UNIVERSITY OF CALIFORNIA

Los Angeles

Developing, Synthesizing, and Automating
Domain-Specific Accelerators

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

by

Jian Weng

2023

ABSTRACT OF THE DISSERTATION

Developing, Synthesizing, and Automating
Domain-Specific Accelerators

by

Jian Weng
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Anthony John Nowatzki, Chair

The once exponential general purpose processors' (e.g. CPUs) growth of speedup driven by transistor scaling is fading, which urges both industry and academia to find more energy-efficient and performant architecture organization. Therefore, research on accelerators specialized for applications of interest emerges because of their promising speedup and energy saving while retaining flexibility. To design and implement specialized accelerators, intensive human effort is required to study the target applications and determine tradeoffs between performance and cost. In addition, these newly proposed hardware often implies lagging compilation techniques, which hinders the programming productivity. All these facts significantly limits the programmable accelerator adoption.

Moreover, all the prior development effort can hardly be reused in other applicable domains, because the current software/hardware co-designed innovations seldom consider modularity for future integration. Therefore, research projects presented in this dissertation aim at significantly reforming the full-stack reconfigurable accelerator design paradigm: Ideally, each software/hardware co-design feature can be comprised in a universal design space for further accelerator composition so that people no longer build accelerators from scratch. Further, an accelerator can be automatically generated based on the given applications of interest written in a unified high-level programming interface.

To achieve this goal, this dissertation develops the framework, DSAGEN, including an accelerator design space with rich software/hardware co-design features, a compiler targets to accelerators with arbitrary design points within this space, and a design automation algorithm that efficiently searches this space. According to our evaluation, the compiler can robustly target multiple application suites on hardware with arbitrary feature combinations. The framework-generated accelerators can have comparable perf/mm^2 compared with prior handcrafted domain-specific accelerators.

In addition, to demonstrate the wide applicability of our approach, the insights and principles learned along with this goal are also applied to applicable research questions: By deploying the DSAGEN-generated accelerator as a reconfigurable overlay on FPGA, it saves orders-of-magnitude time on compilation and reconfiguration compared with conventional high-level synthesis, while retaining flexibility. This approach suggests that a deeply specialized programmable overlay accelerator can potentially supplement the existing FPGA's high-level programming paradigm. Also, the compilation techniques for spatial architectures developed in DSAGEN can be applied to compiling an emerging instruction paradigm specialized for tensor operations — a productive and extensible compilation framework, UNIT, is presented for these instructions. The extensibility of this framework allows developers to easily integrate new instructions by describing the instruction semantics. High-performance code, that outperforms vendor provided libraries up to 2.2×, for end-to-end inferences can be generated by tensorized rewriting, accompanied with our automated tuning strategies.

The dissertation of Jian Weng is approved.

Glenn D. Reinman

Jens Palsberg

Jingsheng Jason Cong

Anthony John Nowatzki, Committee Chair

University of California, Los Angeles

2023

To my beloved wife, daughter, and parents

who supported me for my Ph.D. degree

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My dissertation cannot be possible without the support and guidance from many people.

First of all, I want to emphasize living a life, and operating a family are more important and much harder than solving any research questions. Therefore, I would like to thank beloved my wife, Xiaoyi Wu, who takes care of my daughter's as well as my life, so that I can focus on my research. Thanks to my daughter, Joan Weng, who practices my patience, and encourages me not to give up when facing difficulties. Thanks to my parents, who patiently listened to me, when I was overpressured during my study.

Next, I would like to thank my advisor, Professor Tony Nowatzki, who always guides me to ask insightful questions, patiently leads me to build research skills, and educates me to have a strong obession to perfection. Thanks for being a well self-examplified advisor. I was always moved by his dedication and enthusiasm on research, which pushes me to refine my work and pursue better research quality.

My committee members have been helpful when polishing my work. Thanks to Jason Cong not only for his valuable suggestions throughout my Ph.D. study, but also for the opportunies he offered, for me to present my research at CDSC, and learn a lot from the RTML project. Thanks to Prof. Jens Palsberg's *Modern Compiler Implemented in Java*, which inspired my enthusiasm on compiler techniques starting from my undergrad. Thanks to Prof. Glenn Reinman. I enjoyed a lot working as a TA of CSAPP with him.

Besides, thanks to all my labmates. Without your help, my research cannot be enabled. I first want to thank Sihao Liu, who took care of the hardest part of my core projects. Without his help, my research can be anything. Thanks to Vidushi Dadu, and Zhengrong Wang for their useful hint on debugging and extending the research infrastructures. Thanks to Dylan's dedication on the infrastructures I developed, so that I can pass my torch to him after graduation. Thanks to Christopher Liu for discussing the merging programming paradigms with me. I also would like to thank my most important external collaborator, Yida Wang at AWS, who directs to explore a brand new research area.

Thanks to all my friends in the meme-picture sharing chat group. Both sharing funny pictures, and seeing the shared funny pictures destress me a lot when struggling for endless works.

Finally, I am so grateful to all my sponsors, including Liang Peng, Nanxin Zhang, Mian Lu, Zhiguo Ge, and Tao Jin from HiSilicon, David Ott, and Pratap Subramanium from VMWare, Mu Li from AWS, NSF (espeically Faculty Early Career Development Program, and Runtime Machine Learning project), and UCLA Graduate Division.

VITA

2017	B. Eng. in Computer Science, ACM Honored Class, SJTU, Shanghai, China
2018, 2019	Applied Scientist Intern, Amazon, Palo Alto, CA.
2021	IEEE Micro Top Picks Honorable Mention.
2021	Teaching Assistant of CS33, UCLA, Los Angeles, CA.
2022	DAC 2022 Reviewer.
2022	MICRO 2022 Subreviewer.
2022	Facebook Fellowship Finalist.
2022	MICRO 2022 Best Paper Runner-up.
2022	Qualcomm Innovation Fellowship Finalist.
2022	UCLA Dissertation Year Fellowship.

PUBLICATIONS

Jian Weng, Sihao Liu, Dylan Kupsh, Tony Nowatzki. Unifying Spatial Accelerator Compilation with Idiomatic and Modular Transformations. IEEE Micro Special Issue on Compiling for Accelerators 2022

Sihao Liu=, **Jian Weng=**, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Lucheng Zhang, Jason Cong, Tony Nowatzki. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. International Symposium on Microarchitecture (MICRO'22) 83/369

Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, Tony Nowatzki. UNIT: Unifying Tensorized Instruction Compilation. International Symposium on Code Generation and Optimization (CGO'21) 31/89

Jian Weng=, Sihao Liu=, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: Synthesizing Programmable Spatial Architectures. International Symposium on Computer Architecture (ISCA'20) 77/421

Jian Weng, Sihao Liu, Zhengrong Wang, Vidushi Dadu, and Tony Nowatzki. A Hybrid Systolic-Dataflow Architecture for Inductive Matrix Algorithms. International Symposium on High-Performance Computer Architecture (HPCA'20) 48/248

Jian Weng, Sihao Liu, Vidushi Dadu, and Tony Nowatzki. DAEGEN: A Modular Compiler for Exploring Decoupled Access Execute Accelerators. Computer Architecture Letters (CAL'19)

CHAPTER 1

Introduction

The free ride of transistor scaling that drives the speedup of general-purpose processors (GPP) for decades is approaching its end. Therefore, hardware specialization has attracted significant attention from both academia and industry because of their promising performance and energy efficiency over general purpose processors.

Developing a specialized accelerator as well as its associated software stack requires non-trivial engineering effort to design and implement. Moreover, prior effort on having such a full-stack implementation can hardly be reused when shifting to a new accelerator. To solve this issue fundamentally, this dissertation presents a new accelerator design and implementation paradigm to not only unify the programmable accelerator design space, but also automate the accelerator design process.

1.1 Motivation

In this section, we first overview the inefficiency of general-purpose processors to motivate specialized accelerators, and explain the issues among the current specialized accelerator design and implementation.

Understanding the Inefficiency of GPP General-purpose processors adopt an imperative execution model. Each application is represented as a sequence of instructions (a.k.a. thread). To execute the application, each instruction undergoes a lengthy pipeline, fetching the instructions, decoding the semantics to prepare operands, executing the computation, accessing memory if needed, and writing the results back. This pipeline maintains generality, but it also suffers from high overhead — only a small portions of the energy and time are consumed by computation itself [66]. In reaction to the increasing demands to performance, mechanisms like scoreboard, Tomasulo, super scalar, speculative execution and out-of-order issue were invented to improve the instruction concurrency by aggressively seeking opportunities of issuing more instructions concurrently to the execution pipeline. These mechanisms not only improve the performance, but also intensify the energy consumption and the chip design complexity to enforce the inter-dependent semantics among the instructions.

Designing and Implementing Specialized Accelerators Because of the inefficiency of the general-purpose pipeline, developers seek to break and reform the software/hardware interfaces and the execution model, so that hardware mechanisms specialized for program behaviors of interest (which will later be referred as "idioms") can be built to achieve orders-of-magnitude speedup and energy saving over GPPs.

In academia, specialized accelerators surge in many application domains which span AI/ML [55, 57, 113, 267, 283, 115, 201, 22, 217, 231, 56, 258, 132, 163, 183, 129, 96], databases [144, 278, 133, 74], system [86, 87], genomics [45, 252, 94, 253], and graph processing [111, 74, 241, 290, 77, 128, 196, 269, 287, 51, 50, 20, 300]. In industry, accelerators are widely adopted in data centers, like Microsoft Brainwave [3], Amazon Inferentia [2], and AQUA [1], and Google TPU [131]. Both Apple and QualComm have kept increasing the number of specialized blocks on their SoCs for image, audio, and signal processing in the past decades.

Widely adopted accelerators not only indicate the popularity of specialization hardware, but also cast a question: do we really need so many accelerators? Specialized accelerators all require a lengthy process to study the program idioms, design specialized mechanisms, balance the tradeoffs between performance and cost, and finally implement the hardware.

Besides designing and implementing the hardware itself, the programming interfaces should also be carefully considered when targeting a programmable design. The software/hardware interface is aggressively reformed to encode the program behaviors of interest, so reinventing the software stack always involves non-trivial effort. When the target domain shifts, this engineering-intensive process is likely to be repeated from scratch.

Therefore, instead of proposing yet another accelerator for another important application domain, the core projects presented in this dissertation rethinks the current accelerator design paradigm, and develops a design automation flow to fundamentally save the accelerator design and implementation effort. Ideally, all the software/hardware co-designed innovations should be unified in a universal design space, and programmed from a stable, domain-neutral, and high-level language. Building a new full-stack programmable accelerators no longer has to start from scratch, and can even be automated. The software stack design and implementation will be elaborated in Chapter 4, and the design automation will be discussed in Chapter 6 and Chapter 7 [273, 166].

General-Purpose Extensions Because of the promise of hardware specialization, and in reaction to the increasing demands on computing power, general-purpose hardware vendors also extend their instruction set architecture, from vectorized extensions (e.g. X86 AVX, and ARM SVE) to tensorized instructions[1] (Intel VNNI [9] and NVIDIA TensorCore [6], to their chips. These instruction extensions can achieve significant speedup on the applications they are designed for, but the compilation techniques for these extended instructions are often lagging, which hinders their adoptions. Application developers have to either call vendor-provided libraries, which lacks flexibility when programming new applications, or program in low-level interfaces, which is unproductive to manage low-level interfaces, when using these newly emerging instructions.

[1] We first coined this word “tensorization” for instructions specialized for tensor operations in work, UNIT: Unifying Tensorized Instruction Compilation [274].

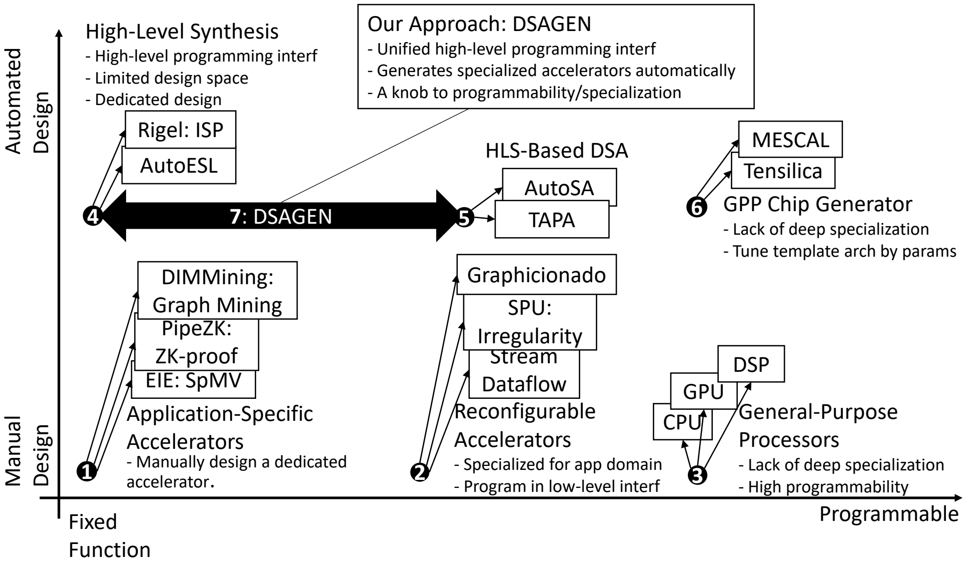

Automated Design
Manual Design
High-Level Synthesis
- High-level programming interf
- Limited design space
- Dedicated design
Our Approach: DSAGEN
- Unified high-level programming interf
- Generates specialized accelerators automatically
- A knob to programmability/specialization
Rigel: ISP
AutoESL
❹
7: DSAGEN
❺
HLS-Based DSA
AutoSA
TAPA
MESCAL
Tensilica
❻
GPP Chip Generator
- Lack of deep specialization
- Tune template arch by params
DIMMining: Graph Mining
PipeZK: ZK-proof
EIE: SpMV
Application-Specific Accelerators
- Manually design a dedicated accelerator.
❶
Graphicionado
SPU: Irregularity
Stream Dataflow
Reconfigurable Accelerators
- Specialized for app domain
- Program in low-level interf
❷
DSP
GPU
CPU
General-Purpose Processors
- Lack of deep specialization
- High programmability
❸
Fixed Function
Programmable

Figure 1.1: A Spectrum of Hardware Specialization Design and Implementation

In this dissertation, we also develop a productive and extensible compilation flow for these tensorized instructions, which will also be discussed in Chapter 8.

1.2 Goal: Automating Specialized Accelerator Design

Specialized accelerator design has long been monopolized by big companies, because of its lengthy design process and intensive human efforts. Multiple teams of people have to iterate over different specialized accelerators at each generation, and the effort spent on accelerators can hardly be reused in other applicable domains, because of the ad-hoc design style and lack of considerations on modularity. As it is shown in Figure 1.1, a spectrum graph for prior specialized hardware design paradigms, many handcrafted prior accelerators, no matter application specific ❶ [113, 292, 78] or programmable ❷ [111, 193, 74] all involve intensive and repeated engineering efforts: Many common program behaviors of interest widely exists in many different applications and application domains; once such a behavior is studied to derive a specialized mechanism, it can potentially be reused for all these applications involved. For example, sparse memory access (e.g. `a[b[i]]`) widely exists in many application domains, including graph processing [74, 111], sparse tensor operation [113], and encryption algorithms [292], but few prior works seek to take advantage of this opportunity to save excessive design effort. Based on this observation, a key insight is that many common hardware/software co-designed features can be extracted and modularized from prior accelerators, and further, a specialized accelerator can be built by composing these features.

In reaction to the high design effort, high-level synthesis (HLS) ❹ [65, 137] and ❺ [59, 268, 34, 238] tries to automate the design of application-specific or domain-specific accelerators by encoding the accelerator design demands in a high-level abstraction. These conventional HLS tools are often application-specific and adopt a very complicated pragma system to expose some excessive low-level details, including memory banking, pipelining, and even some parallelism that is not supposed to be exploited in an imperative and sequential programming interface (e.g. dataflow across multiple loop nests). On the other hand, multiple applications can be captured by domain-specific HLS tools, but their design spaces are limited to the target domain. Another approach to accelerator design automation ❻ [254, 213] is to tune a set of design parameters of a template hardware (e.g. #cores, and LLC capacity) so that the generated accelerator can be better customized for the target application domain. This approach minimizes the accelerator design effort, but adopts a relatively limited design space.

A key goal of this dissertation is to present a fundamentally new accelerator design paradigm to save the excessive design effort of a full stack implementation. By giving the target domain of applications written in high-level programming languages, our framework, DSAGEN ❼ [273], generates a deeply specialized reconfigurable accelerator within a rich design space. Besides design automation itself, this approach also offers a knob to explore the tradeoff between the accelerator performance and the generality by feeding different subsets of applications to the framework, which is shown as a spectrum arrow in Figure 1.1 (other approaches are all points). To achieve this goal, several design demands are imposed to each aspect of this framework:

- **Design Space:** The universal design space should be both *modular* and *general*. By *modular*, it means each software/hardware co-designed feature can be independently integrated to the target hardware. By *general*, this design approach generates domain-specific accelerators, while the target domain itself is agnostic. Therefore, the universal design space should be general enough to cover as many potential domains.

- **Software Stack:** This programming interface should be expressive enough to develop general-purpose applications, and the compiler should be robust to handle any combination of the features. This is the key aspect of this framework, because 1. programs naturally encode the accelerator design demands; 2. the compiler best understands the correspondence between the software behavior and the hardware specialization.

- **Design Space Exploration:** To well leverage the compiler's awareness on the software/hardware co-optimization, an effective and efficient design space explorer should be developed. *Effective* means that it should effectively evolve the candidate hardware by evaluating the software/hardware affinity. Considering the high cost of running the applications and synthesizing the hardware, the explorer should efficiently estimate the software/hardware affinity.

The technical details of this design automation framework, DSAGEN, will be presented in Chapter 6. The acronym, DSAGEN, can be both used as domain-specific accelerator as well as decoupled-spatial architecture. A decoupled-spatial architecture with rich optional specializations will be generated, according to the specialization demands given by the target application domain. The decoupled-spatial architecture paradigm is particularly attractive to build programmable accelerators because of its flexibility and capability on deep specialization, which will be overviewed in the next section.

