to build high-performance design implementations that can be optimized for various objectives: size, power, performance, or solution quality (using bitwidth quantization).

The ML algorithm is specified using the Open Neural Network Exchange (ONNX) format, which is widely supported, thus maximizing interoperability across various programming environments. ONNX represents ML algorithms as a standardized graph to facilitate interoperability across various development environments, including Google Tensorflow, Microsoft CNTK, and Facebook PyTorch. The starting point for VeriGOOD-ML is the PolyMath compiler [62], which translates a high-level ML algorithm description (e.g., ONNX) into our intermediate representation (IR). The IR is a hierarchical representation that we refer to as a simultaneous recursive dataflow graph (*sr*-DFG) that allows a hierarchical view into the structure of a design.

VeriGOOD-ML targets ML engines for both training and inference. It uses three core engines to synthesize hardware from the IR. Two of these are platform-based: **TABLA** [77], for general non-DNN ML algorithms (e.g., linear regression, logistic regression, SVM), and **GeneSys** for general DNN algorithms. TABLA uses a dataflow architecture; the core computation engines in GeneSys are a systolic array (for operations such as convolution) and a SIMD array (for operations such as ReLU and pooling). The platforms are parameterizable, and it is possible to automatically generate hardware with different numbers of processing elements, bitwidths, and on-chip memory configurations. A third approach, **Axiline**, is a hard-coded engine tailored to specific small ML algorithms: it trades off the flexibility of a platform, which can run multiple ML algorithms, for a power-efficient implementation that is tailored to a single algorithm. For TABLA and GeneSys, the platform-based architectures, PolyMath translates the *sr*-DFG into “Codelet” templates that implement the ML algorithm on an instruction set that is specific to the platform. The Axiline implementation is synthesized by translating the *sr*-DFG into dedicated hardware. Our silicon implementation efforts characterize the PPA of core building blocks and develop methodologies that provide PPA tradeoffs that generate Verilog with physical implementation considerations.

Throughout the flow, VeriGOOD-ML optimizes the design for performance, producing a set of designs with Pareto-optimal performance/power/area (PPA) tradeoffs, and connecting these with system-level performance metrics that optimize the power and execution time for implementing an ML algorithm. In particular, a design planner, which performs floorplanning and power grid generation for the macro-intensive layout, is vital in ensuring that the back-end implementation delivers high performance. The flow includes cycle-accurate simulators for each engine, and is coupled with silicon PPA predictors that can be used to perform design-space exploration, yielding optimized ML hardware engines.

4.2 Compiling ONNX to Platform-Specific Instructions

In this section, we describe how the ONNX description of an ML algorithm is converted to an intermediate representation (IR), and together with information about the hardware, is used to perform end-to-end compilation using the PolyMath framework [62] for execution on TABLA, GeneSys, and Axiline.

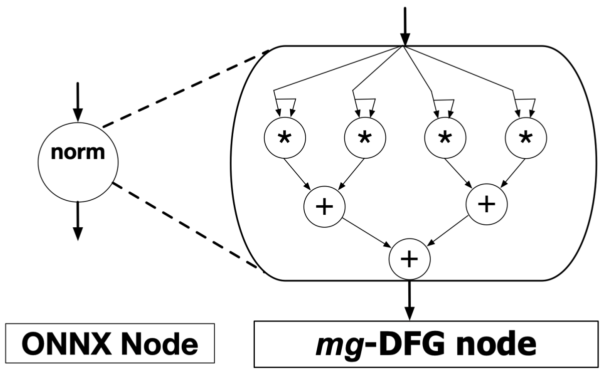


Figure 4.1: Translation of a “norm” ONNX node to the equivalent *sr*-DFG node that contains all the fine-grained operations that constitute a “norm” operation.

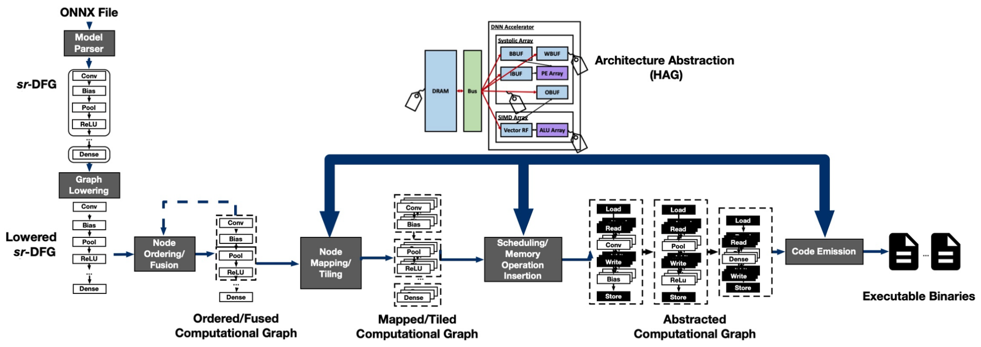


Figure 4.2: ONNX-to-hardware mapping flow through the *sr*-DFG and HAG representations.

Intermediate representation using an* sr*-DFG: To encapsulate operations at multiple levels of hierarchy, we devise a simultaneous recursive dataflow graph (*sr*-DFG), an IR that is recursively defined with the *sr*-DFG nodes. The representation facilitates optimization in several ways: (1) utilizing optimizations that are predeveloped for certain complex operations (e.g., building a binary tree for the L2 norm or optimizing the flow of data for convolution) and (2) simultaneously preserving the capability to perform fine-grained scheduling and mapping optimization.

To translate an ONNX description into an *sr*-DFG, we traverse the ONNX graph, whose nodes represent coarse-grained ML operations on multi-dimensional arrays of input data. During traversal, *sr*-DFG nodes and edges are generated using the attributes of each ONNX operation and its inputs/outputs. The operations that comprise each coarse-grained operation (e.g., multiply-adds that constitute a norm operation, as shown in Fig. 4.1) are added to each *sr*-DFG node using instantiations of predefined templates. We have successfully created *sr*-DFG representations for a variety of benchmarks that cover a variety of machine learning algorithms – both non-DNN (linear regression, logistic regression, support vector machines, recommender systems, backpropagation) and DNN ML algorithms.

Modeling hardware using a HAG: We model the structure of specific accelerator platforms by introducing a reusable hardware abstraction called a hierarchical architecture graph (HAG), with a corresponding architecture description language embedded in Python for targeting different types of accelerators with a unified interface. A series of compilation passes use the HAG for a specific target accelerator for mapping, scheduling, and optimizing programs on the accelerator. Each HAG is comprised of three types of nodes: for computations, for on- and off-chip communication, and for storage. In interaction with the *sr*-DFG, the HAG enables end-to-end compilation by the introduction of hardware-specific attributes to the compilation pipeline.

An architecture description language (ADL) is used to represent the HAG. Such an abstraction enables the compiler to expand its capability from optimizing for single piece of hardware to a heterogeneous computing environment where there are multiple disparate processors and accelerators. This ADL is built on top of Python to improve usability and versatility, easily working in tandem with various machine learning frameworks. To represent diverse types of accelerators, there are several primary attributes that must be included in the abstraction: the ability to (a) model hierarchy (as fine-grained as a single ALU, or as coarse-grained as an entire systolic array); (b) specify compute, storage, and communication components; and (c) annotate each node with attributes/metadata including, but not limited to, storage node capacity, communication bandwidth, input and output ports, latency, and computation node capabilities that describe operations supported by the architecture. Note that the architecture description is primarily intended for compilation purposes, and captures design information at a high level, eschewing a more detailed gate-level description.