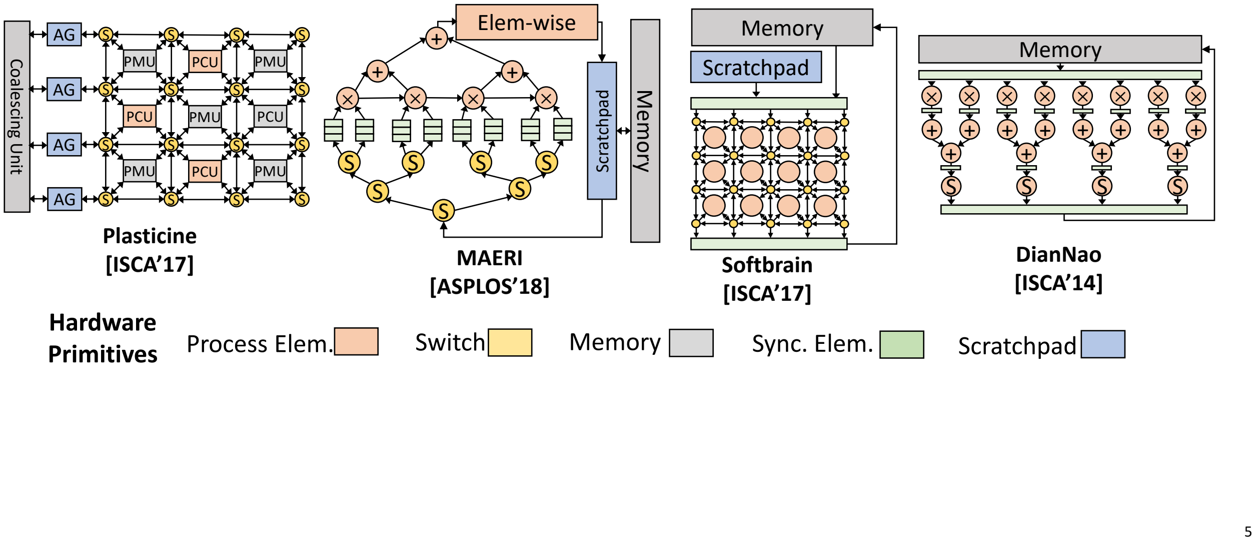


Figure 1.2: Composing prior accelerators by basic primitives

1.3 Decoupled-Spatial Architectures

In this dissertation, we particularly focus on the decoupled-spatial paradigm for our specialized accelerator design automation because of its capability to become deeply specialized for a wide range of potential applications (achieving near-ASIC performance) while retaining flexibility.

This architecture paradigm was studied since the 1980s [236], and has been widely adopted by many prior works [154, 55, 210, 193] as it is shown in Figure 1.2. By decoupling memory access and computation, memory bandwidth can be better utilized while retaining a programmable software pipeline on a spatial architecture. *Spatial* refers to the spatial layout of the architecture's computing resources connected by the on-chip network. These architectures expose the computing resources (also called as processing elements, PE) and on-chip network to their low-level programming interfaces so that instructions can be mapped to computing resources and dependences among the instructions can be routed through the on-chip network. The characterization of spatial architectures will be elaborated in Chapter 2. Built on the top of spatial paradigm, the decoupled data access specialization will be discussed in Chapter 3, and the software stack of this decoupled-spatial paradigm will be explained in Chapter 4.

1.4 Contributions

In this section, the contributions achieved by the published works, including DSAGEN [273], REVEL [276], UNIT [274], and OverGen [166], will be discussed.

Taxonomy of Execution Model of Spatial Architectures The execution model of spatial architecture can be categorized in two dimensions, the timing of the execution and the number of instructions mapped to each single processing elements. These two together enable tradeoffs among performance, programmability, degree of specialization, and hardware cost. We characterize each execution model to determine the scenarios at which each specific execution model is good. This will be in detail discussed in Chapter 2.

Modular ISA for Specialization (Exploration) As discussed above, prior design effort can hardly be reused for future applicable places because of lacking considerations on modularity, and a well-defined design space. We address this issue by modularizing each hardware/software co-designed feature, so that they can be independently integrated/disabled in a target hardware. All the future design effort can be comprised within this design space by either extending existing features or adding new features. This design approach was first practiced in our work REVEL discussed in Chapter 5, and later used as a part of our automated design space exploration in Chapter 6.

A Hardware-Neutral Programming Interface To productively program the accelerator, high-level abstraction is desired. Though the underlying hardware may have an arbitrary set of specialized features, they all follow the same decoupled-spatial paradigm. We develop a unified high-level abstraction for this paradigm. With moderate human hints, the compiler can understand the opportunities of hardware specialization.

Modular Compilation Transformation To bridge the gap between the hardware-neutral programming abstraction and the arbitrary subset of specialized features, the compiler also modularizes hardware-specialized transformation passes. Corresponding to the modular ISA, for each transformation that requires a specific hardware support, there is always a corresponding "fall back" transformation that avoids using this feature. This approach accompanied with the programming interface discussed above will be elaborated in Chapter 4.

A Software/Hardware Co-design Algorithm for Design Space Exploration We build an algorithm to automatically generate a specialized decoupled-spatial architecture deeply specialized for the given set of applications. To efficiently search the design space and guide the candidate hardware evolution, we need to determine software/hardware affinity rapidly by estimating the hardware cost and software performance. Instead of synthesizing the design and profiling each application, we build analytical models to predict these numbers. The hardware cost is estimated by a regression model based on values of each component collected by synthesizing each single hardware component, and the software performance is calculated by estimating the instruction parallelism on the spatial architecture. This design automation algorithm is a key aspect of our design automation framework DSAGEN, which will also be in detail explained in Chapter 6.

A Brand New FPGA Programming Paradigm FPGA programming is known to be difficult and time consuming. Developers either need to manage excessive low-level details by writing register-transfer level (RTL) language or use high-level synthesis (HLS). Though HLS provides a relatively high-level abstraction, it also trades off flexibility and still suffers from long FPGA backend time. In another accomplished work presented in this dissertation, OverGen, we propose a brand new FPGA programming paradigm — auto-generated programmable accelerators are deployed on FPGA as another level of overlay. A unified coarse-grain programmable design not only saves the time of invoking the FPGA backend, but also improves orders-of-magnitude compilation and reconfiguration time. This work is a follow-up of DSAGEN, which will be discussed in Chapter 7.

A High-level and Extensible Programming Interface for Tensorization Hardware vendors extends specialized instructions, including NVIDIA TensorCore, Intel VNNI, and ARM VDOT, for the increasing demands on computing power for tensor operations, but the compilation techniques are lagging. Based on a tensor domain-specific language, TVM [53], a high-level and extensible compilation flow for tensorized instructions is developed, which will be elaborated in Chapter 8.

1.5 Organization

The goal of our proposal is to enable a full-stack ecosystem for designing and implementing specialized accelerators. In the rest of this dissertation, we will first in detail describe the background of the spatial architectures as well as the decoupled access specialization in Chapter 2 and Chapter 3. The software stack of this architectural paradigm will be explained in Chapter 4. Then, we demonstrate a case study of how the principle of composing hardware primitives helps specialized architecture design in our completed work, REVEL, in Chapter 5. Based on the promise of the case study, we will discuss how automated software/hardware co-design of decoupled-spatial architectures is achieved in our completed work, DSAGEN, in Chapter 6. By applying the principles and insights we learned from DSAGEN, we also have works that propose a new FPGA programming paradigm, discussed in Chapter 7, and build an extensible and high-performance compilation flow for the emerging tensorized instruction paradigm, discussed in Chapter 8. Finally, we will discuss the future directions of this area and conclude this dissertation in Chapter 9.

CHAPTER 2

Spatial Architecture Taxonomy

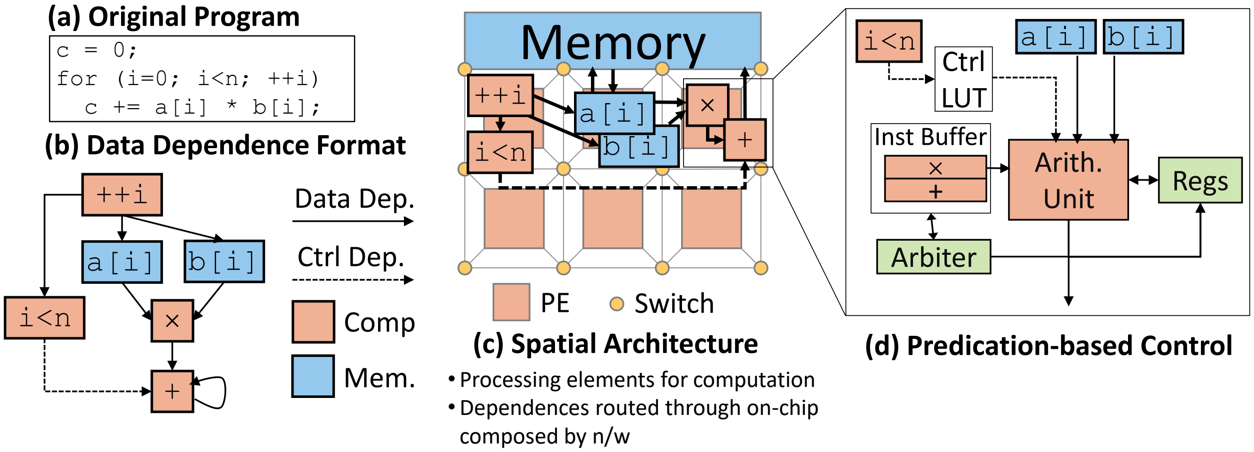
In this chapter, we set up the background of spatial architectures by characterizing spatial architectures in two different aspects, the execution model and the granularity to get poised for the decoupled-spatial paradigm in the next chapter.

2.1 Spatial Execution

In this section, a basic execution model of spatial architectures will be covered, and then the tradeoffs among different spatial architecture variants will be discussed.

Spatial refers to the layout/topology of the computing resources connected and the on-chip network. As it is shown in Figure 2.1(c), the computing resources (also referred as processing elements, PEs) of this spatial architecture adopt a meshed layout connected by on-chip network composed by switches. These computing resources and on-chip network are exposed to the low-level software/hardware interfaces. The processing elements are for the instruction execution, and the on-chip network routes the dependences among these instructions.

To map a dot product program shown in Figure 2.1(a) to a spatial architecture, all the instructions are first represented in a data dependence graph as shown in Figure 2.1(b). Then each aspect of this graph will be isomorphically mapped to a spatial architecture. Each operation is mapped to the processing element, and each edge dependence is routed through the network. Note, because all the instructions are locally buffered in the processing element, conditional execution cannot be supported by branches in imperative thread like



```
c = 0;
for (i=0; i<n; ++i)
  c += a[i] * b[i];
```

Figure 2.1: Mapping a dot product to spatial architecture.

CPU. Instead, as it is shown in Figure 2.1(d), a predication based execution is adopted — besides the operands of addition, an additional predicate is also fed to the processing element to control execution behavior by looking up a control table inside so that the result of accumulation is produced only at the last iteration of the loop.

Mapping a program to a spatial architecture can either be done by manually programming the low-level interfaces or by spatial compilers. There are also semi-automatic works which accepts manually extracted graphs as inputs and schedule them onto the spatial architecture [291, 172, 192, 158]. Such a baseline mapping algorithm will later be elaborated in Section 2.3.

Understanding the Performance Gain over GPP The spatial execution enables a programmable software pipeline. All the instructions are locally buffered in the processing elements so that the overhead of instruction fetching and decoding caused by the imperative pipeline is amortized. Moreover, the performance is longer bound by the size of the instruction issue window. Similarly, the data dependences are routed by the on-chip network instead of accessing a centralized register file, so the resource idle caused by instruction dependences is also minimized.

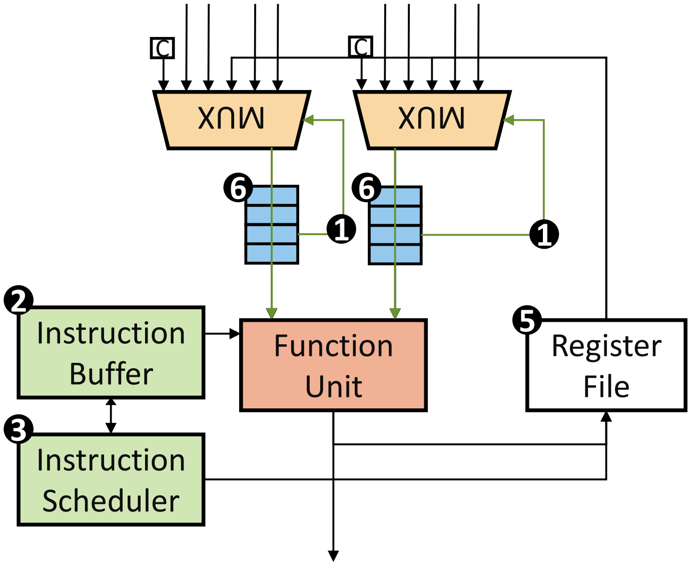


Figure 2.2: The microarchitecture of a spatial processing element (control LUT omitted).

Decoupled-Spatial Paradigm Spatial execution enables high instruction-level parallelism by distributing instructions across the processing elements. On the other hand, because of this distributed nature, each memory access has to be made in scalar and hard to coalesce, which underutilizes the memory bandwidth. This can be solved by the improved *decoupled-spatial* paradigm. The memory access pattern of interest can be encoded in the software/hardware interface to coalesce and better use the memory bandwidth. This architectural paradigm will be discussed in the next chapter.

Variant Taxonomy There are several variants of the spatial execution model, which can be characterized in two dimensions, the instruction buffer size of each PE (either dedicated to one instruction or shared across multiple instructions) and the timing of instruction execution (either statically determined by the compiler or dynamically determined by the runtime). Next, we will stick to Figure 2.2 to explain the micro-architectural support for different execution models and their tradeoffs between cost and specialization. Different execution models are determined by both the on-chip network, and the processing elements. The execution model taxonomy is shown in Figure 2.3, and Table 2.1 gives examples of each

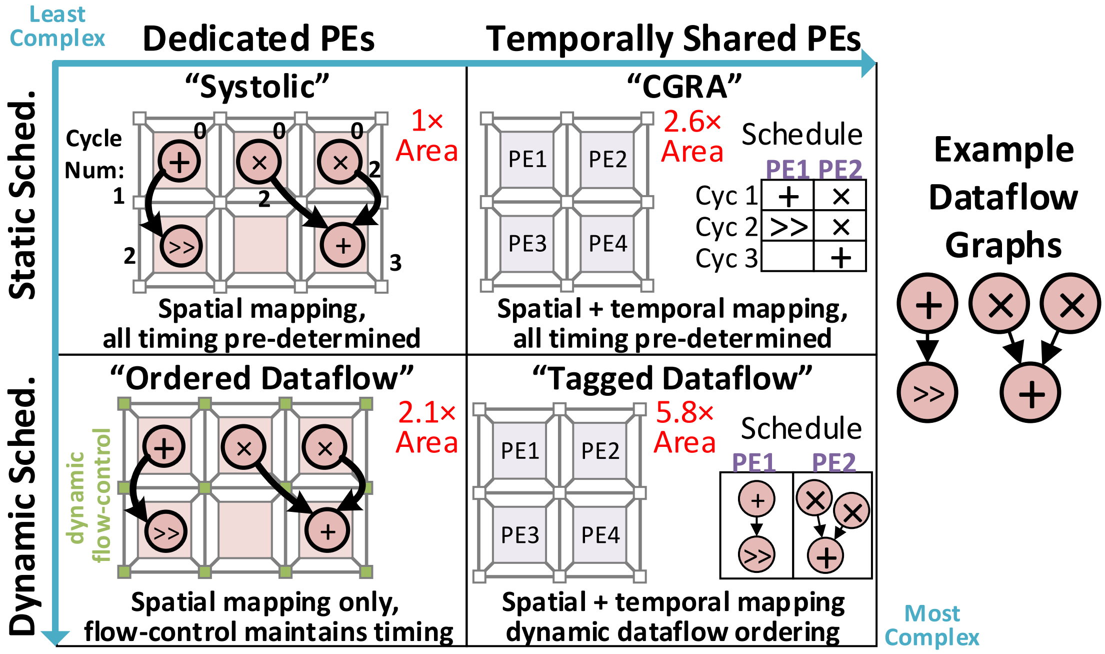


Figure 2.3: A taxonomy of spatial processing elements with an example program.

category. We explain each quadrant:

- **Static/Dedicated (Systolic):** The simplest processing element whose instruction buffer has only one slot ❷, and it has very strict requirement for the timing of the data arrival. For a multi-operand instruction, each operand should reach the processing element at exactly the same time. To meet this timing requirement, the compiler is required to adapt the timing of data arrival using the FIFO buffers (❻ in Figure 2.2). Data can be temporarily stored in these FIFO buffers to lengthen the timing of data arrival. The static timing and small instruction buffer keep the hardware simple by escaping the control logic to check the data availability. On the other hand, since the FU is always active, its power consumption is higher.

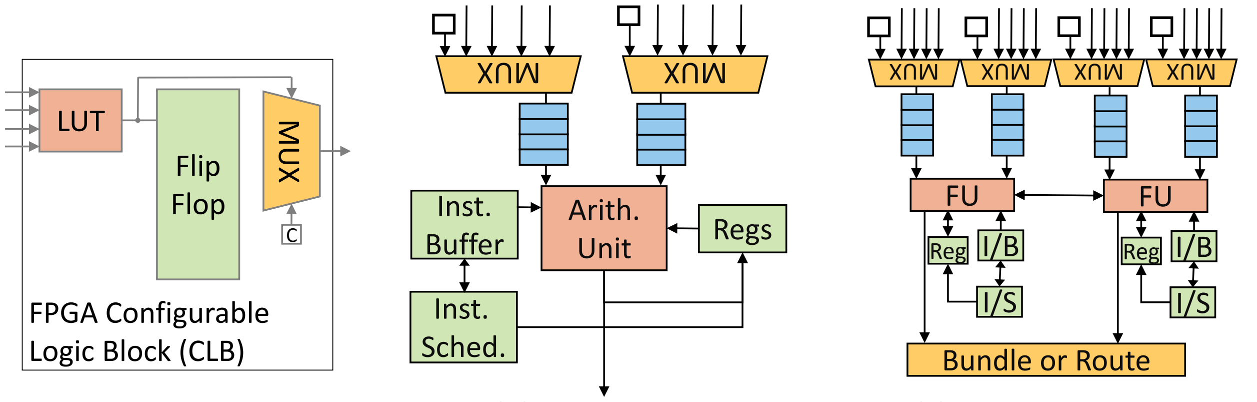
- **Static/Shared (CGRA)** Conventional coarse-grain reconfigurable arrays (the granularity will be explained in Section 2.2) adopts a statically shared execution. Multiple instructions can share computational resources within one single PE, and there is an

	Dedicated PE	**Temporally Shared PE**
Static Scheduling	**“Systolic”** Warp [25], FPCA [69], Softbrain [193], Tartan [179], Piperench [102]	**“CGRA”** MorphoSys [234], Remarc [180], MATRIX [178], ADRES [173], WaveFlow [189]
Dynamic Scheduling	**“Ordered Dataflow”** DySER [104], Q100 [278], Plasticine [210]	**“Tagged Dataflow”** SGMF [263], dMT-CGRA [264], Triggered Instructions [200], WaveScalar [244],TRIPS [44]

Table 2.1: Classifying Architectures within Spatial Taxonomy

arbiter in the PE which determines the instructions to execute in each cycle according to a pre-compiled static schedule. This PE costs additional area and power for a larger instruction buffer ❷ and a larger register file ❺ for intermediate results communicating across instructions, while keeping a simple instruction scheduler ❸. This execution model imposes more challenges to the compiler because larger instruction buffer and register files exponentially enlarge the space of mapping the instructions to the architecture.

- **Dynamic/Dedicated (Ordered Dataflow)** Compared with the static/dedicated PE's, this kind of PEs introduce additional control logic to check the data availability and trigger the instruction execution and data producing/consumption conditionally. Since data producing and consumption rate is determined by the dynamic conditions, the on-chip network should also be extended to support backpressure to dynamically control the data traffic. This dynamic timing offers more semantic flexibility to capture workloads with dynamic data consumption (e.g. key-value join), which will be discussed in Section 4.4.