Compilation to target accelerators: Having devised an abstract representation of different types of accelerator architectures, a multi-stage compilation process can be reused across different HAGs. The stages of compilation, illustrated in Fig. 4.2 consist of:

Operation mapping/scheduling: An *sr*-DFG is ordered to a sequence of operations, and each operation will be mapped to a particular component in the HAG according to the *sr*-DFG node operation and the sequence of capabilities that produce the equivalent operation. In addition, sequences of operations can be fused together according to user-supplied parameters.

Compilation optimization: A search for optimal compilation parameters is performed using specifications of the HAG, such as tiling sizes, loop unrolling factors, dataflow, etc. During this process, data communication instructions/operations, including off-chip communications for both read/store operations, are added according to these parameters.

Code generation for the target HAG: This step is based on the instantiated capabilities from the two previous steps. The compiler combines code templates called Codelets with the *sr*-DFG node attributes and HAG attributes. Codelets represent instruction templates for target accelerators. The *sr*-DFG is converted to this abstraction for every type of accelerator, with the only difference being the underlying instruction template used for binary generation. There are four primary types of Codelets: (i) Compute Codelets that represent instructions for performing computations on data; (ii) Memory Codelets for instructions that move data from one memory location to another (e.g., load from/store to off-chip or on-chip memory); (iii) Loop Codelets that repeat operations over a number of loop levels; and (iv) Control Codelets for instructions that determine program flow. These Codelets are combined to form operations that match the semantics of execution for a given *sr*-DFG node.

The overall compilation flow is depicted in Fig. 4.3, which demonstrates the different stages as well as the ability to apply architectural attributes to the compiler passes. In combination with the code templates associated with Codelets, additional compiler passes were implemented to optimize and transform the program, e.g., datatype transformations, layout transformations, and padding tensors for GeneSys to map data onto the systolic array and SIMD array.

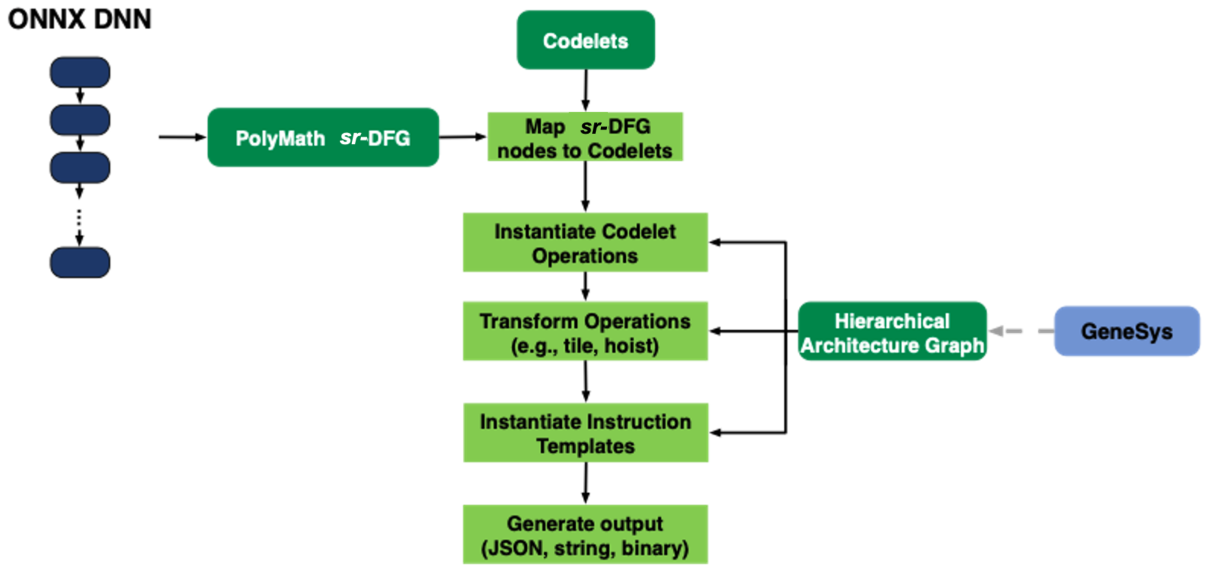

ONNX DNN
Codelets
PolyMath *sr*-DFG
Map *sr*-DFG nodes to Codelets
Instantiate Codelet Operations
Transform Operations (e.g., tile, hoist)
Hierarchical Architecture Graph
GeneSys
Instantiate Instruction Templates
Generate output (JSON, string, binary)

Figure 4.3: Compilation flow combining the HAG and Codelets to apply multiple stages of transformation and optimization.

4.3 Target Hardware Substrates

In this section, we overview three target substrates for VeriGOOD-ML: TABLA for non-DNN ML algorithms, GeneSys for DNNs, and Axiline for ultraefficient hardcoded implementations of small ML algorithms.

4.3.1 The TABLA Platform for Non-DNN ML Algorithms

Overview of the TABLA architecture: The overall TABLA architecture [77] for training and inference for non-DNN ML algorithms is shown in Figure 4.4 and consists of multiple levels of hierarchy. An array of *processing units* (PUs) constitutes the first level. The PUs are connected through two different busing mechanisms – the "neighbor bus" and the "global bus." All PUs are connected to the global bus, and the communication between all the PUs imposes a high pressure on the global bus. The neighbor bus aims to minimize this pressure by enabling the adjacent PUs to send their data through it. Moreover, connecting all PUs to the global bus can result in a race between the PUs. To ensure proper data