(a) Fine-Grain PE
- Programmability at bit level
- Look up several bits (typically 1-4)
- Compose CLBs for various functionality

(b) Coarse-Grain PE
- Programmability at word-level
- Arithmetic capability determined after fabrication

(c) Decomposable PE
- Subword FUs can be composed or decomposed for different precision

Figure 2.4: Spatial PEs with different granularities.

- **Dynamic/Shared (Tagged Dataflow)** The PE's with dynamic instruction scheduler ❸ and shared instruction buffer ❷ have the best flexibility and cost the most power and area. Multiple instructions are buffered in the same PE, and their execution is triggered by the data availability. This amortizes the complexity of spatial mapping, because we only need to find a feasible mapping of the instructions. This execution model is useful to achieve high instruction concurrency across code regions with different execution frequency, which will be discussed in Chapter 5.

A spatial architecture can adopt one or more execution model to get specialized execution model to get specialized for code regions with different execution frequency. This will be in detail studied in my completed work REVEL [276], discussed in Chapter 5.

2.2 Datapath Granularity

Another key dimension of the spatial architectures is the datapath granularity, which enables tradeoffs between among flexibility, programming difficulties, and hardware complexity.

FPGAs FPGAs are the most well known and widely adopted spatial architecture, not only as a computing platform but also for hardware design verification. As it is shown in Figure 2.4(a), FPGAs' PEs are most fine-grain (bit-level reconfigurable), and connected by on-chip network. This bit-level reconfigurability is enabled by loading different values to the lookup table (LUT) in each configurable logic block (CLB). This flexibility makes FPGAs powerful computing platform — the on-chip resources can be reconfigured according to the demands of the workloads, including the number, data type, and precision of the functional units. On the other hand, FPGAs' fine-grain fabrics also suffer from high power and area overhead to support this flexibility, and the global routing.

FPGA developers either have to write RTL to manage excessive low-level details — besides the hardware behaviors, they should also carefully manage the LUT synthesis, computing resource placement, and data routing — or write high-level synthesis with limited control to all the hardware aspects. Both programming interfaces have to undergo a lengthy process to synthesize and export the bitstream implementation. One of the works presented in this dissertation aims at re-proposing a new FPGA programming paradigm by deploying another level of coarse-grain overlay onto FPGA, so that both the compilation time, and programming difficulties can be reduced. In addition, excessive effort spent on the FPGA backend can be saved by a unified programmable design. This work will be elaborated in Chapter 7.

Coarse-Grain Spatial Architectures As it is shown in Figure 2.2(b), in the prior sections, we mainly focus on explaining coarse-grain spatial architectures. Compared with FPGAs, the flexibility of coarse-grain architectures is limited at the word-level. Meanwhile, the number of functional units is determined after the fabrication. Systolic arrays like Google TPU [131] and Amazon Inferentia [2] are typical coarse-grain spatial architectures, which are widely adopted in their data centers.

By trading off the full flexibility, coarse-grain spatial architectures enables orders-of-magnitude better energy efficiency, as well as orders-of-magnitude smaller mapping space, which save significant amount of time for finding a mapping with acceptable quality. Next, we will briefly walk through a spatial mapping algorithm.

Decomposability The discussion above mainly sticks on the tradeoffs between the reconfigurability, and the underlying hardware cost determined by the datapath granularity. Decomposability was proposed in SPU [74] to achieve versatile function units while saving the hardware cost by reusing computing resources across different precisions. For example, a 16-bit adder can be composed by two 8-bit adders.

2.3 Spatial Mapping

In this section, we discuss how instructions are mapped onto a coarse-grain spatial architecture. Each instruction is locally buffered in processing elements with required functional units, and the dependences among the instructions are enforced by the on-chip network. All the instructions are first represented in a data dependence graph, and isomorphically mapped to the spatial datapath. Because of the nature of finding a graph isomorphism, the code mapping space is exponentially large, which makes it particularly hard to find the optimal mapping. Prior works either use heuristic based approximate algorithms [172, 171], convert it to an optimization problem [291, 158], or make co-design decisions to balance the mapping difficulties and qualities [192].

To find such an isomorphic mapping with acceptable time budget and quality, we adopt a heuristic-based algorithm, which iteratively adjust the data dependence graph mapping onto the spatial datapath. This algorithm can be described as iteratively applying these three steps to the software/hardware mapping:

Mapping The algorithm randomly picks some unmapped instructions and map them onto a processing element with capable functional unit and with available instruction buffer slots.

Routing Each pair of operand-consumer relationship is inspected. If both are already mapped, a heuristic-based search algorithm [171] is invoked to route the data from source to the destination (either operands to the newly mapped instructions or newly mapped instruction to the consumers). Over provisioning routing resource is allowed during the process of iteration, and will be captured by the objective function.

Timing For the spatial architecture with static execution, we need to calculate the timing of data arrival. This can be abstracted as finding a set of integer solutions for a group of inequalities. Instead of using a linear programming solver, we use an iterative approximate algorithm to adjust the timing of data arrival. After adding up the latency of routing and executing each instruction, we inspect the capability of each delay FIFO, to figure out the bound of the timing when entering each PE. Then we iteratively apply this process on this graph to make the bound tighter by inspecting the bound of downstream and upstream. If the bound overwhelms (the lower bound is larger than the upper bound), we relax the timing of mismatch. When the bounds converge, we greedily assign delay to each node to minimize the latency of computation.

If one of the steps fail, the mapping algorithm may remove some instructions from the existing mapping, and repeat these three steps, until it finds a proper mapping. When iteratively adapting the mapping, we allow some imperfectness like the over provisioned on-chip resources including PE and routing, and this will be captured by our objective function.

Once the mapping algorithm succeeds, the mapping will be encoded in a bitstream format. When configuring a spatial architecture, this bitstream will be loaded to the configuration registers right near each component, including but not limited to the processing elements, delay FIFO, and switches, of the spatial architecture.

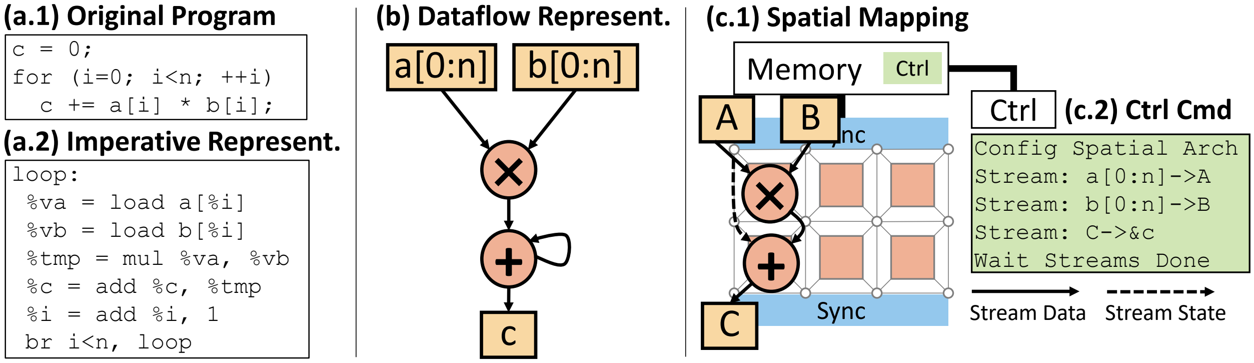
CHAPTER 3

Decoupled-Spatial Architectures

In this chapter, the decoupled-spatial paradigm, the architecture paradigm mainly targeted by this dissertation, will be elaborated. By decoupling memory accesses with particular patterns and offloading them to specialized hardware extensions, the decoupled-spatial paradigm is enabled. Along with explaining the hardware support for this paradigm, the principle of defining design space of the decoupled-spatial architectures will also be briefly covered. Finally, the software/hardware interfaces will be discussed to demonstrate how each hardware specialization is encoded in the ISA.

3.1 Decoupled-Spatial Paradigm

Decoupled-spatial architectures are capable to attain near-ASIC performance under moderate power/area overhead while retaining programmability by specializing the *decoupled* memory access and computation respectively. *Decoupled* memory operations are encoded in control commands (will be explained in the Section 3.3) and executed by a specialized memory engine. Computational instructions are offloaded onto a *spatial* architecture, which has already been explained in the last chapter. Since memory accesses are already decoupled and executed on a specialized memory unit, unlike the processing elements shown in Figure 2.1 (in last chapter), processing elements no longer need a capability of requesting the memory. This keeps the computing resources simple to better utilize the on-chip resources. Next, we overview the decoupled-spatial paradigm through the example shown in Figure 3.1.



```
c = 0;
for (i=0; i<n; ++i)
  c += a[i] * b[i];
```

```
loop:
 %va = load a[%i]
 %vb = load b[%i]
 %tmp = mul %va, %vb
 %c = add %c, %tmp
 %i = add %i, 1
 br i<n, loop
```

```
Config Spatial Arch
Stream: a[0:n]->A
Stream: b[0:n]->B
Stream: C->&c
Wait Streams Done
```

Figure 3.1: Example of decoupled-stream program and hardware mapping.

We still use the same application, a simple dot product, shown in Figure 3.1(a). To map it to the decoupled-spatial paradigm, the memory access and computation are decoupled and represented in a dataflow graph format shown in Figure 3.1(b). Different from Figure 2.1, the memory accesses are no longer represented in arrays slices (e.g. `a[0:n]` and `b[0:n]`) rather than scalars (e.g. `a[i]` and `b[i]`). Each element in the array slices will flow through this graph to execute the program.

To support this execution paradigm, the computational portion of the dataflow graph will be isomorphically mapped to a **spatial architecture** as shown in Figure 3.1(c.1) using the algorithm described in Section 2.3. The **host controller** will issue a configuration command to the spatial architecture to load the spatial bitstream. **Synchronization FIFOs** are injected between the memory and the spatial architecture, which not only serve as the operand interfaces of the spatial architecture, but also reason a timing of data arrival when adopting static timing (discussed in Section 2.1).

As discussed in the last chapter, the scalar memory accesses generated by each processing element can hardly be coalesced, which leads to redundant memory requests to the same cacheline that waste the memory bandwidth. The decoupled-spatial paradigm seeks to get specialized for this program behavior by encoding the memory access with particular patterns in coarser grain format. A set of memory accesses under a nest of loops can be encoded in a constant number of control commands. These commands are issued from the host controller to the **memory controller**. These encoded memory accesses are called "stream"s. The memory controller arbitrates the execution of each stream, and maintains a state machine to generate coalesced memory accesses.

Moreover, our decoupled-spatial architecture has a unique mechanism specialized for the loop-dependent program behaviors. The dot product accumulates the result of multiplication, and the behavior of the adder is different when entering and leaving the loop — reset the accumulate register to 0 when entering and produce the result when leaving. This behavior is controlled by the state of the "stream"s. Both the memory stream and stream state will be explained in Section 3.2.2.

All the coordination, including spatial configuration, data access, and phase synchronization, are encoded in specialized control commands and issued by a host controller as shown in Figure 3.1(c.2). Specific to our implementation, we use a light-weighted RISC-V core as the controller so that when the decoupled-spatial architecture cannot accelerate a program portion, it can still fall back to the host execution. All these bold keywords are the key components of a decoupled-spatial architecture. The functionality and their associated design parameters will be discussed in Section 3.2.

Key Tradeoffs Besides amortizing the imperative overhead, including instruction fetch, decode, and result write back, as discussed in Section 2.1, the decoupled spatial paradigm coalesces the memory access to better utilize the memory bandwidth. The instruction-level parallelism among computational instructions are maximized when the data bandwidth sustains.

On the other hand, this paradigm even more aggressively reforms the execution model. The memory access and computation are executed on different hardware aspects, which imposes more challenges on the software stack development. The compilation techniques for this paradigm will be explained in Chapter 4.

3.2 Hardware Support and Design Parameters

Each aspect of a decoupled-spatial architecture is already overviewed when going through the example shown in Figure 3.1. In this section, we discuss the functionality of each component as well as their associated design parameters.

3.2.1 Spatial Architecture

The spatial architecture paradigm was already covered in the Chapter 2. Under the context of decoupled-access execution, spatial architectures are composed by following components:

- **Processing Elements (PE)** are the key to instruction execution. Instructions are buffered and executed locally in each PE. The design parameters of a processing element are the timing of execution, size of the instruction buffer and register file, and arithmetic capability, which were already discussed in Section 2.1. As discussed above, PEs for decoupled-spatial architectures no longer need the capability of requesting the memory which keeps each PE simple.

- **On-Chip Network** is composed by switches, which is the key of the programmability of the datapath. Because of the distributed nature of the PE's, spatial architectures do not have a centralized register file to store the intermediate results, so all the dependences among the instructions are routed through this network. The key design parameter of the switches are the routing degrees and the capability of back pressure. The back pressure, accompanied with certain PE parameters, enables dynamic execution, which were already discussed in Section 2.1.

- **Delay FIFO's** serve as buffers to delay the timing of data arrival. This is critical for the PE's with static execution to match timing of data arrival. The key design parameter is the size of the buffer, which enables a tradeoff between the mapping difficulties and the cost of on-chip resources [192].

- **Synchronization Elements** are essentially FIFO buffers injected between components without a static timing (e.g. memory with caches, and dynamic PE's) and static elements (static PE's). The purpose is to synchronize multiple inputs of a computation instance to enable static reasoning about the timing of all dependent events. The coordination is implemented by a programmable ready logic, which can be configured statically to allow multiple synchronization elements to fire data together. In the rest of this proposal, we these elements are also called "ports". These ports also serve as the interfaces of the spatial architecture to enable data communication.

Spatial architectures are attractive to our accelerator design automation goal because of its flexibility and scalability. Computing power can easily be scaled up or down by integrating different components. With different design parameters adopted for each component, spatial architectures can have different execution models, which enables a tradeoff among cost, flexibility, and specialization, as it was already discussed in Chapter 2.1.

3.2.2 Coordination and Data Access

Control Host: As mentioned above, coordination across different hardware aspects are done by a host controller. The memory access streams, inter-port data communication, and synchronizations are encoded in hardware intrinsics issued by the control host to a stream dispatcher. The stream dispatcher arbitrates the order of the streams, and dispatches streams to corresponding modules. In our works, we use a single-issue RISC-V core as our control host [193] so that when it comes to something that cannot be specialized by the decoupled-spatial paradigm, it can still fall back to general-purpose execution. All coordination commands are enabled by extending the RISC-V ISA, and the supported commands will be elaborated in Section 3.3.

Memory Controller/Stream Engine: Instead of reading data in scalars or vectors, decoupled-spatial paradigm aggressively encode memory accesses in more coarse-grain pattern, streams. Streams are defined by a group of memory accesses under loop nests. Both dense (e.g. `a[i*n+j]`) and sparse (e.g. `a[b[i]]`) memory accesses are supported.

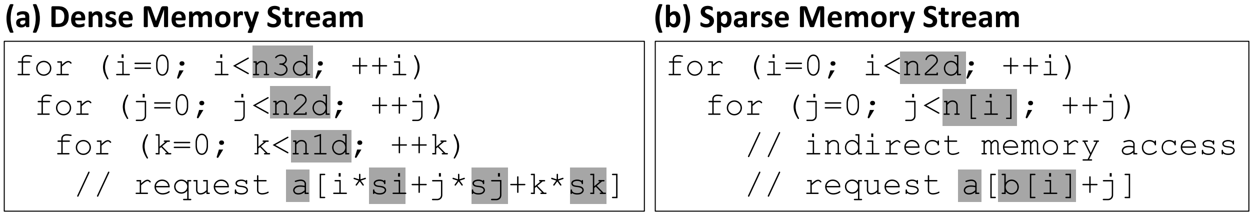
By extracting several essential parameters of the memory access pattern, these encoded memory streams can be encoded (encoding will be explained in Section 3.3) in control commands issued from the host controller to the memory stream engines. How these essential parameters of the memory access pattern are analyzed and encoded in stream commands will be discussed in Chapter 4. Memory stream engines are associated with different memory modules (e.g. scratch memory or main DRAM).

3.3 Decoupled-Spatial ISA

The ISA of decoupled-spatial architecture we target in this work is derived from Softbrain [193]. Table 3.1 shows the specification of hardware intrinsics for configuration, data access, and synchronization. All our works use this ISA as a prototype. This ISA extension and modularization will be discussed in Chapter 4, Chapter 5, and Chapter 6.

Spatial Architecture Configuration The spatial mapper will encode the mapping in a bitstream for spatial architecture configuration. The `Config` loads a configuration bitstream with a specific `size` from the given `addr`ess.

Memory Streams The memory accesses with particular patterns are encoded in specialized hardware intrinsics. Figure 3.2 shows the capability of supporting both dense and sparse streams. Those shadowed parameters, including the iterations of each loop, the coefficient of each loop variable, and the array address, are essential to encode the streams. Because of the limited number of operands in an instruction, we use a `ConfigReg` instruction to load these shadowed values in instructions, and use `LinearStream` and indirect `IndirectStream` to specify the associated information, including flags and ports, of memory streams.



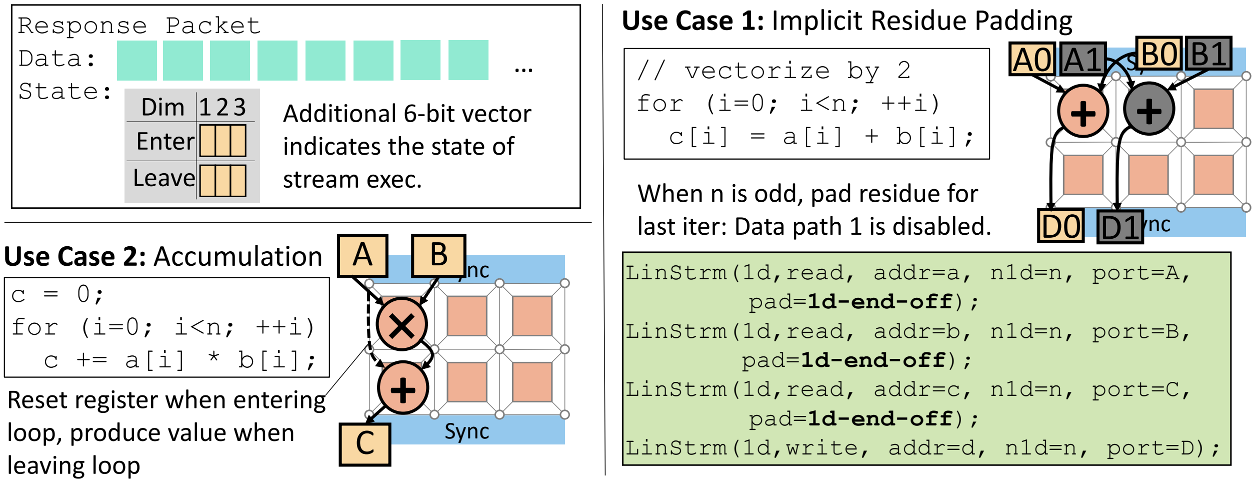
(a) Dense Memory Stream

```
for (i=0; i<n3d; ++i)
 for (j=0; j<n2d; ++j)
  for (k=0; k<n1d; ++k)
   // request a[i*si+j*sj+k*sk]
```

(b) Sparse Memory Stream

```
for (i=0; i<n2d; ++i)
  for (j=0; j<n[i]; ++j)
    // indirect memory access
    // request a[b[i]+j]
```

Figure 3.2: The semantics of the streams encoded. Up to 3-d dense stream is supported, and the sparse access pattern is widely used in graph processing.



```
Response Packet
Data:
State:
```

Dim	1 2 3
Enter	□□□
Leave	□□□

Additional 6-bit vector indicates the state of stream exec.

Use Case 1: Implicit Residue Padding

```
// vectorize by 2
for (i=0; i<n; ++i)
  c[i] = a[i] + b[i];
```

When n is odd, pad residue for last iter: Data path 1 is disabled.

```
LinStrm(1d,read, addr=a, n1d=n, port=A,
         pad=1d-end-off);
LinStrm(1d,read, addr=b, n1d=n, port=B,
         pad=1d-end-off);
LinStrm(1d,read, addr=c, n1d=n, port=C,
         pad=1d-end-off);
LinStrm(1d,write, addr=d, n1d=n, port=D);
```

Use Case 2: Accumulation

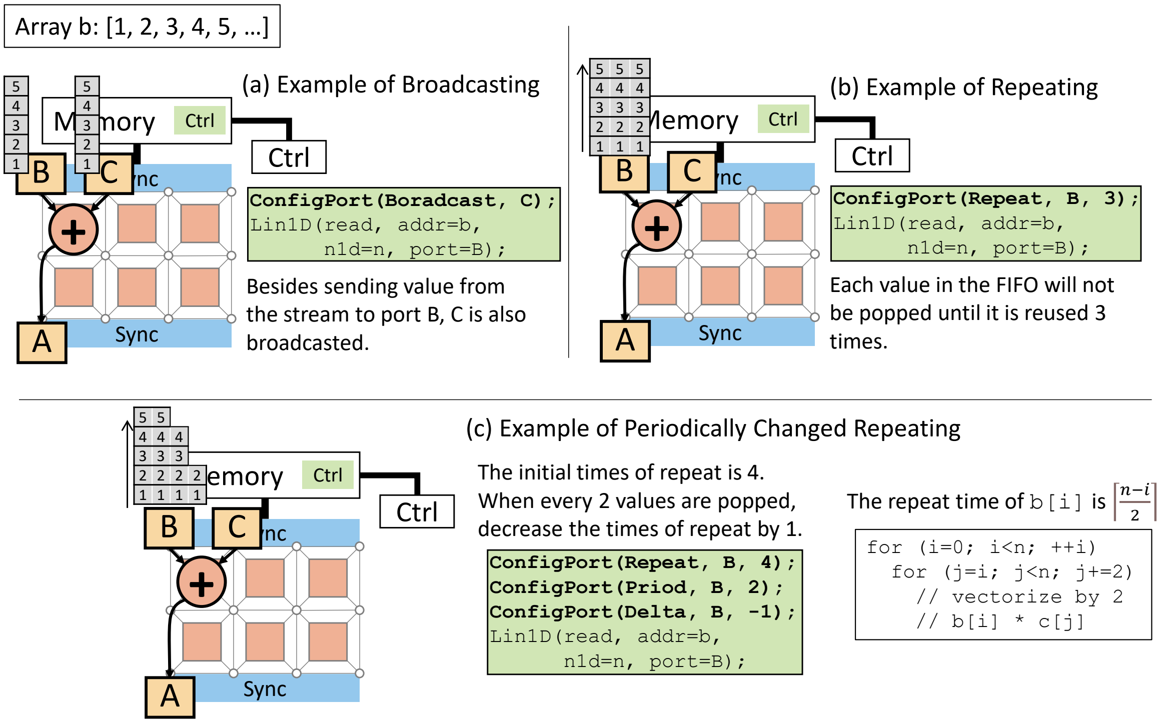
```
c = 0;
for (i=0; i<n; ++i)
  c += a[i] * b[i];
```

Reset register when entering loop, produce value when leaving loop

Figure 3.3: Stream state encoding, and two typical use cases of stream state.

The memory stream derives the other additional mechanisms, data generation stream, and implicit data padding, which will next be explained.

Data Generation Stream Essentially, the encoded dense memory stream generates a sequence of memory addresses and request data using these memory addresses. Instead of requesting data, we also support directly exposing these values to applications. This is useful when a sequence of constant values are required (e.g. passing a constant coefficient to the computation).



(a) Example of Broadcasting

```
ConfigPort(Boradcast, C);
Lin1D(read, addr=b,
      n1d=n, port=B);
```

Besides sending value from the stream to port B, C is also broadcasted.

(b) Example of Repeating

```
ConfigPort(Repeat, B, 3);
Lin1D(read, addr=b,
      n1d=n, port=B);
```

Each value in the FIFO will not be popped until it is reused 3 times.

(c) Example of Periodically Changed Repeating

The initial times of repeat is 4. When every 2 values are popped, decrease the times of repeat by 1.

```
ConfigPort(Repeat, B, 4);
ConfigPort(Priod, B, 2);
ConfigPort(Delta, B, -1);
Lin1D(read, addr=b,
      n1d=n, port=B);
```

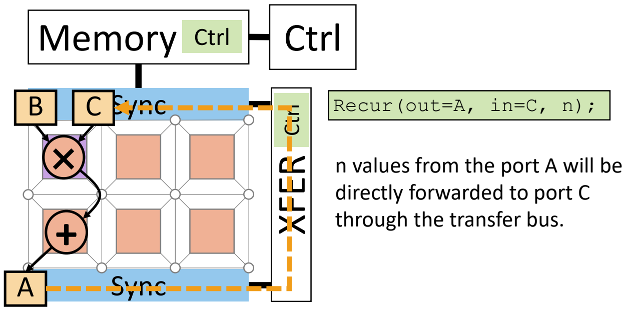
The repeat time of `b[i]` is $\lceil \frac{n-i}{2} \rceil$

```
for (i=0; i<n; ++i)
  for (j=i; j<n; j+=2)
    // vectorize by 2
    // b[i] * c[j]
```

Figure 3.4: Examples of `ConfigPort`

Stream State Besides having data itself, the response packet requested by memory streams also associates a 6-bit metadata, to record the state of the stream execution to indicate if it is entering/leaving a loop as it is shown in Figure 3.3. This is useful for implicit data padding accumulation — loop residue happens at the last iteration of the innermost loop, and data accumulation needs to be reset when entering and value should be produced when leaving. This mechanism was first developed in Chapter 5, because the inductive nature of the workloads make it impossible to find a perfect tiling.

Configure Port As aforementioned, coordination FIFO's are injected to serve as interfaces between components with different timing. These coordination FIFO's are exposes as interfaces so that the data accessed by the data stream intrinsics can be forwarded to the target operands specified by the ports. The ports can also be used to control the data consumption rate. Figure 3.4 shows how each field of a port affects the behavior of a read stream:



`Recur(out=A, in=C, n);`

n values from the port A will be directly forwarded to port C through the transfer bus.

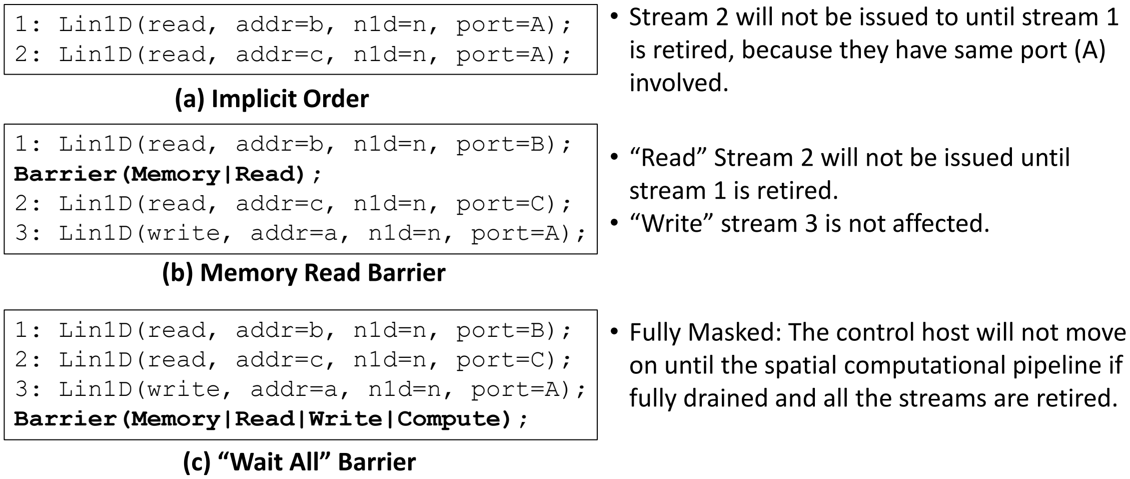
Figure 3.5: A simple example of data recurrence.

- (a) shows that for a read stream destined to port B, it will also be duplicated to port C because of broadcast configuration.
- (b) shows that each element of this stream is repeated 3 times when feeding to the spatial architecture.
- (c) is an even more advanced case of (b), the repeat time can be changed periodically. This is especially useful for inductive kernels, which will be explained in Section 5.3.1.

Recur This instruction is especially useful when we want to use some value right after it is produced from the output port. Instead of writing this value to the memory and synchronize the whole accelerator, the value can be directly forwarded from output port to the input port through a recurrence bus as it is shown in Figure 3.5

Stream Order The order of data streams, including memory stream, recur stream, and data generation stream, can either be implicitly or explicitly enforced. By implicitly, as it is shown in Figure 3.6(a), if there is no barrier between two streams, and these two streams involve a same port, the order of these two streams will be their order in the command thread.

As it is shown in Figure 3.6(b)&(c), barriers are provided to enforce the order among a type of streams. A bitmask operand is provided in a barrier instruction is to specify



```
1: Lin1D(read, addr=b, n1d=n, port=A);
2: Lin1D(read, addr=c, n1d=n, port=A);
```

- Stream 2 will not be issued to until stream 1 is retired, because they have same port (A) involved.

(a) Implicit Order

```
1: Lin1D(read, addr=b, n1d=n, port=B);
Barrier(Memory|Read);
2: Lin1D(read, addr=c, n1d=n, port=C);
3: Lin1D(write, addr=a, n1d=n, port=A);
```

- “Read” Stream 2 will not be issued until stream 1 is retired.
- “Write” stream 3 is not affected.

(b) Memory Read Barrier

```
1: Lin1D(read, addr=b, n1d=n, port=B);
2: Lin1D(read, addr=c, n1d=n, port=C);
3: Lin1D(write, addr=a, n1d=n, port=A);
Barrier(Memory|Read|Write|Compute);
```

- Fully Masked: The control host will not move on until the spatial computational pipeline if fully drained and all the streams are retired.

(c) “Wait All” Barrier

Figure 3.6: The examples of stream orders with and without `Barrier`

the “type” of streams to be enforced — read, write, memory, scratchpad, and recurrent. Any instructions with the specified type after the barrier will not be issued to until all the instructions with the involved types before the barrier retire. Figure 3.6 shows the examples on how barriers affect the order of stream execution.

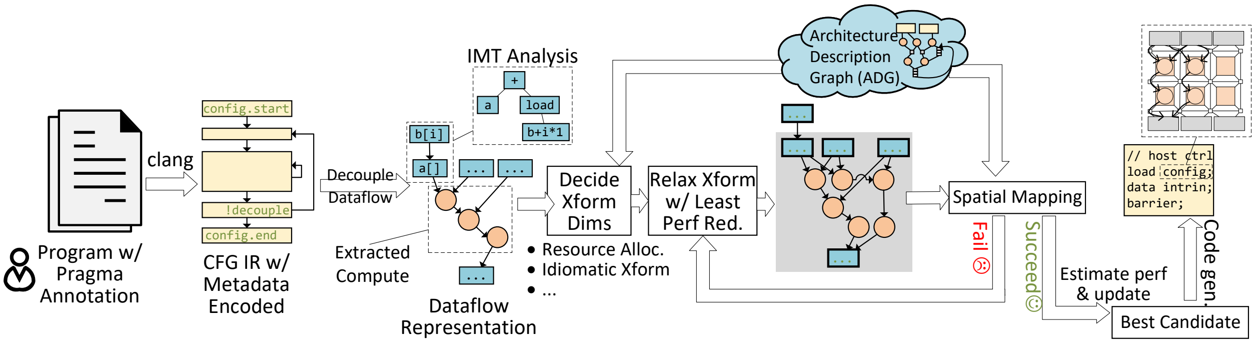
Summary: The decoupled-spatial ISA encodes program behaviors of interest in the exposed software/hardware interfaces which enable a deep degree of specialization, while retaining high flexibility. In the rest of this dissertation, we will discuss how we develop accelerators, software stacks, and automate the design process under this paradigm.

Instruction	Parameters
`Config`	`addr`: The address of the bitstream `size`: The size of the bitstream
`ConfigPort`	`port`: The port involved `field`: Broadcasting, repeat, period, or delta. `value`: The value of the selected field to be set.
`ConfigReg`	`reg`: The register ID. Refer the shadowed boxes in Figure 3.2 for more details. `value`: The value load to the register.
`LinearStream`	`dimension`: The number of loop levels `port`: The source/destinationport `mem`: DMA or scratchpad `action`: Generate value, or access pointer `op`: If accesses pointer, read or write. `padding`: Implicitly pad the datapath to fill the lanes at the end of this stream.
`IndirectStream`	`index_port`: The data source of the index `op`: read, write, or atomic binary update `port`: The port involved as destination or source of operand
`Recur`	`src`: The output port of the data source `dest`: The in port of the destination n: The number of elements to forward
`Barrier`	`mask`: The streams involved by this barrier

Table 3.1: The hardware intrinsics and the argument list of each intrinsic.

CHAPTER 4

Idiomatic Compilation for Decoupled-Spatial Architectures



```
// host ctrl
load config;
data intrin;
barrier;
```

Figure 4.1: An overview to decoupled-spatial compiler.

In the last chapter, we have already explained the decoupled-spatial architecture and demonstrated the promise of this paradigm. This chapter presents the software stack for the decoupled-spatial architectures, from high-level abstraction to compiler transformation. The compiler is overviewed in Figure 7.2. Since the spatial mapping is already well explained in Section 2.3, this chapter will mainly focus on decoupling the computation and the memory accesses, as well as mapping the program idioms to the hardware specialization.

Two key design demands are imposed for building such a software stack: First because of the generality of the decoupled-spatial paradigm, the programming interface should also be general-purpose. Therefore, instead of using a domain-specific language, a general-purpose language, C, is adopted. However, C is originally designed for imperative execution, To avoid excessive compilation work while retaining a high-level programming interface, the compiler relies on additional information provided by developers by annotating the program with several pragmas, which will be explained in Section 4.1.

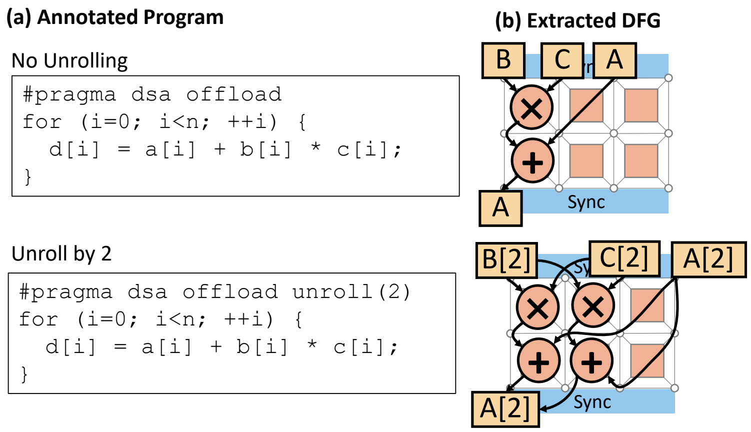
Second, the main goal of this dissertation is to automate specialized accelerator design, so instead of targeting a specific hardware, the compiler should be able to robustly translate code for various hardware within the design space. To stay robust across any combination of the optional features of a decoupled-spatial architecture, the compiler should be aware of the underlying hardware capability and make transformation decisions accordingly. The underlying hardware capability is encoded in an architecture description graph, which will be in detail explained in DSAGEN (Chapter 6), and we develop modularized compiler transformations — each optional-feature-oriented transformation will also have a fallback transformation that escapes this feature. This guarantees the success of compilation at cost of the performance when a specialized hardware support is unavailable. This will be explained in Section 4.3 and Section 4.4.

This compiler is evaluated on three application suites MachSuite [216], DSP from REVEL [276], and Xilinx Vitis to demonstrate both its performance and robustness across hardware with various feature combinations.

4.1 Programming Interfaces

To fulfill the first design demand while retaining sufficient information to the compiler to understand the opportunity of taking advantage of hardware specialization, we adopt C as our high-level programming interface. To compile a piece of C program to a decoupled-spatial architecture, it should undergo:

1. Determining the code regions to be offloaded to the decoupled-spatial paradigm.
2. Determining the concurrency among the offloaded code regions.
3. Decoupling the computational instructions and the memory operations.



```
#pragma dsa offload
for (i=0; i<n; ++i) {
  d[i] = a[i] + b[i] * c[i];
}
```

```
#pragma dsa offload unroll(2)
for (i=0; i<n; ++i) {
  d[i] = a[i] + b[i] * c[i];
}
```

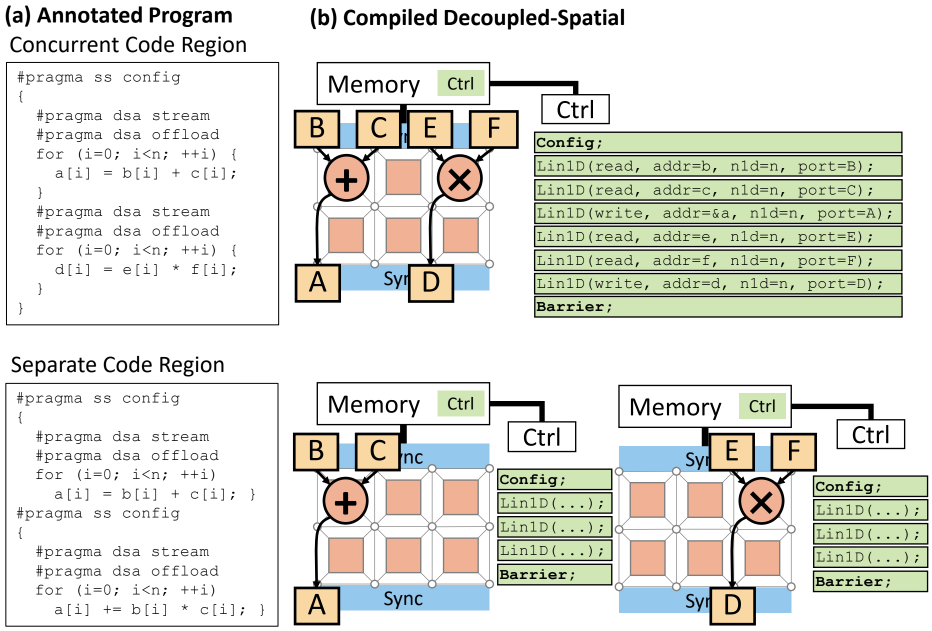
Figure 4.2: Tuning different resource occupation through the unrolling degree

- Analyzing the decoupled memory operations, and encode them in hardware intrinsics.
- Extracting the dataflow graph of the decoupled computational instructions and map them onto the spatial architecture.