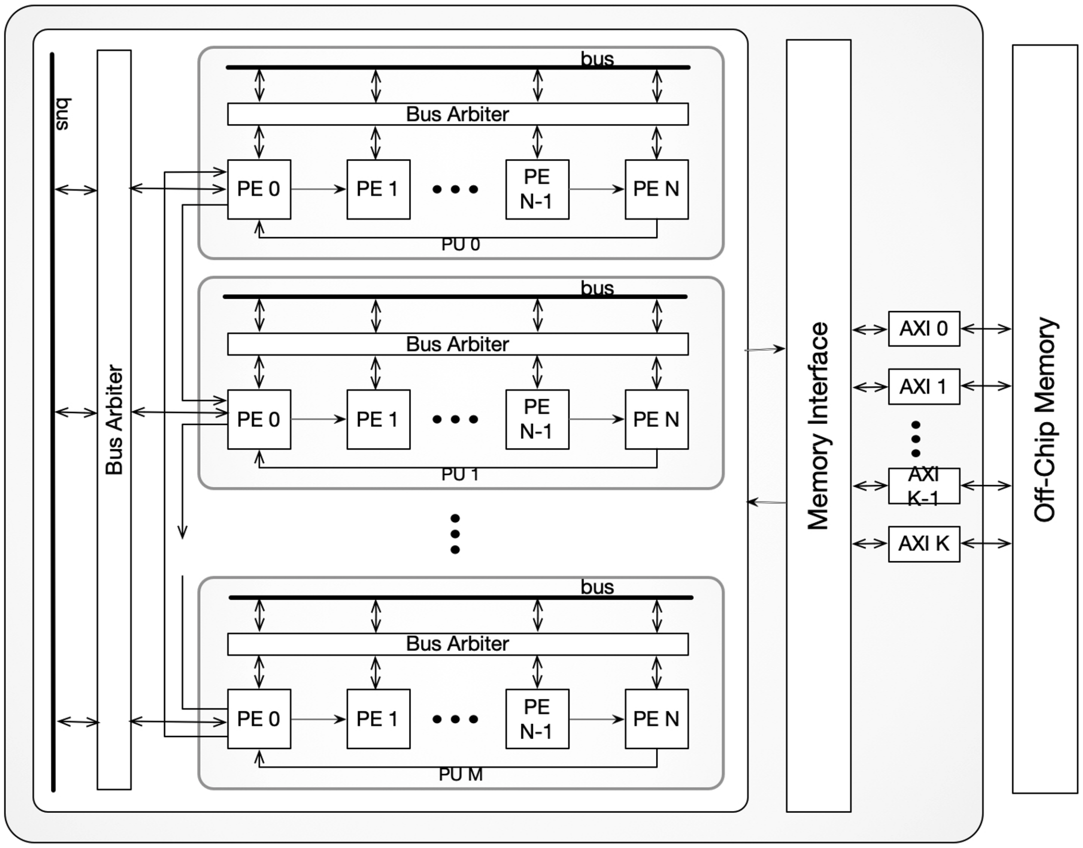


Figure 4.4: An overview of the template-based architecture for the non-DNN accelerator.

transfer between PUs, a bus arbitration module is implemented.

At the next level of hierarchy, each PU comprises of a set of *processing engines* (PEs). Similar to buses for inter-PU communication, there are two buses for inter-PE communication. The *bus arbiter* consists of a single leader controller per PU and one follower controller for each PE. The leader controller determines which PE has control of the bus in a given cycle, and the follower controller has a write buffer and a set of read buffers (one for each PE/PU), organized as FIFOs. In each cycle, data is popped from the write buffer of the source PE and written to the read buffer of the destination PE.

Cycle-accurate software simulator: To facilitate testing and verification of the architecture, we have designed and developed a cycle-accurate simulator in software that emulates the architectural behaviors of the proposed system. The simulator allows the user to provide the input program as an *sr*-DFG file and a configuration file that sets the parameters of the template architecture described in the above sections such as number of PEs per PU. Taking the configuration file as an input allows users to further test the behavior of the architecture with varying degrees of parameterization, e.g., to analyze the performance impact of changing the number of PEs per PU. Based on cycle-by-cycle analysis, the simulator can emulate the execution of a given program and output performance metrics such as total number of cycles, PE and PU utilization, and scratchpad utilization.

4.3.2 The GeneSys Platform for DNN Algorithms

Overview of the GeneSys architecture: The overall system view of the GeneSys DNN accelerator is shown in Fig. 4.5. The accelerator consists of two core components: a systolic array and a SIMD array. Data is supplied to the engine through the input buffer (IBUFF), output buffer (OBUFF), instruction memory (IMEM), weight buffer (WBUFF), and bias buffer (BBUFF). These interfaces harbor programmable address generator modules and controller FSMs that together generate the addresses and requests to load or store a tile of data from/to off-chip memory. The address generators perform strided address pattern generation and generate addresses in the off-chip memory and read/write the corresponding data from/to on-chip buffers and populate the on-chip memory. These interfaces also include tag logic that is in charge of handling double-buffered data transfer to hide the latencies of Load/Store operations and also facilitate prefetching. Among these interfaces, the interface for OBUFF and SIMD array handles both load and store operations, while the other interfaces handle only load operations. These interfaces are fully programmable through the instruction set architecture (ISA) of the GeneSys accelerator.

The **systolic array**, which performs convolution and matrix multiplication operations for the convolution and fully-connected layers, is a 2D array of $M \times N$ processing engines (PEs), equipped with dedicated on-chip weight buffers, as in [106, 107]. To boost the operating

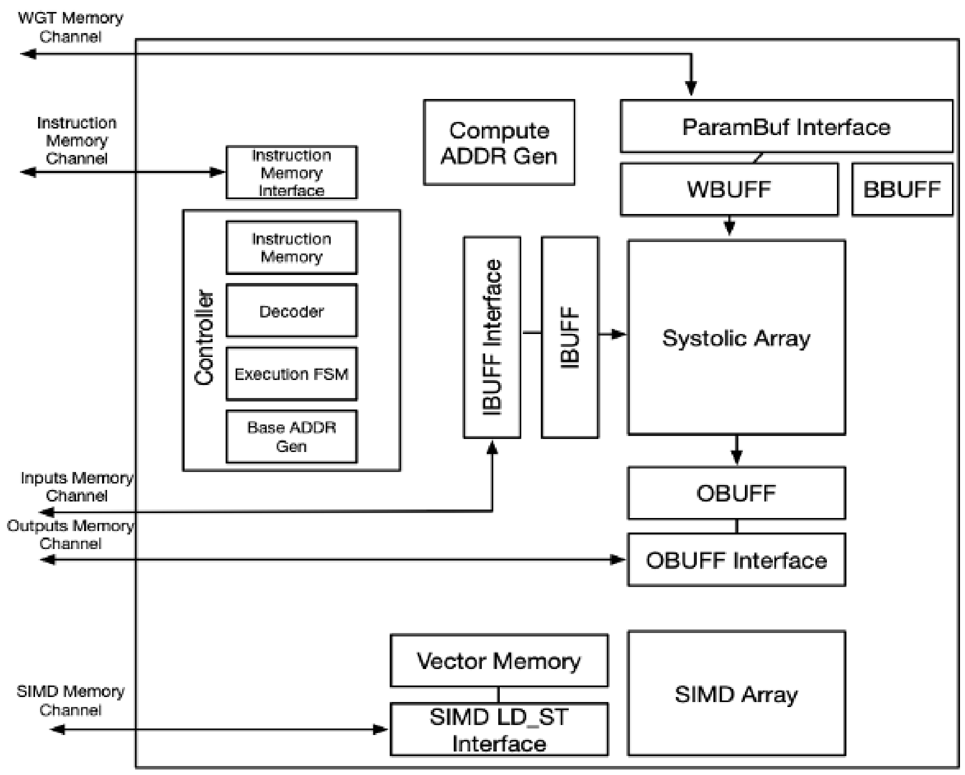


Figure 4.5: A block diagram of the overall system view of GeneSys, the VeriGOOD-ML DNN accelerator.