4. Removing‘ the instructions offloaded onto the decoupled-spatial paradigm.

To find a balance between the human intervention and the compiler effort, we leave step 1, and 2 for application developers to decide. Figure 4.2 show an example of an annotated program. These additional information can be provided by annotating the original C program with our extended pragmas:

`#pragma dsa offload`: This pragma annotates either a compound statement or a loop, which indicates the computational instructions in the annotated code region will be offloaded onto the spatial architecture. As it is shown in Figure 4.2, when application development, users are allowed to use an optional clause `unroll` to specify the unrolling degree to tune the resource occupancy. Different loop behaviors (e.g. elementwise and reduction) may lead to different unrolling transformation, which will be covered in Section 4.3. When the unrolling degree is -1, the unrolling degree is enumerated by the compiler.



Concurrent Code Region

```
#pragma ss config
{
  #pragma dsa stream
  #pragma dsa offload
  for (i=0; i<n; ++i) {
    a[i] = b[i] + c[i];
  }
  #pragma dsa stream
  #pragma dsa offload
  for (i=0; i<n; ++i) {
    d[i] = e[i] * f[i];
  }
}
```

```
Config;
Lin1D(read, addr=b, n1d=n, port=B);
Lin1D(read, addr=c, n1d=n, port=C);
Lin1D(write, addr=&a, n1d=n, port=A);
Lin1D(read, addr=e, n1d=n, port=E);
Lin1D(read, addr=f, n1d=n, port=F);
Lin1D(write, addr=d, n1d=n, port=D);
Barrier;
```

Separate Code Region

```
#pragma ss config
{
  #pragma dsa stream
  #pragma dsa offload
  for (i=0; i<n; ++i)
    a[i] = b[i] + c[i]; }
#pragma ss config
{
  #pragma dsa stream
  #pragma dsa offload
  for (i=0; i<n; ++i)
    a[i] += b[i] * c[i]; }
```

```
Config;
Lin1D(...);
Lin1D(...);
Lin1D(...);
Barrier;
```

```
Config;
Lin1D(...);
Lin1D(...);
Lin1D(...);
Barrier;
```

Figure 4.3: Different instruction concurrency caused by different compound body annotation

`#pragma dsa stream`: This pragma annotates a loop, which indicates the memory operations under this loop level are restricted, so that memory operations can be safely decoupled and encoded in hardware intrinsics.

`#pragma dsa config`: This pragma annotates a compound body, which indicates all the code regions annotated by `offload` will be concurrent on the spatial architecture. As it shown in Figure 4.3, we can use this pragma to explore the concurrency among offloaded code regions.

Next, we overview how the compiler take advantage of these annotated pragmas to transform a C program to decoupled-spatial architectures.

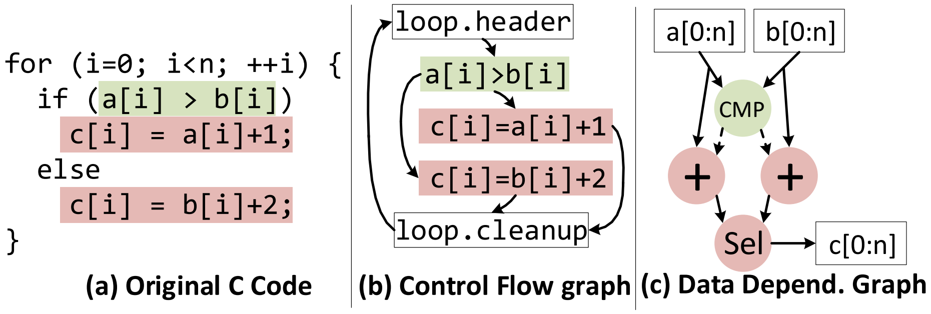
4.2 Transformation Overview

Decoupling Computation and Memory Access After annotation, the compiler starts with the code region annotated by the `offload` to gather the code regions for decoupled-spatial rewriting. After gathering these regions, the computation and memory access are sliced for analysis.

Data Dependence Transformation: After slicing the memory operations, we transitively traverse the dependence relation among the remaining computational instructions to gather a data dependence graph. As discussed in Section 2.1, the branch conditions cannot be executed by instruction fetching on a spatial archtiecture, so they will be converted to predications of the instructions in those blocks [91]. We use a variant of the transformation to program dependence graphs. Figure 4.4 (a) shows that the original code and the control flow graph with two branches, and Figure 4.4 (c) shows the transformed data dependence graph — both branches will be executed, and a selector will select the proper value according to the result of the comparison.

Analyzing the Program Idioms: Our compiler respectively analyzes the program idioms among the computation and memory access so that program behaviors of interests can be captured by hardware specialization. We develop our unique data structure, idiomatic memory tree (IMT), to represent results of the memory access analysis, which will be discussed in the next section.

Modular Transformation After the analysis, the compiler understands the correspondence between program idioms and hardware specialization, but not every optimization can be applied because of the availability of the optional specialized features, and the computing resource occupancy. To fulfill the second requirement of the compilation (robustness across any decoupled-spatial architectures), the compiler inspects the capability of the underlying hardware, and explores a set of legal transformations by enumerating the tunable dimensions (e.g. unrolling degree of each offloaded region). To guarantee the success of transformation, a modularized strategy is adopted. For a code rewriting specialized for a hardware feature, there will always be a fallback transformation that escapes this feature when unavailable.



```
for (i=0; i<n; ++i) {
  if (a[i] > b[i])
    c[i] = a[i]+1;
  else
    c[i] = b[i]+2;
}
```

(a) Original C Code (b) Control Flow graph (c) Data Depend. Graph

Figure 4.4: Transform the control dependence to data dependence

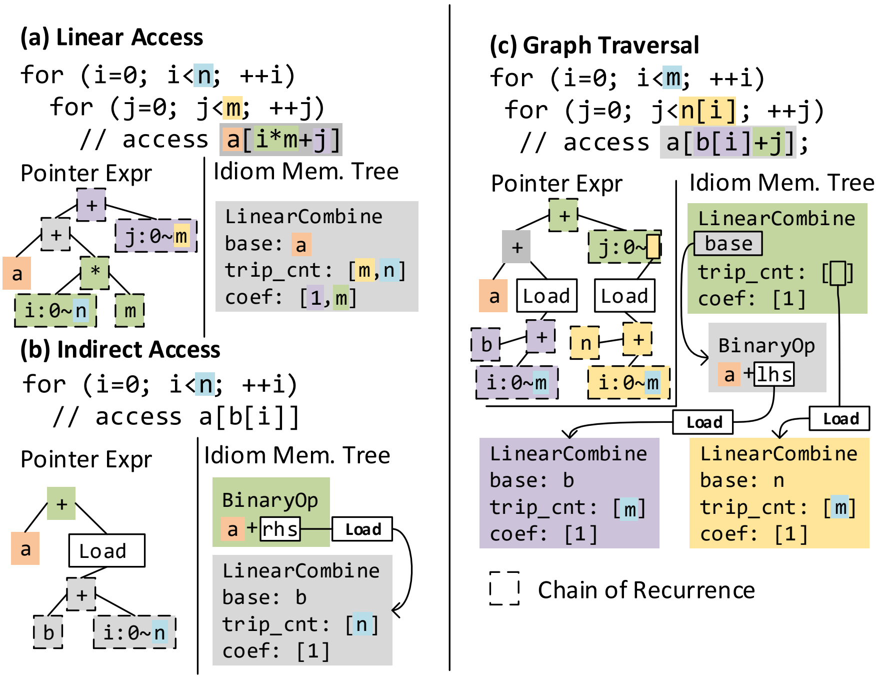
Code Generation These extracted computation instructions will be removed from their original site, since they are offloaded onto the spatial architecture. The decoupled memory operations will be hoisted and encoded in data intrinsics discussed in Section 3.3. After these transformations, some instructions without consumers, and empty loop bodies will be introduced. Therefore, we will invoke the compiler's generic `O3` optimization passes to cleanup the dead code.

4.3 Idiomatic Analysis Transformation

After extracting the memory access and computation, the compiler for the decoupled-spatial idiomatic ISA must find the best-suited idiom that is applicable to a program region. Here, we explain a set of broadly applicable idiom analysis and transformations for memory access and computation.

4.3.1 Decoupled Memory Access

Expressing memory in terms of predefined coarse grain idioms — i.e. streams — has many benefits, including better-utilizing memory bandwidth and reducing control instructions and core-accelerator communication. To aid mapping/optimization on streams, we develop the idiomatic memory tree (IMT).



```
for (i=0; i<n; ++i)
  for (j=0; j<m; ++j)
    // access a[i*m+j]
```

```
for (i=0; i<n; ++i)
  // access a[b[i]]
```

```
for (i=0; i<m; ++i)
 for (j=0; j<n[i]; ++j)
  // access a[b[i]+j];
```

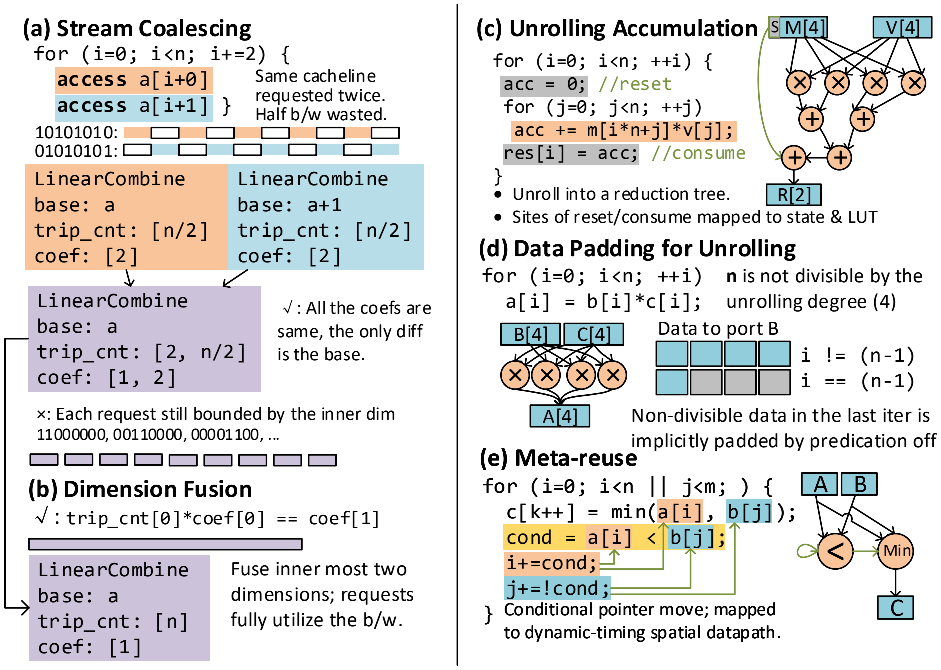
Figure 4.5: An example of analyzing memory access pattern.

Idiomatic Memory Tree Prior works on program idiom analysis under loops, like chain of recurrence (CR)[1][85], mainly focus on analyzing the expressions of loop variables and invariants, and have limited support on memory-dependent expressions.

To store and analyze the idiomatic memory behaviors in a structured way, our insight is that complicated program behaviors can be composed by a set of simple primitive idioms. To explain, Figure 4.5(c) shows a complicated graph traversal example. This program behavior is composed of an affine pattern in the inner loop and an indirect pattern in the outer loop. Next, we introduce IMT nodes that capture primitive behaviors.

LinearCombine Figure 4.5(a) shows an example of linear memory accesses. Dashed boxes annotate the sub-expressions that can be analyzed by CR. By walking through these expression nodes, we can extract the coefficient of each loop variable and represent the pointer

[1] Our implementation uses LLVM's ScalarEvolution for CR.



(a) Stream Coalescing

```
for (i=0; i<n; i+=2) {
  access a[i+0]
  access a[i+1] }
```

Same cacheline requested twice. Half b/w wasted.

```
LinearCombine
base: a
trip_cnt: [n/2]
coef: [2]
```

```
LinearCombine
base: a+1
trip_cnt: [n/2]
coef: [2]
```

```
LinearCombine
base: a
trip_cnt: [2, n/2]
coef: [1, 2]
```

√ : All the coefs are same, the only diff is the base.

×: Each request still bounded by the inner dim 11000000, 00110000, 00001100, ...

(b) Dimension Fusion

√ : `trip_cnt[0]*coef[0] == coef[1]`

```
LinearCombine
base: a
trip_cnt: [n]
coef: [1]
```

Fuse inner most two dimensions; requests fully utilize the b/w.

(c) Unrolling Accumulation

```
for (i=0; i<n; ++i) {
 acc = 0; //reset
 for (j=0; j<n; ++j)
  acc += m[i*n+j]*v[j];
 res[i] = acc; //consume
}
```

- Unroll into a reduction tree.
- Sites of reset/consume mapped to state & LUT

(d) Data Padding for Unrolling

```
for (i=0; i<n; ++i)
  a[i] = b[i]*c[i];
```

n is not divisible by the unrolling degree (4)

Non-divisible data in the last iter is implicitly padded by predication off

(e) Meta-reuse

```
for (i=0; i<n || j<m; ) {
  c[k++] = min(a[i], b[j]);
  cond = a[i] < b[j];
  i+=cond;
  j+=!cond;
}
```

Conditional pointer move; mapped to dynamic-timing spatial datapath.

Figure 4.6: Generic program idioms

expression, `a[i*n+j]`, in a linear combination format:

$$\sum_k \mathrm{i}_k \times \mathrm{coef}_k + \mathrm{base}$$

BinaryOp Figure 4.5(b) shows a pointer expression (`a[b[i]]`) without loop variable directly involved in the operands, which cannot be analyzed by CR. Thus, we use a `BinaryOp` node to wrap this node and recursively analyze both operands.

Load Continuing with the example shown in Figure 4.5(b), a memory load is involved on the rhs operand, which indicates an indirect memory operation. We wrap the memory load with a `Load` node, and recursively analyze the pointer expression of this memory load. In this case, the load pointer can be handled by CR, and yields a `LinearCombine` node.

The IMT is useful for generic optimizations (explained next) and to choose the right set of idioms for code generation.

Fusing Stream Idioms Because memory requests are issued at line granularity, stride-access wastes bandwidth. We address this through two transformations:

Stream Coalescing Figure 4.6(a) shows if we generate streams for `a[i]` and `a[i+1]` separately, these two streams will request each cacheline twice. Streams which have the same coefficients, and where the base differs by one, are coalesced by appending a new dimension (see purple box, Figure 4.6(a)).

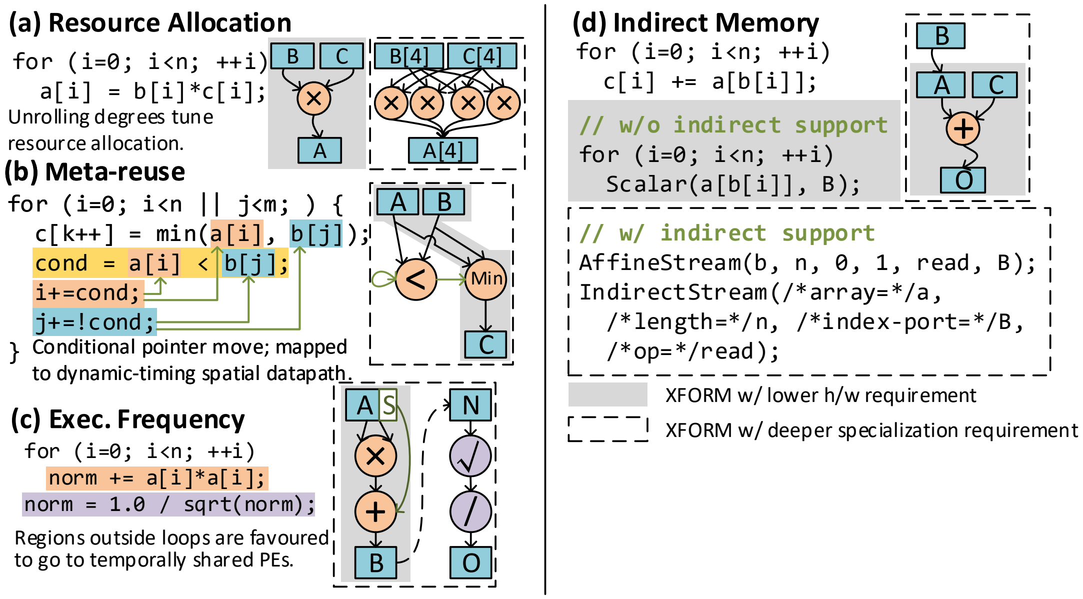
Dimension Fusion Continuing with Figure 4.6(b), after appending a new dimension, two 1-d streams become a 2-d stream with two continuous dimensions. Therefore, our compiler will fuse subsequent stream dimensions when their outer dimension's coefficient equals the coefficient times the trip count of the inner loop.

In the code generation phase, our compiler matches the analyzed and optimized IMT on idiomatic ISA to fill in parameters. A `LinearCombine` node indicates affine memory access, so it encodes as many as possible linear dimensions supported on the hardware. A `BinaryOp` node with a `Load` involved indicates an indirect memory access, so the compiler extracts the

4.3.2 Computational Idioms

Accumulator Accumulation manifests as a loop carried dependence that involves itself — see Figure 4.6(c). To specialize for this idiom, the intermediate accumulated results can be stored implicitly in the instruction (later allocated to a PE register), and the data output/accumulator-reset is controlled by a stream *state* metadata of an operand stream.

Data Padding As shown in Figure 4.6(d), the trip count of the innermost dimension can be indivisible by the unrolling degree, which normally requires loop peeling for the final iterations. Our compiler specializes for idiom by generating memory streams with different padding flags according to their consumers. For elementwise operations, stream data is padded to fill with invalid data to predicate off the unused datapath. For accumulator


(a) Resource Allocation

```
for (i=0; i<n; ++i)
  a[i] = b[i]*c[i];
```

Unrolling degrees tune resource allocation.

(b) Meta-reuse

```
for (i=0; i<n || j<m; ) {
  c[k++] = min(a[i], b[j]);
  cond = a[i] < b[j];
  i+=cond;
  j+=!cond;
}
```

Conditional pointer move; mapped to dynamic-timing spatial datapath.

(c) Exec. Frequency

```
for (i=0; i<n; ++i)
  norm += a[i]*a[i];
norm = 1.0 / sqrt(norm);
```

Regions outside loops are favoured to go to temporally shared PEs.

(d) Indirect Memory

```
for (i=0; i<n; ++i)
  c[i] += a[b[i]];
// w/o indirect support
for (i=0; i<n; ++i)
  Scalar(a[b[i]], B);
// w/ indirect support
AffineStream(b, n, 0, 1, read, B);
IndirectStream(/*array=*/a,
  /*length=*/n, /*index-port=*/B,
  /*op=*/read);
```

▨ XFORM w/ lower h/w requirement

┄ XFORM w/ deeper specialization requirement

Figure 4.7: Modular transformations with fallbacks.

operations, stream data is padded with zeros to avoid adding extra control/muxing.

Meta-reuse Figure 4.6(e) describes an algorithm for merging two lists, by repeatedly increasing the iterator of the smaller value. To specialize for this, input elements elements are popped/reused according to the result of comparison. This shortens the dependence chain on control, but causes data-dependent consumption rate on the spatial architecture; thus, dynamic-scheduled hardware elements are required.

4.4 Modular Compilation

To stay robust across accelerators, we use modular transformations with fallbacks for each idiom and develop an exploration algorithm to quickly converge to the optimal set of transformations.

4.4.1 Modular Transformations with Fallbacks

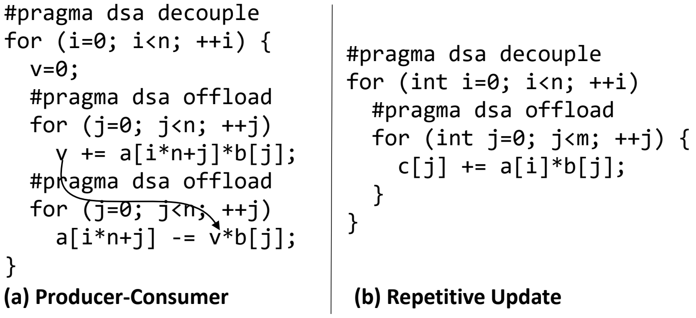
As mentioned above, our compiler guarantees the success of compilation by modularizing transformations. Fallback transformations (shadowed regions in Figure 4.7, and listed in Figure 4.8 and Figure 4.9) either uses less resources or non-idiomatic hardware features when optional features unavailable. Here, we give six examples:

High Allocation→Low allocation Executing more loop iterations simultaneously is faster but requires more resources. Thus, a knob to resource allocation is the unrolling degree, as shown in Figure 6.4(a).

Meta-reuse→Host Execution Meta-reuse control requires dynamic PEs/switches which are more expensive. A fallback is to execute related instructions on the control core: the comparison, the green arrows, and the memory streams `a[i]` and `b[i]`.

Temporal PEs→Dedicated PEs/Host Execution Figure 6.4(c) shows that `norm` computation is out of the loop body and involves two expensive operations. Offloading instructions with lower execution frequency to dedicated PEs leads to low resource utilization, so these instructions are favored to go to temporally shared PEs. If shared PEs are not available, the compiler first falls back to dedicated PEs if enough are available, then falls back to the control core.

Indirect Memory→Scalar Memory A memory operation pointer expression involves another memory operation indicates an indirect memory access. Accelerators without indirect-memory specialization require a fallback transformation: The compiler will perform indirect accesses as a series of scalar accesses (single element stream); this requires an order-of-magnitude more core/accelerator communication. Figure 6.4(d) shows the different generated stream commands depending on whether indirect memory is available.



```
#pragma dsa decouple
for (i=0; i<n; ++i) {
  v=0;
  #pragma dsa offload
  for (j=0; j<n; ++j)
    v += a[i*n+j]*b[j];
  #pragma dsa offload
  for (j=0; j<n; ++j)
    a[i*n+j] -= v*b[j];
}
```

(a) Producer-Consumer

```
#pragma dsa decouple
for (int i=0; i<n; ++i)
  #pragma dsa offload
  for (int j=0; j<m; ++j) {
    c[j] += a[i]*b[j];
  }
}
```

(b) Repetitive Update

Figure 4.8: Two idioms benefit from data recurrence

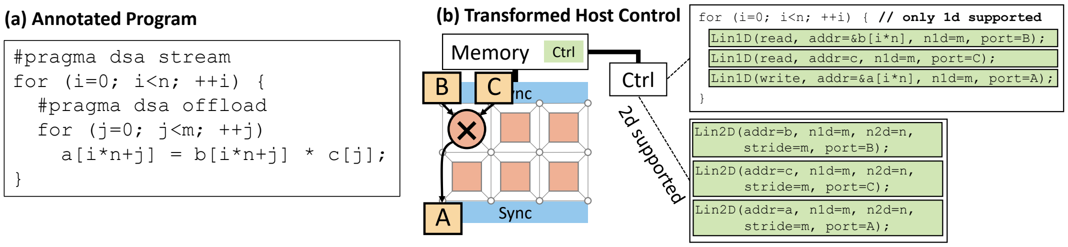
Data Recurrence→Barriers: Enforcing the data depencence by stalling the whole system harms the performance. We find two idioms are quire useful to avoid this when the recurrence bus is available:

- **Producer-Consumer:** Consider the example shown in Figure 4.8(a) where a value `v` produced by the first offloaded region is consumed by the second. Instead of waiting for the first region write this value to a host register and sending it back to the spatial architecture, we can use the transfer bus directly forward the value from the output port to the input port.

- **Repetitive In-Place Update:** As it is shown in Figure 4.8(b), the array `c` is updated repetitively. To avoid unnecessary memory traffic and synchronization, our compiler will try to convert the in-place read/write of `c` to transfer bus communication. The compiler first analyzes the length of this stream to see if it overwhelms the size of on-chip buffer. If so, this loop will be tiled by the maximum factor that fits in the on-chip buffer so that we can take advantage of data recurrence in each tiled portion.

If not, we fall back to memory read/write enforced by barriers.

High-Dimension Stream→Lower-Dimension Streams As it is shown in Figure 4.9, if the stream engine of the underlying decoupled-spatial architecture is not capable to handle high-dimension streams, the compiler will generate a group of low-dimension streams



```
#pragma dsa stream
for (i=0; i<n; ++i) {
  #pragma dsa offload
  for (j=0; j<m; ++j)
    a[i*n+j] = b[i*n+j] * c[j];
}
```

```
for (i=0; i<n; ++i) { // only 1d supported
  Lin1D(read, addr=&b[i*n], n1d=m, port=B);
  Lin1D(read, addr=c, n1d=m, port=C);
  Lin1D(write, addr=&a[i*n], n1d=m, port=A);
}
```

```
Lin2D(addr=b, n1d=m, n2d=n,
      stride=m, port=B);
Lin2D(addr=c, n1d=m, n2d=n,
      stride=m, port=C);
Lin2D(addr=a, n1d=m, n2d=n,
      stride=m, port=A);
```

Figure 4.9: Different stream encoding caused by different loop annotation

wrapped by loops to achieve the same semantics. This guarantees the success of compilation while introducing more overhead to the host controller.

4.4.2 Transformation Space Exploration

As aforementioned, idioms and modular features are dimensions of a transformation space. When it comes to multiple loops and program behaviors of interest, the transformation space grows exponentially. Meanwhile, invoking the spatial scheduling algorithm is expensive, making it difficult to try every possibility. Therefore, we adopt a somewhat-greedy search algorithm. The basic steps are:

1. The compiler first determines all the explorable dimensions for each concurrently mapped program region, according to relevant transformations and the hardware capability in the ADG.

2. For each region and dimension, the compiler "relaxes" it (e.g. reduce the unrolling degree of one of the loops, or disable indirect memory encoding), and then estimate the performance reduction caused by the relaxation based on the expected ILP and memory bandwidth of any streams.

3. The transformation with the least performance reduction is applied, and the transformed IR is fed to the spatial scheduler for hardware mapping. If it fails, eliminate the transformation point and go to step 2; else, the the feasible transformation point

is returned.

Because this algorithm enumerates transformation points in a roughly decreasing order of the ideal performance, the first successful mapping is likely to have the best performance. In addition, we also adopt a region-balance strategy to prune this space — resource allocations where a low execution frequency code region has a higher resource allocation than a higher-frequency code region will be skipped.

4.5 Methodology

Software Stack: The compiler is implemented by extending the Clang frontend for pragma parsing, and LLVM for ISA extension and IR transformation.

Benchmarks We select 9 from MachSuite, 9 from Xilinx Vitis, and 5 DSP workloads, each with their own prevalent program idioms. The data type, size, and computation intensity of each is shown in Table 4.1.

Hardware Setup We choose the *AMD EPYC 7702P* as our CPU baseline. All the benchmarks run on this are compiled by `gcc -O3`.

Accelerators are generated with DSAGEN [273]. The accelerator controller is a single-issue RISCV core with extended ISA. Both the AMD CPU and the accelerator have the same L1/L2 cache size (32KB, 512KB) and bandwidth (64B/cycle). The AMD CPU has nearly 24GB/s DRAM bandwidth, and the accelerator has 20GB/s memory bandwidth.

We start with a general accelerator with full specialized features and the spatial architecture is 5×5 (16 dedicated multiplier PE's, 8 dedicated adder PE's, one temporally shared PE with full arithmetic capability) mesh-topology.

We then use DSAGEN to auto-generate accelerator targets for each benchmark suite with three different degrees of specialization.

- *Capability Specialization (Cap)* indicates the architecture adopts all the specialized fea-

Workloads	crs/ellpack	gemm	nw	stcl-2d	stcl-3d	viterbi	merge	radix
Size	496×4	64^3	128^2	34^2	34^3	140×64	2048	2048
DType	f64	i64	i64	i64	i64	f64	i64	i64
Op/DRAM	0.16	4	0.13	1.99	1.14	0.1	0.5	0.06
Feat.	Ind. Mem	Basic	Basic	Basic	Basic	Basic	Dyn. Timing	Ind. Mem

Specialization		Cap.	+FU	+Topo
Area (mm2)	FU	0.15	0.11	0.08
	SW	0.03	0.02	0.02
	Port	0.03	0.03	0.02
	Spad	0.12	0.12	0.12
	Total	0.36	0.31	0.27

(a) MachSuite

Workloads	acc	acc-sqr	acc-wei	grey	blur	cha-ext	conb	drvt	vecmax
Size	128^2×4								
DType	i16								
Op/DRAM	0.16	0.33	0.66	0.4	0.5	n/a	0.5	0.5	0.16
Feat.	Basic								

Specialization		Cap.	+FU	+Topo
Area (mm2)	FU	0.15	0.05	0.05
	SW	0.02	0.02	0.02
	Port	0.03	0.02	0.02
	Spad	0.04	0.04	0.00
	Total	0.28	0.18	0.13

(b) Vitis

Workloads	chol	fft	mm	qr	solver
Size	48^2	2048	32^3	48^2	48^2
DType	f64	f32x2	f64	f64	f64
Op/DRAM	3.07	6.8	1.33	4.08	0.24
Feat.	Shared PE	Basic	Basic	Shared PE	Basic

Specialization		Cap.	+FU	+Topo
Area (mm2)	FU	0.15	0.08	0.07
	SW	0.03	0.02	0.02
	Port	0.02	0.02	0.02
	Spad	0.04	0.04	0.04
	Total	0.29	0.20	0.19

(c) DSP

Table 4.1: Benchmark specification and generated hardware characteristics.

tures required and a generic mesh topology. Both floating point and integer functional units are included.

- *FU Specialization (+FU)* indicates the unused functional units will be trimmed off.
- *Topology Specialization (+Topo.)* indicates the topology (the connectivity of hardware components) is specialized to the applications.

The bottom of Table 4.1 shows the area breakdown of these accelerators. All these numbers are synthesized by Synopsys DC @28nm.

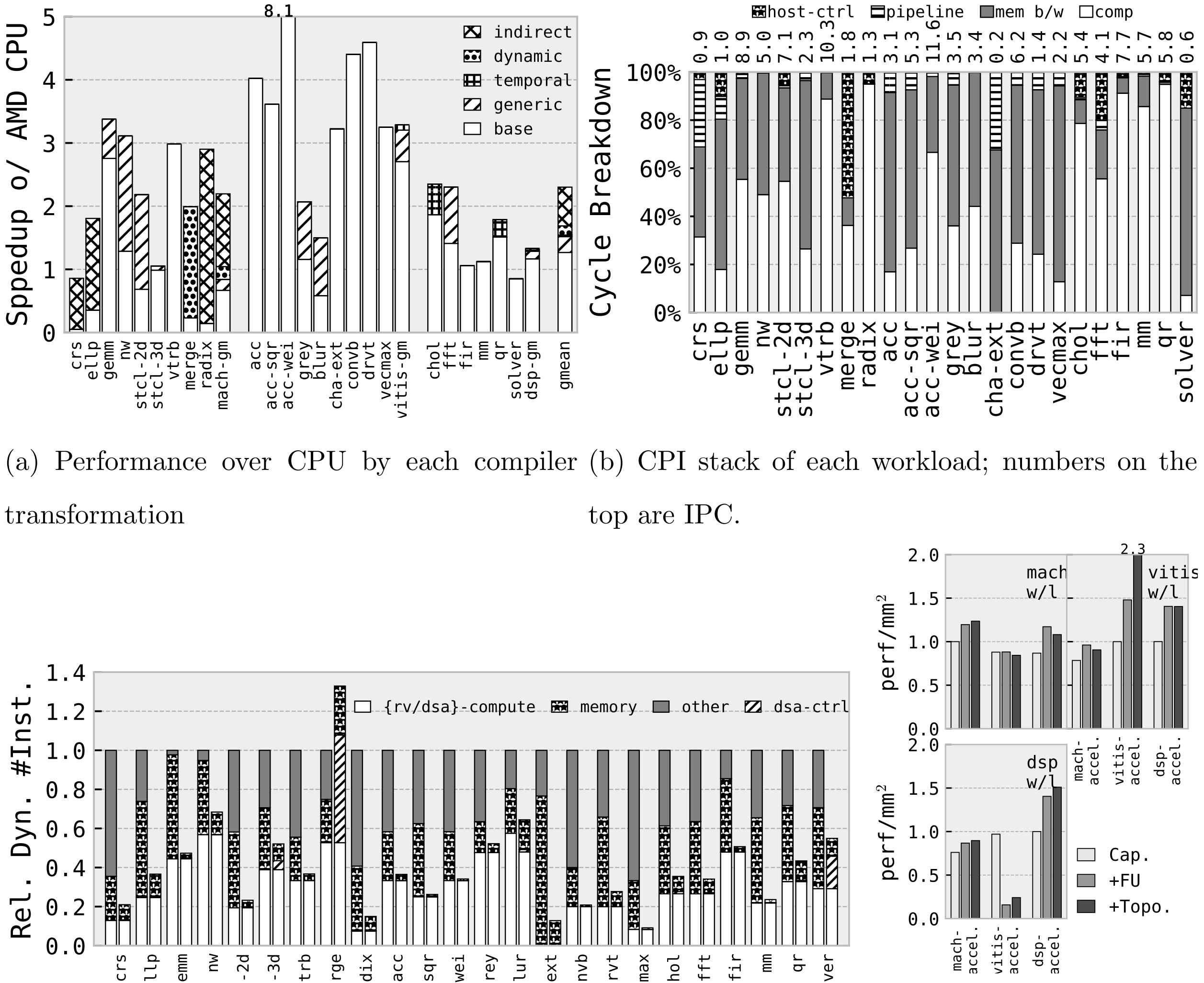
Simulation We develop a cycle-level simulator for performance estimation, by integrating a spatial architecture simulator to a gem5 single-issue core.

4.6 Evaluation

We evaluate the compiler's performance and robustness. The key takeaways are:

- Our compiler achieves 2.2×, 3.3×, and 1.3× speedup on the three workload suites, MachSuite, Vitis, and DSP, respectively.
- The generated binaries reduce dynamic RISCV instructions by mean 99.8% on the control core.
- Our compiler allows graceful performance degradation when compiling to accelerators with different degrees of specialization.

Idiomatic Transformation: Figure 4.10a shows the performance of each workload on the general initial accelerator when incrementally enabling optimizations. It achieves mean 2.3× speedup and 98.7× area-normalized speedup over the CPU.



(a) Performance over CPU by each compiler transformation

(b) CPI stack of each workload; numbers on the top are IPC.

(c) Dynamic instruction reduction enabled by decoupled-spatial compilation.

(d) Perf/mm^2 normalized to capability specialized architecture.

Figure 4.10: Performance and specialization study

Base is the code generation without any idiomatic optimization, which just transforms the program into decoupled dataflow and maps the decoupled aspects to specialized units, e.g. linear memory access to stream engine.

Generic refers to the stream coalescing and dimension fusion optimization. `gemm`, `nw`, `stcl-2d`, `stcl-3d`, `grey`, `blur`, and `fft` all have adjacent scalar access in the innermost loop body, thus benefiting from stream coalescing and dimension fusion. The performance of these 7 workloads is improved by mean 1.8× compared with *base* optimizations.

Temporal means offloading instructions to temporally shared processing elements. `cholesky` and `qr` both have code regions with more than one instruction outside the loop nest. Offloading these to shared PE's enables higher ILP and avoids control core serialization. Thus, their speedup is improved by mean 1.2×.

Dynamic supports the capability of conditionally popping data. Only `merge` benefits from dynamic capabilities; falling back to control instructions on the single issue control-core would cost 8.6× speedup.

Indirect enables parallel indirect memory access, and `radix`, `crs`, and `ellpack` all benefit by 11.5× speedup over feeding indirect access data one-by-one to the accelerator.

Speedup Implication: To study the source of speed up, we demonstrate the dynamic instruction reduction and the cycle breakdown. By compiling the original C codes to RISCV and simulating on gem5, we count the dynamic instructions. Figure 4.10c shows that 66% of the dynamic instructions are removed, and 99.8% of RISCV instructions are eliminated.

The instructions are in four categories: computation, memory, other, and dsa control. The "memory" and "other" category constitute the biggest savings, because of the stream encoded memory. For simplicity, we compare here against a non-vectorized RISCV baseline, but idiomatic memory streams can coalesce multiple memory operations into one request; "other" instructions mainly include pointer expression and loop control: these can be expressed by encoded streams to reduce instruction count by orders-of-magnitude.

Moreover, Figure 4.10b shows that besides computation, memory bandwidth is the second largest portion of the execution time, which indicates that speedups are mainly from high ILP enabled by the decoupled-spatial execution.

An outlier is merge sort, which not only has 1.3× dynamic instructions, but is also bounded by control instructions. Our current idioms can only capture the inner loop of merge sort. A loop nest with affine outer loop and data-dependent inner loop is not supported, and is a candidate for broadening the idioms.

Robustness over Specialization: We demonstrate the compiler's robustness on accelerators for three domains with different degrees of specialization.

Figure 4.10d shows the relative perf/mm^2 normalized by the capability-specialized accelerator. Our compiler can robustly target architectures with different feature combinations while exploiting the hardware/software affinity. Performance is retained on the designs specialized for the target domain. Topology-specialized have reduced area at the expense of performance on non-targeted domains.

4.7 Related Work

Idiomatic ISAs Prior idiomatic ISA constructs include streams [229] and streams+vectorization [81]. Prior idiomatic spatial architecture ISA's include database [261] and sparse processing primitives [74].

Accelerator Compilers Prior works developed general-purpose compilers for accelerators. DySER's compiler [105] developed a dataflow representation (AEPDG) which separates memory and computation. SARA [291] parallelizes sequential programs across spatial accelerator tiles. ParallelXL's compiler [52] targets general-purpose dynamic task scheduling. However, these compilers are for an accelerator with fixed capabilities.