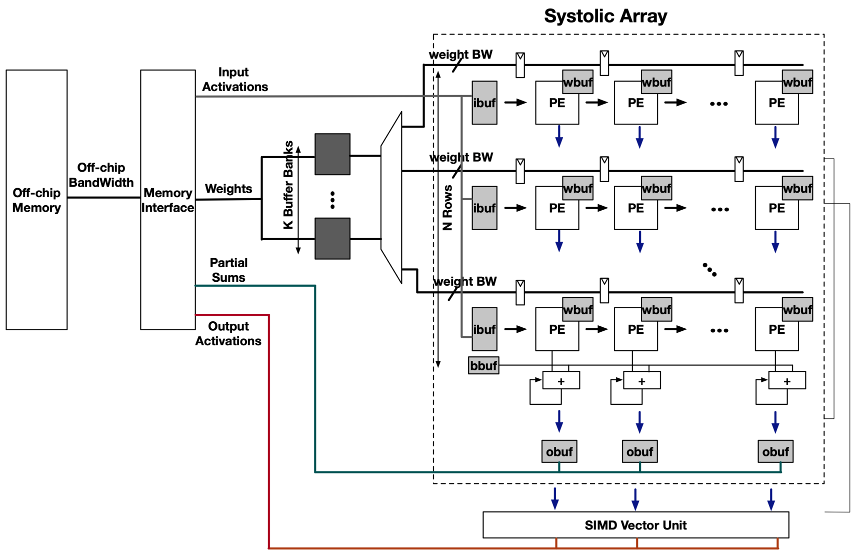


Figure 4.6: Execution flow of the GeneSys systolic array accelerator.

frequency, we pipeline the inputs and weights across the columns of the array and the partial sums across the rows of the array. In systolic execution, the inputs (activations) flow horizontally, are multiplied by the weights in each PE and are then accumulated vertically along the columns of the systolic array. This systolic execution also facilitates mapping the matrix-multiplications and convolutions to the array and simplifies the control logic. The IBUFF is multibanked and each bank feeds a row of the systolic array. The output buffers are also multibanked, each bank for each column of the systolic array, storing the partial sums and output activations.

Figure 4.6 depicts a more detailed diagram of the implementation of the systolic array. Each processing engine consists of (1) a weight scratchpad that stores the weight values on-chip and (2) a multiply-accumulate unit that performs a multiplication between the inputs and weights and an accumulation of the partial results to perform the matrix-multiplication or convolution operation with the systolic array. Each PE is equipped with four registers that aim to support the pipelined execution: a register for the output results, a register for the received input that will be forwarded to the adjacent PE in the systolic array, and two registers for handling the read accesses from the weight scratchpad (one register for the read request and one for the read address; the read request and read addresses for the weight scratchpads are shared across the 2-D array of PEs). Each PE is a template design and the size of the weight scratchpad, precision of the input, weight, partial sum and also the bitwidth of the multiply-accumulate logic in addition to the registers are parameterizable during architectural synthesis, according to the demands of the application.

For address generation, we design a memory walker module that can automatically generate the addresses for executing convolution/matrix-multiplication operations on the systolic array, leveraging the insight that the data layout and memory patterns of DNNs are generally regular, without branch/jump instructions. This module is configured with a set of parameters such as the number of loop iterations and the base address in the memory, and can then generate addresses automatically as:

$$address = base\_address + loop\_iterator \times stride$$

The **SIMD Vector Unit** is a $1 \times N$ array that performs computations for DNN layers other than convolution and fully-connected layers, such as pooling, activation, and other element-wise operations. The pipeline stages of this SIMD processor are generally similar to a MIPS processor with a major difference: since memory access patterns in DNNs are regular, the register file is eliminated to save Load/Store instructions. With this design, we directly read from the on-chip scratchpads that store the data, execute the operations, and then write it back to the destination scratchpad. We have designed a custom ISA to program this architecture. There are two classes of instructions in this ISA: execution instructions (ALU, CALCULUS, COMPARISON, DATATYPE CAST), and setup instructions (DATATYPE CONFIG, ITERATOR CONFIG, LOOP).

A training-capable GeneSys implementation consists of additional layers and operations beyond the inference engine for performing gradient computations and parameter updates. Training operations must support computations of loss gradient with respect to input and weight: for a convolution layer, these are mapped to a convolution operation, and for a fully connected layer, they are implemented as a GEMM operation. For training, GeneSys supports a softmax layer, a common generic model for multiple operations (e.g., parameter updates for 1D, 2D, and 4D tensors; loss gradient computation for the ReLU layer and for element-wise addition of two tensors; reduction of a tensor along its dimensions), and estimated models for the batch normalization layer, including operations during the forward and backward pass.

GeneSys performance simulator: Our simulator for DNN execution on GeneSys takes the following two files as inputs: (1) a specification of the hardware configuration, in the form of a .json file, and (2) the compiler output, as a .json file containing a high-level description of each DNN layer, e.g., the dimensions of the input/output tensors, order of execution of the loops, tile sizes for the tensors and datatypes.

The simulation framework is attuned to the fully parameterizable nature of GeneSys by accepting the specific hardware attributes:

- the dimensions of the 2D PE array, the sizes of each of the on-chip buffers, namely, WBUFF, IBUFF, OBUFF, and BBUFF for the systolic array, and vectory memory, immediate memory, and instruction memory buffers for the SIMD array.
- bit-widths of all types of data (filter, input, bias, psum, output for the systolic array; input, psum, output for the SIMD array).
- the number of cycles required by various arithmetic operations.
- off-chip bandwidth of each memory interface.

For each layer of a DNN, either executed on the systolic array or SIMD array, the simulator outputs the following performance statistics: the number of accesses for each of the on-chip buffers for each datatype, the number of accesses for the off-chip DRAM for each datatype, the number of accesses for the pipeline registers, the number of various arithmetic operations, the number of on-chip compute cycles, the number of stall cycles while the Systolic array or SIMD array remain idle waiting for data to be fetched from the off-chip DRAM, and the total number of execution cycles.

4.3.3 The Axiline Approach for Hard-Coded ML Hardware

The Axiline generator develops dedicated, hard-coded implementations of small algorithms, for both ML training and inference, to achieve high performance and low power. For TABLA and GeneSys, the parameters for the platform can be selected according to target applications, but may be used to run other applications. In contrast, Axiline is intended to be very specific to the ML algorithm that it implements, and it trades off adaptability for performance. By building a hardcoded implementation, we can achieve maximum performance and efficiency, at the expense of flexibility.

The Axiline generator outputs RTL by creating a mapping from an *sr*-DFG input to unit constructs such as inner products, adders, and multipliers. The simplest version of Axiline begins with an *sr*-DFG without loops and translates it to a combinational implementation. However, the cost of implementing a larger *sr*-DFG, or one with loops, may become prohibitive due to the large volume of data to be processed. For such scenarios, we develop an iterative architecture that serially processes parts of the input data over multiple cycles.

The generator works in three steps: first, it generates the lowered data flow graph for an Axiline ML algorithm; next, it calculates the bitwidth for each node, based on the given bitwidth of activation, weight and bias, and finally, it generates Verilog code for each node/block and combines them with the template. A representative multicycle pipelined architecture that can be used for several non-DNN benchmarks (e.g., SVM, logistical regression, and linear regression) is shown in Fig. 4.7. The architecture maps the *sr*-DFG

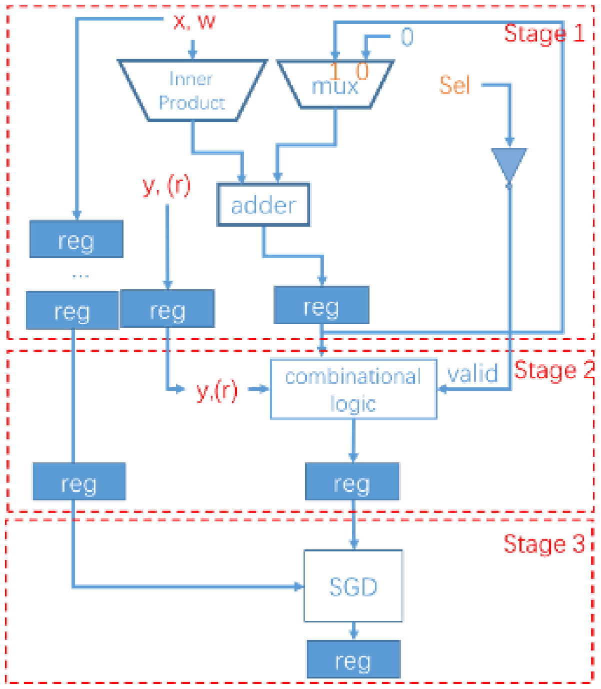


Figure 4.7: Pipeline implementation for Axiline benchmarks.

into three pipeline stages: Stage 1 performs an inner product computation, and is followed by Stage 2, which implements a combinational function, where the precise function depends on the benchmark. For example, for linear regression, the combinational logic in block 2 would be a multiplier, and for logistic regression benchmark, it should be a sigmoid function and a multiplier. Block 3 is for stochastic gradient descent, consisting of two multipliers and one adder. The inner product size in Stage 1 is parameterized. Therefore, the input bandwidth can be parameterized for different FPGAs. The computation proceeds iteratively by processing data through this pipeline.

4.4 Synthesizing Hardware

The VeriGOOD-ML compiler takes an ML algorithm from an ONNX-level description to Verilog RTL. The next step in synthesis is to go from Verilog to GDSII. A critical first step in back-end implementation of machine learning algorithms to advanced-node silicon,

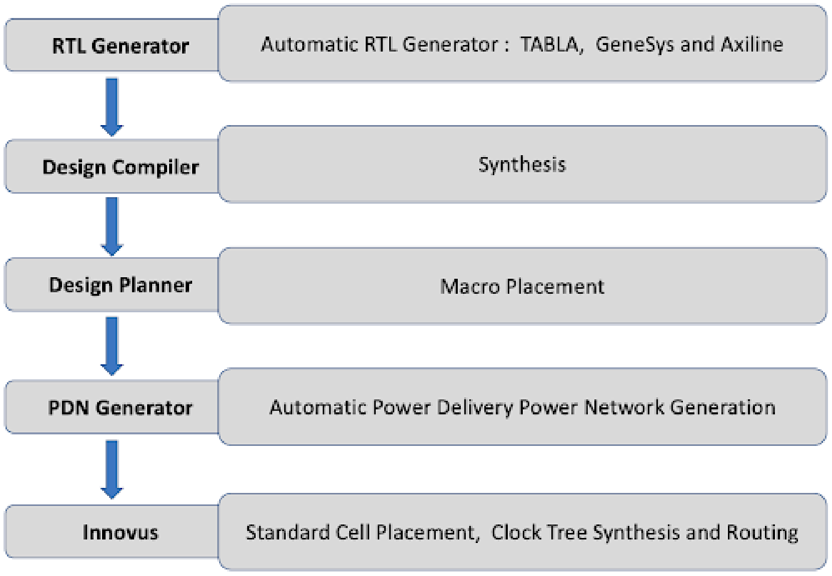

RTL Generator
Automatic RTL Generator : TABLA, GeneSys and Axiline
Design Compiler
Synthesis
Design Planner
Macro Placement
PDN Generator
Automatic Power Delivery Power Network Generation
Innovus
Standard Cell Placement, Clock Tree Synthesis and Routing

Figure 4.8: An overview of the SPR flow from RTL to GDSII.

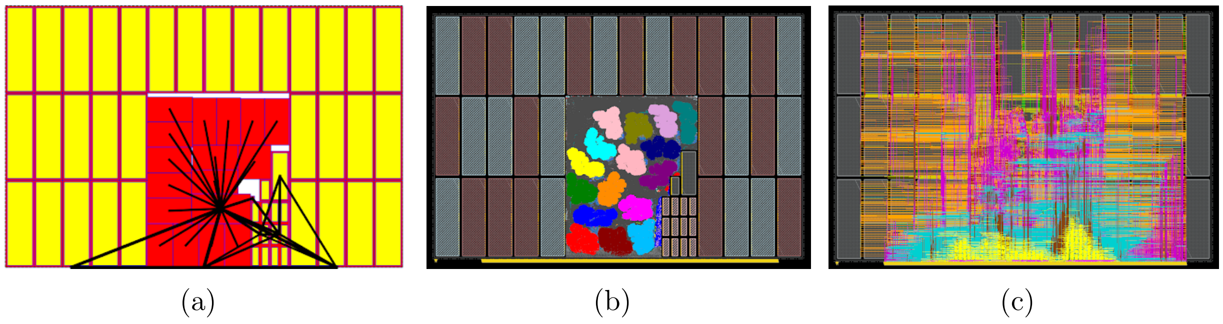


Figure 4.9: Back-end synthesis of a GeneSys engine showing (a) signal flow on primary interconnects, (b) the generated floorplan, (c) the final result of SPR.

particularly with automatically generated RTL, is design planning. ML accelerators are inherently very structured, and optimal silicon implementation requires a design flow to leverage that structure to create a high-quality floorplan. This is a critical first step that is essential both for physical synthesis and place-and-route. A suboptimal floorplan can result in poor PPA and increased turnaround time for design closure.

Historically, design planning has initially been performed by the front-end designer who understands the RTL design hierarchy and connectivity and further refined by the back-end engineer, who understands the floorplan effects and utilizes constraints from the SoC regarding block outline and pin positions. As design complexity increases, this becomes practically impossible; moreover, for auto-generated RTL, there is no front-end designer who understands the design. Hence there is a critical need for an automated design planning tool that is compatible with commercial EDA tools.

VeriGOOD-ML uses a design planning flow and key engines that have been implemented in the open-source OpenROAD tools [11, 3] so as to bridge generated RTL Verilog to successful physical implementation outcomes. In our flow, we pass the result of design planning to a place-and-route flow using commercial tools; in future, a fully OpenROAD-based flow will be targeted. The overall synthesis, place and route (SPR) flow is shown in Fig. 4.8.

Our in-house design planner is designed to mimic the way expert chip designers perform floorplanning. A significant challenge is related to the fact that these designs are dominated by macros that correspond to memory modules that implement various on-chip buffers. This adds complexity to the tasks of floorplanning, which must leverage design regularity, and power delivery network (PDN) generation, which must handle PDN blockages in several metal layers at the macro locations.

The design planner first creates an efficient abstraction model of the netlist by analyzing attributes such as the logical hierarchy, data flow, the connection between macros and input-output pins, and timing-critical paths. The planner then uses the abstraction model to guide the generation of the floorplan. This model helps back-end engineers to gain better insights into the design and therefore reduces the number of iterations required to make the design flow converge. Four engines that are invoked sequentially:
(1) The *auto-clustering engine* converts the gate-level netlist representation of the design into a clustered netlist, in which nodes are clusters and nets are bundled connections between clusters. To generate this clustered netlist, we first create clusters based on logical hierarchy and then group small clusters based on connection signatures. To handle macro regularity, we group macros with different sizes into different hard macro clusters. We

then add virtual connections between hard macro clusters and input/output IOs based on dataflow and latency.