Spatial Architecture Design FAST [289] incorporates loop reorganization into the DSE for ML-accelerators. CGRA-ME [62] and SNAFU [100] are CGRA generation frameworks that are flexible across topologies and resource allocation. REVAMP [33] and AURORA [247] have automated DSE for spatial architecture parameters (but not topology or capabilities).

High-Level Synthesis Vivado HLS also adopts a C+pragma programming interface, bit its pragmas are more complex. The key difference is that HLS generates a fixed hardware design for a single application.

Our modular compilation approach enables robustness across a significantly larger design space of programmable architectures, including different topologies and parameters.

CHAPTER 5

Hybrid Systolic-Dataflow Spatial Architecture for Inductive Workloads

In this chapter, we will introduce our work, REVEL, Reconfigurable Vector Lanes. Beyond targeting the inductive dense linear algebra kernels itself, this project works as a proof of concept for the idea of defining a unified software/hardware co-design space. All each hardware component of which all those software/hardware codesigned features are composed can be comprised within this space for further reuse. Moreover, further software/hardware codesigned innovations can also be done by extending existing components or adding new components to this space. This idea will be in detail discussed in Chapter 6.

Dense linear algebra kernels have long been the workhorse of signal processing for decades. The oncoming proliferation of the 5G standard just intensifies this condition[1]. The *inductive* nature of many algorithms in this domain, accompanied with small matrix sizes caused by the antenna array, makes the parallelism in these algorithms are hard to exploit, which will be characterized in Section 5.2.1.

We address these challenges by applying decoupled-spatial paradigm and integrating specialized mechanisms. In the rest of this chapter, we will first explain the kernels we target, and then discuss the challenges in these kernels. In Section 5.3.1, we discuss which set of the specialized mechanisms for the challenges are enabled. Finally, we evaluate it and conclude this project.

[1] When we first worked on this project in 2017, the 5G standard was still ongoing.

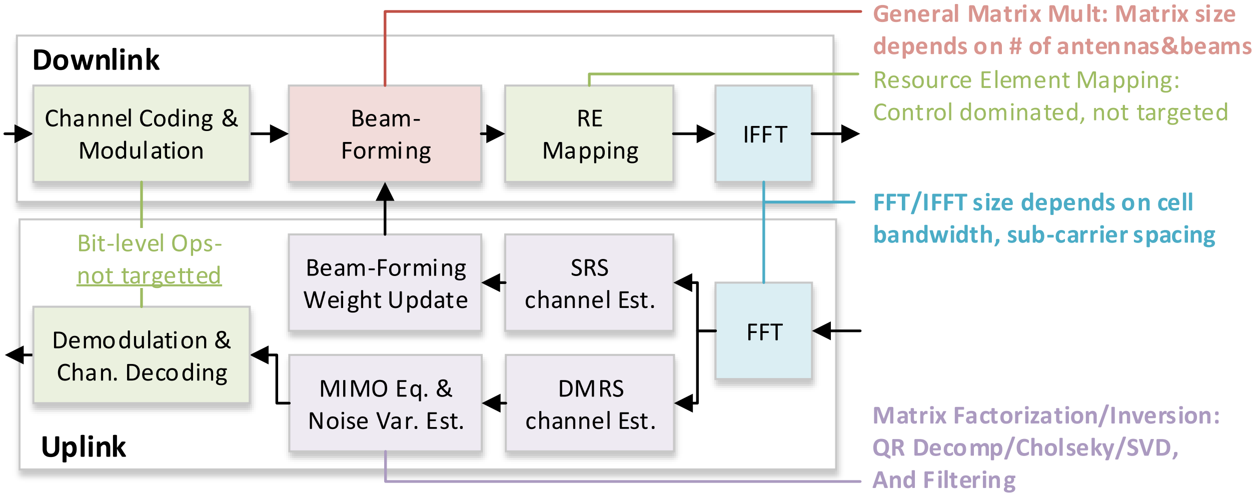

General Matrix Mult: Matrix size depends on # of antennas&beams
Resource Element Mapping: Control dominated, not targeted
Downlink
Channel Coding & Modulation
Beam-Forming
RE Mapping
IFFT
FFT/IFFT size depends on cell bandwidth, sub-carrier spacing
Bit-level Ops- not targetted
Beam-Forming Weight Update
SRS channel Est.
FFT
Demodulation & Chan. Decoding
MIMO Eq. & Noise Var. Est.
DMRS channel Est.
Uplink
Matrix Factorization/Inversion: QR Decomp/Cholseky/SVD, And Filtering

Figure 5.1: A typical 4G/5G Transmitter/Receiver Pipeline

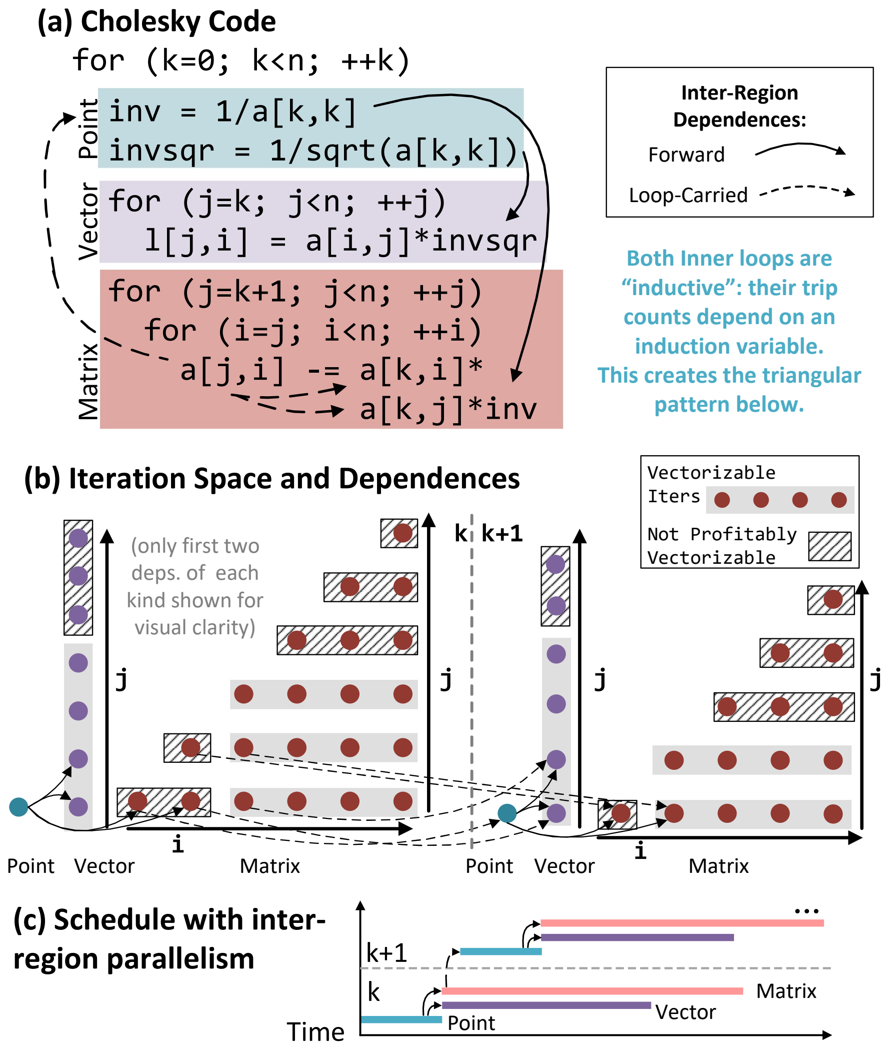
5.1 Wireless Signal Processing Pipeline

Figure 5.1 shows typical 4G/5G transmitter/receiver stages: Channel coding and modulation involve mostly bit-level arithmetic. RE mapping is a short resource allocation phase which is not computation intensive. The BF stage involves mostly matrix multiplication, coming from spatial signal filtering. Filters and FFT of several varieties are also very common [130, 295, 177].

The challenging workloads are mostly in MIMO equalization and channel estimation. These include Singular Value Decompose used by noise reduction, QR Decomposition used for signal detection, and Cholesky and Solver used in channel estimation. Because of the size of the antenna array in the base station, the sizes of these matrices are typically very small, ranging from 12 to 32.

5.2 Motivation

To design a specialized accelerator for wireless signal processing, we not only need to target conventional compute intensive workloads like FFT, filter, and matrix multiplication, but



(a) Cholesky Code

```
for (k=0; k<n; ++k)
  inv = 1/a[k,k]
  invsqr = 1/sqrt(a[k,k])
  for (j=k; j<n; ++j)
    l[j,i] = a[i,j]*invsqr
  for (j=k+1; j<n; ++j)
    for (i=j; i<n; ++i)
      a[j,i] -= a[k,i]*
                a[k,j]*inv
```

Both Inner loops are "inductive": their trip counts depend on an induction variable. This creates the triangular pattern below.

(only first two deps. of each kind shown for visual clarity)

Point
Vector
Matrix
Inter-Region Dependences:
Forward
Loop-Carried
(b) Iteration Space and Dependences
Vectorizable Iters ●
Not Profitably Vectorizable ▨
k
k+1
j
i
Point
Vector
Matrix
Point
Vector
Matrix
(c) Schedule with inter-region parallelism
k+1
k
...
Matrix
Vector
Point
Time

Figure 5.2: Inductive Workload Example: Cholesky

also need to target inductive workloads, like QR, Cholesky, and SVD. Our goal is to have a unified accelerator specialized for both kinds of workloads. Since the acceleration for FFT, filter, and multiplication has already been well studied, next we only discuss the challenges of inductive workload in Section 5.2.1, and explain why the state-of-the-art solution cannot perfectly resolve these challenges.

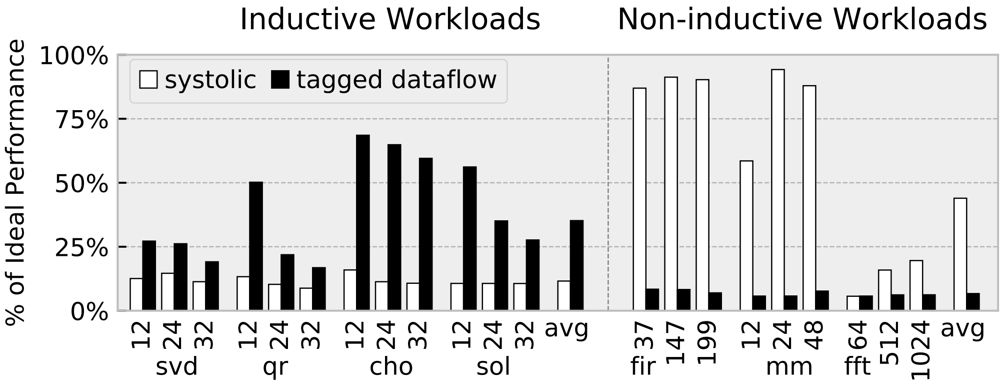


Figure 5.3: Spatial architecture performance normalized to equal-provisioned ideal ASIC's.

5.2.1 Characterizing Inductive Algorithms

Inductive Workloads To characterize the properties of inductive workloads, we take Choleksy as an example. The term, *inductive*, indicates that the algorithm has loop carried dependence and recursively applies a process to a sub-problem — as it is shown in Figure 5.2 (a), Cholesky iteratively applies three phases of computation on a shrinking matrix controlled by the outer loop `k`. This algorithm contains three interdependent phases are *point*, *vector*, and *matrix* computations. This workload is unique to FFT, filter, and matrix multiplication, because:

- **Imperfect Loop Body** Figure 5.2 (a) shows that the loop body of this algorithm is no longer perfect. The outermost loop `k` can be separated into three interdependent regions. The result produced by the *scalar* region is consumed by both the *vector* and *matrix* region.
- **Shrinking Matrix Size** Figure 5.2 (b) shows that because the loop trip count of the innermost loop `j` keeps changing, it is impossible to have a perfect loop tiling factor.

5.2.2 Challenges

General-Purposed Specialization The conventional general-purpose *multi-threading* and *vectorization* cannot specialize for these workloads well. As it is shown in Figure 5.2(b), if we view the dependence in elementwise granularity, each phase can be executed as early as the data is available. However, to achieve the execution schedule shown in Figure 5.2(c), the overhead of thread synchronization and context switch for just a few elements is higher than the benefit gained from the parallelism. This is especially true when the matrix sizes are small, and shrinking. Moreover, because of the shrinking matrix size, it is impossible to find a perfect loop tiling factor to transform the code into SIMD instructions without residue instructions as it is shown in Figure 5.2(b).

Decoupled-Spatial Paradigm Decoupled-spatial accelerators seem promising since the instruction execution is driven by the data availability, which avoids the overhead of synchronization in *multi-threading*. To adopt decoupled spatial architecture, a very first question to answer is which execution model to adopt. As we discussed before, there are two key dimensions of the execution model, the size of the instruction buffer and the timing of instruction execution. Under a fixed power/area budget, we stress the decoupled-spatial architectures with both systolic and tagged dataflow execution model, and the results are shown in Figure 5.3. Each type of spatial architecture can only perform well on a subset of the workloads.

Just like SIMD, statically scheduled spatial PE's (Systolic and CGRA) can only parallelize the innermost loop [140] like SIMD, so the inter-dependent code regions are executed in serial. Host controller intervention is inevitably required to coordinate the non-uniform produce/consume rate between the code regions. The dynamically scheduled PE's in ordered dataflow can handle this, but each of them is dedicated to only one instruction. When it comes to too many instructions in the concurrent code regions (e.g. QR and SVD), the dedicated instruction buffer can easily be overwhelmed. Then the execution falls back to serializing the execution of each region and reconfiguring the accelerator. Tagged dataflow

PE's can buffer multiple instructions in a single PE to achieve the code region concurrency. However, it suffers from not only high area/power overhead, but also difficulty of scheduling the instructions. As it is shown in Figure 5.3, in compute-intensive workloads because of the resource contention across instructions, the performance of the tagged dataflow is poor. We claim that none of the execution model of decoupled-spatial architecture can well specialize this application domain alone, so it urges us to develop new specialized architecture mechanisms.

5.3 REVEL: Reconfigurable Vector Lanes

In this section, we introduce the developed hardware specializations to resolve the challenges discussed in the section above, and then we scale it up to a multicore system for both latency and throughput.

5.3.1 Composing Hardware Primitives

Hybridizing Systolic and Dataflow A key observation is that, in inductive workloads, program regions have mismatched instruction count and execution frequency — program regions in outer loops often have more instructions, and the body of innermost loops are highly data-parallel multiplications and additions. These data-parallel compute intensive regions can be well specialized by systolic PE's. On the other hand, code regions with less execution frequency often have more numbers of instructions. It will be highly desirable to keep these instructions on the spatial PE with larger instruction buffer with fewer computation resources provided. *Tagged dataflow* is attractive to this idiom, because it can fit in multiple instructions and resilient to the timing variation caused by inductive execution. Therefore, we decide to integrate a few tagged dataflow in the systolic PE mesh.

The dataflow processing elements are embedded into the on-chip network. The communication between the systolic and shared PE's should be carefully enforced by the compiler

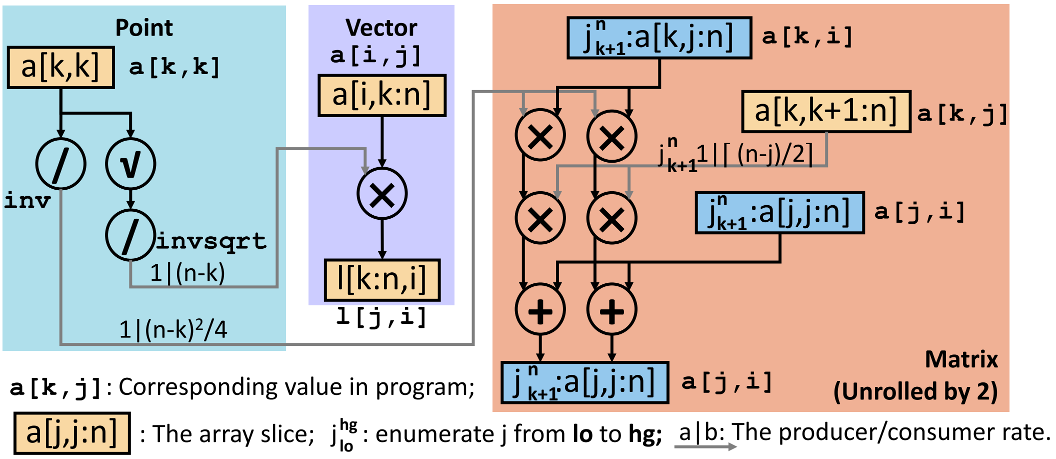
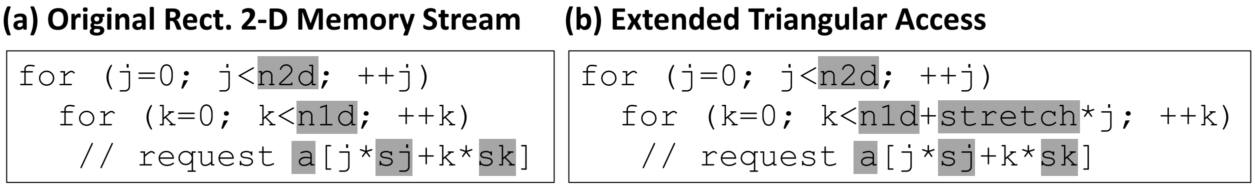


Figure 5.4: The dataflow of Cholesky

— synchronization FIFO intervention are required to reason a timing of data arrival and execution when there are multiple instructions mapped to a single shared PE or shared PEs want to pass data to systolic PEs. In addition, the shared dataflow PE can still be used as a dedicated PE with a very low priority. Only in this case, the shared and systolic PE's can communicate directly through the on-chip network.

Producer-Consumer Because of The different execution frequency also introduces non-uniform producer/consumer rate among different code regions. If the value is consumed in a higher rate than it is produced, this value should be reused multiple times. This kind of communication across codes regions should be coordinated by the synchronization elements (ports) to comply the timing of systolic PE's: a value are fired to the spatial architecture, but this value will not be popped from the FIFO until the time of reuse if over. For example, as it is shown in Figure 5.4, the `invsqrt` and `inv` produced by the "point" are reused (n-k), and $\frac{(\text{n-k})^2}{4}$ times respectively in the "vector" and "matrix" region. The port repeat feature in Figure 6.4 should be integrated. Moreover, because the memory operation `a[k,j]` is under loop `i`, and the trip count of loop `i` depends on loop `j`, the time of reuse should decrease when loop `j` moves forward. More specifically, because of the loop unrolling, we have a



(a) Original Rect. 2-D Memory Stream

```
for (j=0; j<n2d; ++j)
  for (k=0; k<n1d; ++k)
   // request a[j*sj+k*sk]
```

(b) Extended Triangular Access

```
for (j=0; j<n2d; ++j)
  for (k=0; k<n1d+stretch*j; ++k)
   // request a[j*sj+k*sk]
```

Figure 5.5: The extended triangular memory stream support

ceiling in the formula of time of reuse — the time of reuse should decrease by 1 when every 2 elements are popped from the FIFO. This feature was first proposed in this work and has already been clarified in Section 3.3.