(2) The *shape engine* determines possible aspect ratios and area for each macro based on core size of floorplan and target utilization. For each hard macro cluster, we enumerate all possible minimum-area packings.

(3) The *macro placement engine* places all the clusters and finalizes the shape of each cluster. In this phase, we use a sequence-pair representation of clusters in the netlist, and simulated annealing to optimize the cost function. The cost function includes area, wirelength, and several penalty function terms, e.g., for overflowing the given layout region (fixed-outline constraint), or for notches or blocked pin accesses in the macro placement.

(4) Finally, the *pin alignment engine* determines the location and orientation of each individual macro. In this phase, we pack macros within each hard macro cluster, again using simulated annealing of a sequence-pair representation.

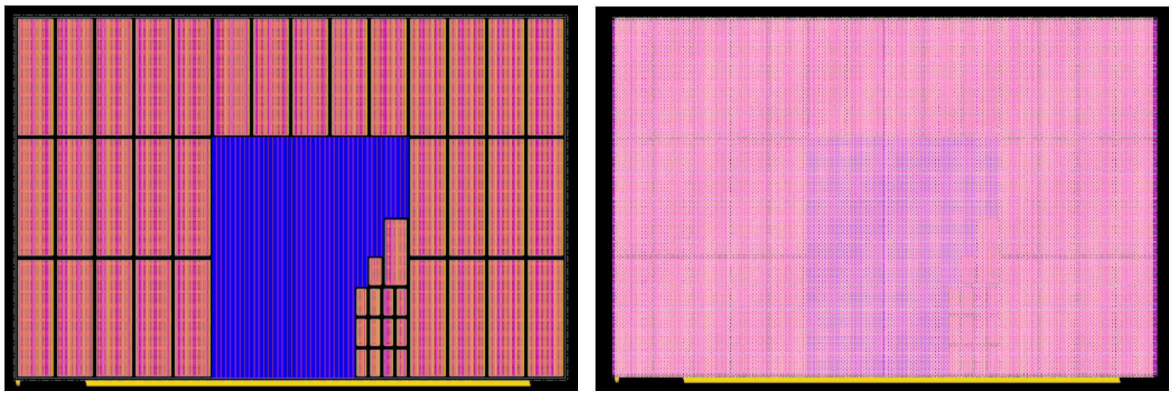

(a) (b)

Figure 4.10: The PDN on (a) layers M1–M7 (b) layers M1–M13.

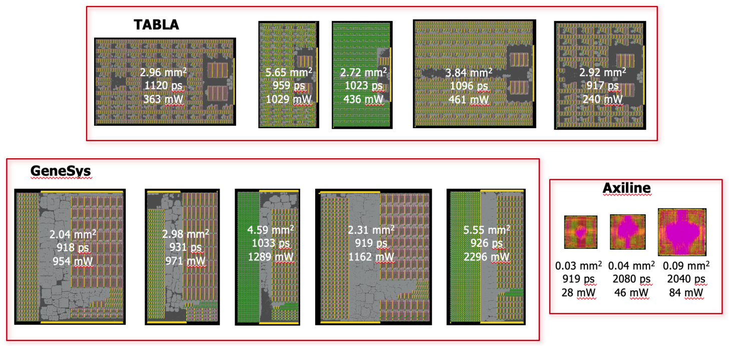
We implement our designs based on the GF12LP technology using 13 metal layers. An Arm memory compiler is used to build dual-port register files. For each logical memory size (address and bit width), the configuration that yields the smallest area is chosen. Fig. 4.9 shows the data flow, the floorplan from our design planner, and the final place-

and-route on a commercial back-end for the GeneSys SIMD example. The automatically generated PDN is illustrated in Fig. 4.10.

Using this back-end implementation flow, we are currently in the process of taping out a chip that implements a GeneSys engine. Aside from core GeneSys components, the design includes an on-chip global buffer that interacts with the external off-chip memory, as well as mixed-signal circuits such as VCOs, synthesized using ALIGN [66].

4.5 Results

We have applied the VeriGOOD-ML flow to perform training and inference on a variety of ML algorithms, exploring the space of design configurations to optimize application-level performance metrics. For a variety of design configurations of a specific platform (TABLA or GeneSys), we generate the Pareto-optimal PPA curves for the hardware engine using our back-end implementation methodology. This yields the power and frequency characteristics of the platform. Using the cycle-accurate simulator, we track the performance of the ML algorithm on the platform, e.g., the number of cycles required to perform the computation and the memory access patterns that dictate stalls and power dissipation. Based on this, we determine the power and execution time of the ML algorithm on the platform. For example, for DNN execution on GeneSys, we combine the performance statistics provided by the simulator with the power-performance characteristics (i.e., energy per operation, clock frequency, dynamic and leakage power of various hardware components) of Pareto-optimal PPA design points provided by our backend Synthesis Place-and-Route flow to compute the energy consumption, power (both on-chip and off-chip), and runtime. For Axiline, the mapping is performed directly to report the power and execution time. In this section, we provide a snapshot of a set of results obtained from exercising VeriGOOD-ML. A variety of design implementations have been built, up to post-SPR; a sample set is shown in Fig. 4.11. These implementations create a Pareto-optimal set of designs that form the basis for the results shown below.



(a)

Figure 4.11: Configurations at multiple PPA points for TABLA, GeneSys, and Axiline with post-SPR layout (on-chip power only reported).

Classification and localization problem using SVM on TABLA: We exercise an SVM on the WLAN Indoor Localization benchmark [1] dataset. Data preparation consists of the following steps. We first import the WiFi RSSI dataset, the smartphone geomagnetic dataset, the timestamp datafile, and the PointsMapping dataset that contains the placeID-to-XY coordinate mapping. Next, we merge the RSSI dataset with PointsMapping dataset by PlaceID, so that we have XY coordinate and placeID data for RSSI measurements. Finally, we merge the RSSI dataset and Smartphone Geomagnetic dataset together according to the timestamp datafile. The final preprocessed dataset after these operations consists of a table with 11,498 rows and 143 columns that contains all the relevant feature data.

Next, we implement both training and inference for the SVM algorithm in the PolyMath domain-specific language and compile it to the *sr*-DFG representation, followed by a TABLA-backend translation pass, which produces the binary executable as well as necessary configuration and RTL files for TABLA. We consider multiple design implementations of the TABLA platform, and report a set of Pareto-optimal points in Table 4.1.

ResNet50 on GeneSys: We implement ResNet50 on multiple instantiations of GeneSys,

Table 4.1: Training and inference results for the SVM on various TABLA configurations.

#PUs	#PEs/PU	Frequency	Area	Power	Training Runtime	Inference Runtime
8	8	1GHz	2.96mm^2	1.28W	30.6min	0.21ms
8	8	0.25GHz	2.96mm^2	0.29W	122.3min	0.85ms
8	16	1GHz	5.65mm^2	1.90W	26.3min	0.17ms
8	16	0.25GHz	5.65mm^2	0.56W	105.1min	0.68ms

Table 4.2: Inference results for ResNet50 on GeneSys.

PE array size	Bitwidth	Frequency	Area	Power	Execution Time*
16×16	4	1.09GHz	2.0mm^2	0.44W	25.6s
16×16	4	0.27GHz	3.0mm^2	0.10W	89.1s
32×32	8	1.04GHz	8.5mm^2	1.04W	10.0s
64×64	4	0.97GHz	18.9mm^2	1.31W	6.9s

(*reported for 1024 single-stream inference)

each with a different configuration, corresponding to a different size for the PE and SIMD arrays, and different bitwidths. The results for these configurations for single-stream inference, where a query is sent after a previous query is complete, are summarized in Table 4.2. The designs correspond to different Pareto-optimal points, e.g., a design that is optimized for area; a slower design at a low power point; a higher-bitwidth design optimized for classification accuracy; and the largest design that is optimized for speed. The memory interface is assumed to connect to an external HBM2 memory.

Axiline results: Table 4.3 shows the result of implementing Axiline for a training on a set of non-DNN benchmarks. For the logistic regression and SVM benchmarks, two different design points are shown. In all cases, the execution times (which exclude memory fetch times) for Axiline, area, and on-chip power are smaller than those for a platform-based method due to the custom-optimized nature of the engine. The total power is dominated by the off-chip power:in this case, we also assume an HBM2 external memory interface.

Table 4.3: Training results for non-DNN benchmarks on Axiline.

Benchmark	# Features	Frequency	Area	Execution time	On-chip power	Total power
Logistic regression	54	495MHz	0.024mm^2	4.70ms	24mW	0.47W
		500MHz	0.014mm^2	6.98ms	13mW	0.31W
SVM	200	500MHz	0.042mm^2	6.01ms	46mW	3.42W
		497MHz	0.030mm^2	10.05ms	27mW	2.04W
Linear regression	784	492MHz	0.091mm^2	0.37ms	84mW	4.45W

4.6 Conclusion

In this paper, we have presented the VeriGOOD-ML flow for automated ML hardware synthesis. The ONNX representation of an ML algorithm is represented as an IR in the form of a *sr*-DFG, which is then translated to one of the three VeriGOOD-ML engines. Based on the HAG that represents the architecture configuration, the flow translates the IR to an implementation on TABLA (for non-DNN algorithms) or GeneSys (for DNNs), including code generation for the ISA for the corresponding platform. The translation to Axiline is performed directly from the *sr*-DFG. The design then goes through back-end synthesis. Results on a variety of ML algorithms illustrate the efficacy of the flow for a range of ML algorithms, for multiple Pareto points.

4.7 Acknowledgement

Chapter 4 is a partial reprint of the material as it appears in: H. Esmaeilzadeh, S. Ghodrati, J. Gu, S. Guo, AB. Kahng, JK. Kim, **S. Kinzer**, R. Mahapatra, SD. Manasi, E. Mascarenhas, S. Sapatnekar, R. Varadarajan, Z. Wang, H. Xu, B. Yatham, and Z. Zeng, “VeriGOOD-ML: An Open-Source Flow for Automated ML Hardware Synthesis”, int *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, November 1–4 2021, Virtual. The dissertation author was a co-investigator and co-author of this paper.

Bibliography

[1] Geo-magnetic field and WLAN dataset for indoor localisation from wristband and smartphone data set. http://archive.ics.uci.edu/ml/datasets.

[2] Open-sourced specialized computing stack for accelerating deep neural networks.

[3] The OpenROAD project. github.com/The-OpenROAD-Project.

[4] VeriGOOD-ML: Verilog generator, optimized for designs for machine learning. https://github.com/VeriGOOD-ML/public.

[5] Texas instruments c6000tm dsp, 2007.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[7] ACTLab. TABLA source code, 2017.

[8] Muhammad Aurangzeb Ahmad, Carly Eckert, and Ankur Teredesai. Interpretable machine learning in healthcare. In *Proceedings of the 2018 ACM international conference on bioinformatics, computational biology, and health informatics*, pages 559–560, 2018.

[9] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, 2020.

[10] Byung Hoon Ahn, Prannoy Pilligundla, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020.

[11] T. Ajayi, V. A. Chhabria, Mateus Fogaça, S. Hashemi, A. Hosny, A. Kahng, Minsoo Kim, J. Lee, U. Mallappa, Marina Neseem, G. Pradipta, S. Reda, Mehdi Saligane, S. Sapatnekar, C. Sechen, M. Shalan, William Swartz, L. Wang, Zhehong Wang, M. Woo, and Bangqi Xu. Toward an open-source digital flow: First learnings from the openroad project. In *Proc. DAC*, 2019.

[12] Vahide Aklaghi, Amir Yazdanbakhsh, Kambiz Samadi, Hadi Esmaeilzadeh, and Rajesh K. Gupta. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *ISCA*, 2018.

[13] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *ISCA*, 2016.

[14] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. *arXiv*, 2016.

[15] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA, 2010. ACM.

[16] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[17] Mirko Bordignon, Kasper Stoy, and Ulrik Pagh Schultz. Generalized programming of modular robots through kinematic configurations. In *International Conference on Intelligent Robots and Systems*, 2011.

[18] N. Chandramoorthy, G. Tagliavini, K. Irick, A. Pullini, S. Advani, S. A. Habsi, M. Cotter, J. Sampson, V. Narayanan, and L. Benini. Exploring architectural heterogeneity in intelligent vision systems. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2015.

[19] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, 2018. USENIX Association.

[20] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31:3389–3400, 2018.

[21] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.

[22] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[23] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, , Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38:8–20, March 2018.

[24] Lucian Codrescu. Architecture of the hexagon 680 dsp for mobile imaging and computer vision. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–26. IEEE, 2015.

[25] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[26] John D. Davis, Zhangxi Tan, Fang Yu, and Lintao Zhang. A practical reconfigurable hardware accelerator for boolean satisfiability solvers. In *Proceedings of the 45th Annual Design Automation Conference*, New York, NY, USA, 2008. Association for Computing Machinery.

[27] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[28] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A unified backend for targeting fpgas from dsls. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8, July 2018.

[29] Yashar Deldjoo, Mehdi Elahi, Massimo Quadrana, and Paolo Cremonesi. Using visual features based on mpeg-7 and deep learning for movie recommendation. *International journal of multimedia information retrieval*, 7(4):207–219, 2018.

[30] Alberto Delmas, Sayeh Sharify, Patrick Judd, and Andreas Moshovos. Tartan: Accelerating fully-connected and convolutional layers in deep learning networks by exploiting numerical precision variability. *arXiv*, 2017.

[31] Stéfan Van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523, 2011.

[32] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*, 2019.

[33] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[34] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. *Communications of the ACM Research Highlights*, 58(1):105–115, January 2015.

[35] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. ACM, June 2018.

[36] Marco Frigerio, Jonas Buchli, and Darwin G. Caldwell. A domain specific language for kinematic models and fast implementations of robot dynamics algorithms. In *International Workshop on Domain-Specific Languages and Models for Robotic Systems*, 2015.

[37] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[38] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.

[39] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO*, October 2020.

[40] Soroush Ghodrati, Hardik Sharma, Sean Kinzer, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, Doug Burger, and Hadi Esmaeilzadeh. Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic. In *PACT*, 2020.

[41] Grouplens. Movielens dataset.

[42] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A data-movement-aware scheduling language for gpus. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 71–82, 2020.

[43] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, Oct 2016.

[44] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[45] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *ISCA*, 2016.

[46] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July–Aug. 2011.

[47] Kaiming He, X. Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[49] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, and Christopher W Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. *arXiv*, 2018.

[50] John L Hennessy and David A Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[51] Boris Houska, Hans Joachim Ferreau, and Moritz Diehl. Acado toolkit—an open-source framework for automatic control and dynamic optimization. *Optimal Control Applications and Methods*, 32(3):298–312, 2011.

[52] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019.

[53] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.

[54] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[55] Qualcomm Inc. Hexagon nn, 2019.

[56] A. K. Jain, X. Li, P. Singhai, D. L. Maskell, and S. A. Fahmy. Deco: A dsp block based fpga accelerator overlay with low overhead interconnect. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, May 2016.

[57] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[58] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[59] Mina Kamel, Kostas Alexis, Markus Achtelik, and Roland Siegwart. Fast nonlinear model predictive control for multicopter attitude tracking on so (3). In *2015 IEEE Conference on Control Applications (CCA)*, pages 1160–1166. IEEE, 2015.

[60] D. Kehlet. Accelerating innovation through a standard chiplet interface: The advanced interface bus (aib). https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerating-innovation-through-aib-whitepaper.pdf.

[61] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory. In *ISCA*, 2016.

[62] S. Kinzer, J. K. Kim, S. Ghodrati, B. Yatham, A. Althoff, D. Mahajan, S. Lerner, and H. Esmailzadeh. A Computational Stack for Cross-Domain Acceleration. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[63] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators.

In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 296–311, New York, NY, USA, 2018. ACM.

[64] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 68–80, New York, NY, USA, 2018. ACM.

[65] Felipe Kuhne, Joao Manoel Gomes da Silva Jr, and Walter Fetter Lages. Mobile robot trajectory tracking using model predictive control. In *Latin American Robotics Symposium*, 2005.

[66] K. Kunal, M. Madhusudan, A. K. Sharma, W. Xu, S. M. Burns, R. Harjani, J. Hu, D. A. Kirkpatrick, and S. S. Sapatnekar. ALIGN: Open-source analog layout automation from the ground up. In *Proc. DAC*, pages 77–80, 2019.

[67] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

[68] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

[69] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[70] Chris Lattner and Jacques Pienaar. Mlir primer: A compiler infrastructure for the end of moore's law, 2019.

[71] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[72] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, and Hoi-Jun Yoo. Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision. In *ISSCC*, 2018.

[73] M. Lichman. UCI machine learning repository, 2013.

[74] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

[75] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[76] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, Yunji Chen, and Tianshi Chen. Cambricon: An instruction set architecture for neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 393–405. IEEE, 2016.

[77] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 14–26. IEEE, 2016.

[78] The Mathworks, Inc., Natick, Massachusetts. *MATLAB version 9.3.0.713579 (R2017b)*, 2017.

[79] Alexander McCaskey and Thien Nguyen. A mlir dialect for quantum assembly languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 255–264. IEEE, 2021.

[80] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. VTA: an open hardware-software stack for deep learning. *CoRR*, abs/1807.04188, 2018.

[81] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Lianmin Zheng, Eddie Yan, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, July 2019.

[82] G. W. Morris and M. Aubury. Design space exploration of the european option benchmark using hyperstreams. In *2007 International Conference on Field Programmable Logic and Applications*, pages 5–10, 2007.

[83] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin. The microarchitecture of a real-time robot motion planning accelerator. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[84] Erdal Mutlu, Ruiqin Tian, Bin Ren, Sriram Krishnamoorthy, Roberto Gioiosa, Jacques Pienaar, and Gokcen Kestor. Comet: A domain-specific compilation of high-performance computational chemistry. In *Languages and Compilers for Parallel Computing: 33rd International Workshop, LCPC 2020, Virtual Event, October 14-16, 2020, Revised Selected Papers*, pages 87–103. Springer, 2022.

[85] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[86] Huy Nguyen and Minh-Le Nguyen. A deep neural architecture for sentence-level sentiment classification in twitter social networking. In *International Conference of the Pacific Association for Computational Linguistics*, pages 15–27. Springer, 2017.

[87] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. A compiler infrastructure for accelerator generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 804–817, 2021.

[88] Thomas Norrie and Nishant Patil. Google's training chips revealed: Tpuv2 and tpuv3.

[89] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. *ACM SIGPLAN Notices*, 48(6):495–506, 2013.

[90] Nvidia. Dense linear algebra on gpus.

[91] Nvidia. Nvidia cuda fast fourier transform library.

[92] Nvidia. Nvidia nvblas library.

[93] Nvidia. Nvidia toolkit.

[94] Nvidia. Nvidia cuda sdk - image/video processing and data compression, 2008.

[95] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *ISCA*, 2017.

[96] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[97] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017.

[98] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, December 2017.

[99] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016.

[100] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 58–68, New York, NY, USA, 2018. ACM.

[101] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018.

[102] Jacob Sacks, Divya Mahajan, Richard C. Lawson, and Hadi Esmaeilzadeh. Robox: An end-to-end solution to accelerate autonomous control in robotics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 479–490, Piscataway, NJ, USA, 2018. IEEE Press.

[103] Mohammad Samragh, Mojan Javaheripi, and Farinaz Koushanfar. Encodeep: Realizing bit-flexible encoding for deep neural networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 19(6):1–29, 2020.

[104] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based c++ library for linear algebra. In *Journal of Open Source Software*, 2016.

[105] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. Llhd: A multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–271, 2020.

[106] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[107] Hardik Sharma, Jongse Park, Balavinayagam Samynathan, Behnam Robatmili, Shahrzad Mirkhani, and Hadi Esmaeilzadeh. DnnWeaver v2.0: From tensors to FPGAs. In *Proc. Hot Chips*, October 2016.

[108] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. *ISCA*, 2018.

[109] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. pages 97–108, 06 2014.

[110] Guy L. Steele, Jr. Parallel programming and code selection in fortress. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 1–1, New York, NY, USA, 2006. ACM.

[111] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ICML'11, pages 609–616, USA, 2011. Omnipress.

[112] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proc. VLDB Endow.*, 8(11):1214–1225, July 2015.

[113] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[114] Yatish Turakhia, Gill Bejerano, and William J. Dally. Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 199–213, New York, NY, USA, 2018. ACM.

[115] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[116] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[117] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[118] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. Extending halide to improve software development for imaging dsps. *ACM Trans. Archit. Code Optim.*, 14(3), August 2017.

[119] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[120] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems*, ICPADS '12, pages 684–691, Washington, DC, USA, 2012. IEEE Computer Society.

[121] Tim Zerrell and Jeremy Bruestle. Stripe: Tensor compilation via the nested polyhedral model. *arXiv preprint arXiv:1903.06498*, 2019.

[122] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[123] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.