Triangular Streams The blue boxes in Figure 5.4 show that the inner loop iteration is controlled by the outer loop. To achieve the same access pattern with our dense memory access pattern support, we have to issue excessive commands. Since we only have a single-issue control host, the performance can easily be bounded by issuing massive control commands. Therefore, as it is shown in Figure 5.5, we extend the original 2d memory access from (a) to (b). By passing an additional parameter `stretch`, the inner loop iteration can change when the outer loop moves forward. This practices the idea of making software/hardware co-innovations within a unified design space discussed in the introduction of this chapter.

Implicit Padding We already have triangular stream support, but the performance will still be bounded by massive control commands when unrolling is adopted. As it is shown in Figure 5.2, because the trip count of inner loops depends on the outer loops, it is impossible to find a perfect tiling factor to benefit from the data parallelism achieved by unrolling. As it was discussed in the background section, implicit padding can be done by taking advantage of the additional meta information of the stream execution state. When every time the state machine that tracks the execution of a memory stream comes to the end of the innermost loop, predication-off that matches the residue of the tiling factor should be implicitly fed to the coordination FIFO. The triangular stream and implicit padding together enables low

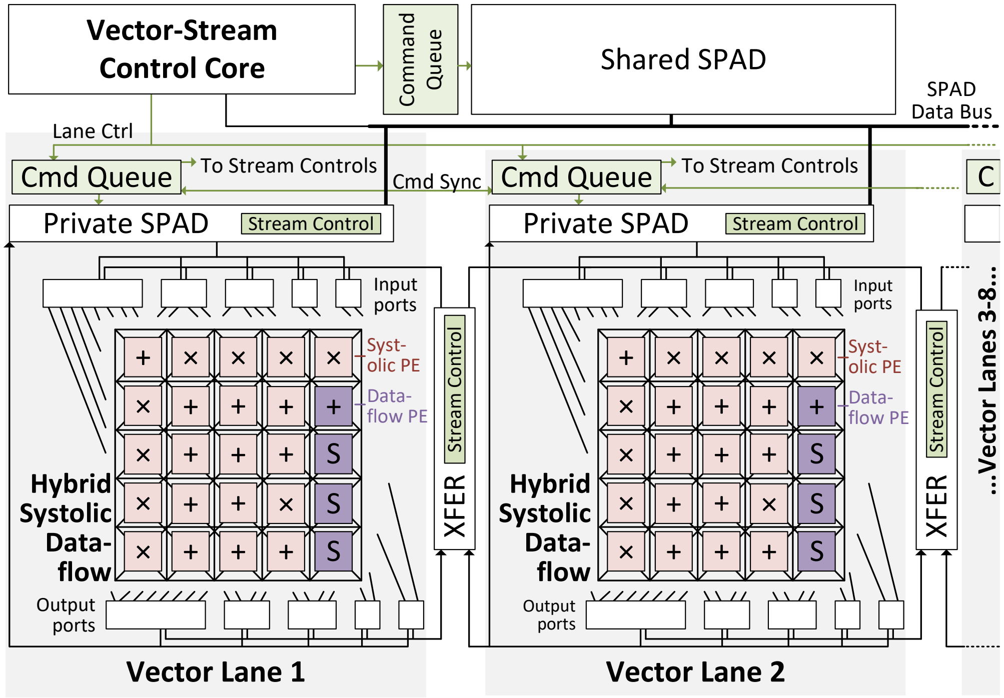


Figure 5.6: The architecture of REVEL.

control overhead to execute the complicated data access.

5.3.2 Scaling Up

Rationale We have to scale up our design for the purpose of both latency and throughput. We can either have a large spatial architecture or have multiple lanes of relatively small spatial architectures. To sustain the computation on a large spatial architecture, higher memory bandwidth and more coordination resources should also be added. Moreover, a large spatial architecture may lead to an exponentially larger space to explore the mapping. The excessive mapping time makes it impractical. Therefore, we decide to use multiple duplications of small spatial architectures. The final architecture of REVEL shows in Figure 5.6.

ISA Extension We extend new hardware intrinsics over the ISA discussed in Section 3.3 to support multi-lane scaling up. As it is shown in Table 5.1, we have

Instruction	Parameters
`Context`	`mask`: The bitmask of the accelerator lanes.
`ConfigOffset`	`offset`: An offset applied on the starting address.

Table 5.1: The extended ISA

- **Context:** This instruction configures a bitmask predication that indicates the streams are broadcasted to the REVEL lanes.
- **ConfigOffset:** This instruction configures the offset of the starting address for those accelerators with bitmask predication enabled. This instruction enables task partition with low control overhead.

5.4 Evaluation Methodology

REVEL Modeling Table 5.2 shows REVEL hardware parameters. All blocks are modeled at a cycle level in a custom simulator, which is integrated with a gem5 model of a RISCV inorder core [39, 221], extended for vector-stream control. To compare against state-of-the-art spatial architectures, we create a custom simulator for each. We synthesized REVEL's prototype using Synopsys DC, 28nm tech library. The design meets timing at 1.25GHz. An open source triggered instructions implementation was our reference for the temporal fabric [218]. Results from synthesis are used to create an event-based power model and area model.

ASIC Analytical Models These optimistic models (Table 5.3) are based on the optimized algorithms, and are only limited by the algorithmic critical path and throughput constraints, with equivalent FUs to REVEL. ASIC area and power models only count FUs and scratchpad.

Revel Lane (× 8)	Spatial Fabric	PEs	14 add, and 3 sqrt/div, 9 mult
		Div/Sqrter	Lat.: 12 Cyc., Thr.: 1/5 Cyc.
		SubwrdSIMD	4-way Fixed-point, 2-way FP
		Dataflow PE	1x1 (32 Instruction Slots)
	Vector Ports	Width	2×512, 2×256, 1× 128, 1× 64 bit
		Depth	4-entry FIFO
	Stream Ctrl	Stream Table	8 Entries for 8 concurrent streams
		Cmd Queue	8-Entry Cmd Queue
	SPAD	Structure	8Kb, Single-bank
		Bandwidth	512 Bits (1R/1W Port)
	Net.	Data Bus	2×512 Bit (SPAD+XFER)
		Spatial Mesh	64-bit Circuit-switched Mesh
Ctrl	Core	RISCV ISA [27], 5-stage, single-issue, 16kb d$, insts. added for stream-commands	
Shr.	SPD	Structure: 128Kb, Single-bank Bandwidth: 512 Bits (1R/1W Port)	
Net.		Inter-lane: 512 Bit Data Bus (8-bit Cmd Sync) Shared scratchpad Bus: 512 Bit Shared Bus	

Table 5.2: REVEL Parameters

SVD	QR		MM
$4dm + 2\mathrm{QR}(n) + \lceil \frac{n^3}{8_{\text{vec}}} \rceil$	$7dn + 2\sum_{i=1}^{n}(i + \lceil \frac{i}{2_{\text{vec}}} \rceil n)$		$\lceil \frac{n}{8_{\text{vec}}} \rceil mp$
Solver	FFT	Cholesky	Centro-FIR
$2\sum_{0}^{n-1} \max(\lceil \frac{i}{4_{\text{vec}}} \rceil, d+2)$	$\frac{n}{8_{\text{vec}}} \log n$	$\sum_{i=1}^{n-1} \max(\lceil \frac{i^2}{2_{\text{vec}}} \rceil, 4\mathrm{d})$	$\lceil \frac{n-m+1}{4_{\text{vec}}} \rceil m$

Table 5.3: Ideal ASIC Models. m, n, p are the matrix dims. (except SVD, where m is the number of iterations and Centro-FIR, where m is the filter size), x_{vec} indicates x-vectorized, and d is the latency of div/sqrt.

Workload	Data Size	Lanes
SVD	**12**,16,24,**32**	1
QR	**12**,16,24,**32**	8
Cholesky	**12**,16,24,**32**	8
Solver	**12**,16,24,**32**	1
FFT	**64**,128,512,**1024**	1
GEMM	**12**,**48**x16x64	8
FIR	**37**,**199**x1024	8

Table 5.4: Workload Parameters. "small","large" sizes bolded

Workload Versions We evaluate batch size 1 and 8, requiring different optimizations: For batch 1, REVEL spreads work across lanes (if possible), and for batch 8 each lane operates over one input. Table 5.4 shows data-sizes and # lanes in batch 1.

Comparison Methodology For fairness, we compare designs with similar ideal max. FLOPs (except GPU, which has more):

- **TI 6678 DSP (@1.25GHz)** 8-core DSP, each core has 16-FP adders/multipliers, using DSPLIB_C66x_3.4.0.0.

- **OOO Core: Intel Xeon 4116 (@2.1GHz)** Conventional OOO processor using highly optimized Intel MKL library. (8 cores used)
- **GPU: NVIDIA TITAN V (@1.2GHz)** GV100 graphics processor using cuSOLVER, cuFFT, and cuBLAS NVIDIA CUDA library as our gpu benchmark. GPU's peak FLOPs is >10× *higher* than REVEL.
- **Spatial:** The *systolic* design is similar to Softbrain [193], and *dataflow* is similar to Triggered Insts. [200]. FUs and #lanes are the same.

5.5 Evaluation

Our evaluation has four main goals. First to quantify the speedups over state-of-the-art CPUs, DSPs, Spatial, and GPUs. Second, to characterize the sources of benefits behind the specialization of inductive parallelism, as well as the remaining bottlenecks. Third, to understand the sensitivity to architecture features. Finally, to compare the area/power/performance with ASICs. Overall, we find that REVEL is consistently better than all state-of-the-art designs, often by an order of magnitude.

5.5.1 Performance

Overall Speedup Speedups over DSP for batch 1 are shown in Figure 5.7. The DSP and CPU have similar mean performance. REVEL attains up to 37× speedup, with geomean of 11× and 17× for small and large data sizes. REVEL is 3.5× and 3.3× faster than dataflow and systolic.

Performance for batch 8 is in Figure 5.8. For small and large sizes, REVEL gets a speedup of 6.2× and 8.1× over the DSP and CPU. REVEL's dataflow/vector-stream model provides 4.0× speedup over dataflow, and 2.9× over systolic.

REVEL provides factors speedup over state-of-the-art.

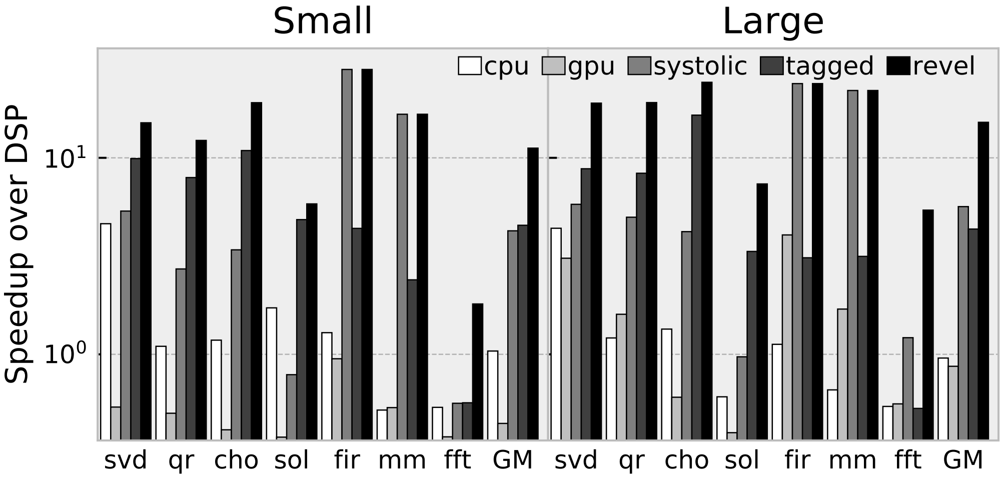


Figure 5.7: Performance Tradeoffs (batch size=1)

REVEL vs CPU Parallelism Figure 5.9 shows the scaling of REVEL's performance against the MKL's library's CPU version for different sizes of Cholesky and thread counts. Observe that when multi-threading is first enabled in MKL (>= matrix size 128), it actually hurts performance. This is because of the inherent fine-grain dependences, which REVEL supports natively.

Inductive dataflow can parallelize much finer-grain dependences than with CPU threading.

Benefits from Hardware/Software Mechanisms To understand the sources of improvement, we evaluate four versions of REVEL with increasingly advanced features. We start with the systolic, then add inductive streams, hybrid systolic-dataflow, and finally stream predication to enable efficient vectorization. Figure 5.10 shows the results.

Inductive memory and dependence streams improve all workloads by reducing control and increasing parallelism. Even FFT benefits by using inductive reuse to reduce scratchpad bandwidth. QR and SVD have complex outer-loop regions, so do not benefit as much until after adding hybrid systolic-dataflow, which enables more resource allocation for inner-loop regions.

Solver was also accelerated by the heterogeneous fabric because it is latency sensitive,

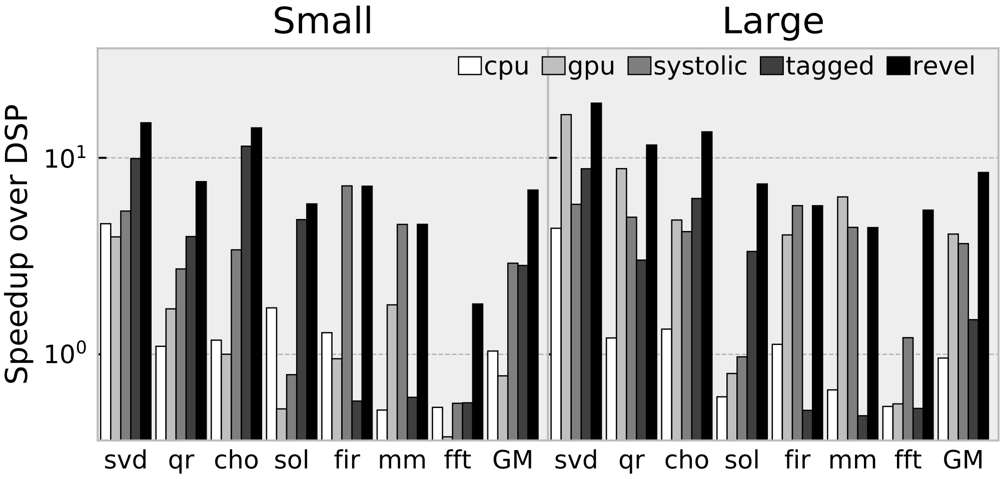


Figure 5.8: Performance Tradeoffs (batch size=8)

and collapsing less critical instructions can reduce latency. The vectorized workloads also receive large gains from stream predication by reducing the overheads of vectorization.

The vector-stream ISA and hybrid systolic-dataflow architecture together enable high performance.

Cycle-Level Bottlenecks Figure 5.11 overviews REVEL's cycle-level behavior, normalized to systolic. To explain the categories, *issue* and *multi-issue* means that one or multiple systolic regions fired, and *temporal* means only a temporal dataflow fired during that cycle. All other categories represent overhead, including the *drain* of the dedicated fabric, *scr-b/w* and *scr-barrier* for bandwidth and synchronization, *stream-dpd* for waiting on dependences, and *ctrl-ovhd* for waiting on the control core.

The clearest trend is that our design reduces the control overhead dramatically. For some kernels, REVEL is able to execute multiple regions in the same cycle, especially for larger matrices. One outlier is FFT with small data; it requires multiple reconfigurations, each requiring the pipeline to drain.

Exploiting inductive parallelism increases parallel work and reduces control, enabling better performance.

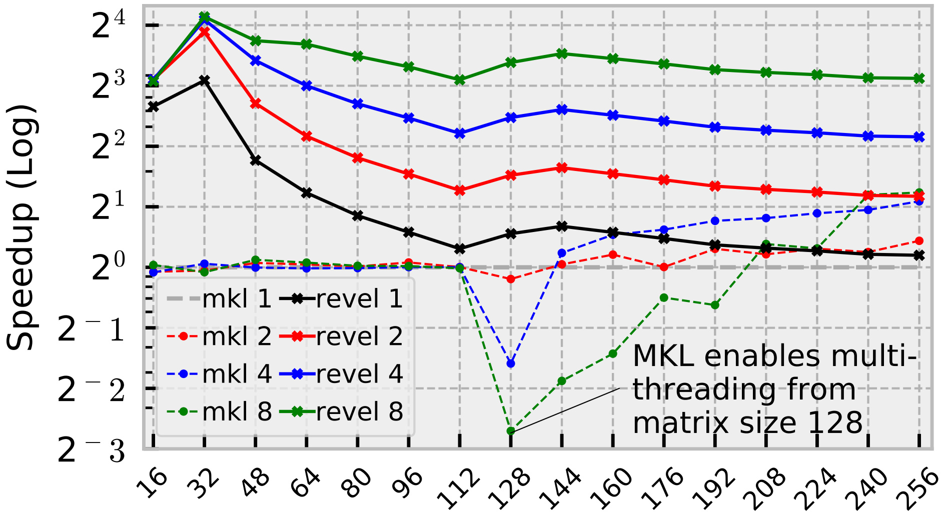


Figure 5.9: CPU vs REVEL Scaling

Dataflow PE Allocation Tagged dataflow PEs are helpful on inductive workloads, but expensive. A tagged-dataflow PE costs $>5\times$ more area than a systolic PE ($2822\mu m^2$ versus $16581\mu m^2$). Figure 5.12 shows REVEL's performance and area sensitivity. SVD has the largest demand on dataflow PEs, so are affected the most. The effects on other workloads are neglectable, so we choose 1 dataflow PE to minimize the area penalty.

5.5.2 Area and Power Comparison

Breakdown Table 5.5 shows the power/area breakdown; the largest source (especially power) comes from FP units. REVEL is $1.93mm^2$, and 1.63 Watts.

Comparing against CPU and DSP Figure 5.13 shows the relative performance/area normalized to the CPU after adjusting the technology. The DSP achieves a high performance/mm^2, and REVEL is able to achieve even higher performance with a moderate area overhead. REVEL has 1089× performance/mm^2 advantage over the OoO core, and 7.3× over the DSP.

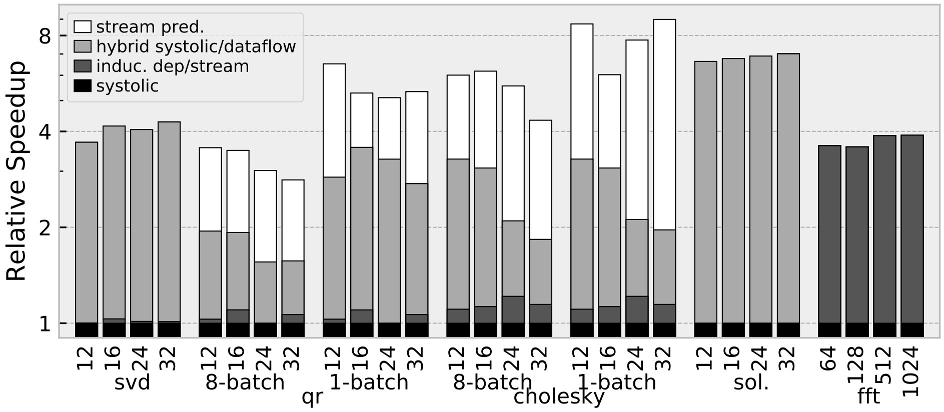


Figure 5.10: Performance Impact of Each Mechanism.

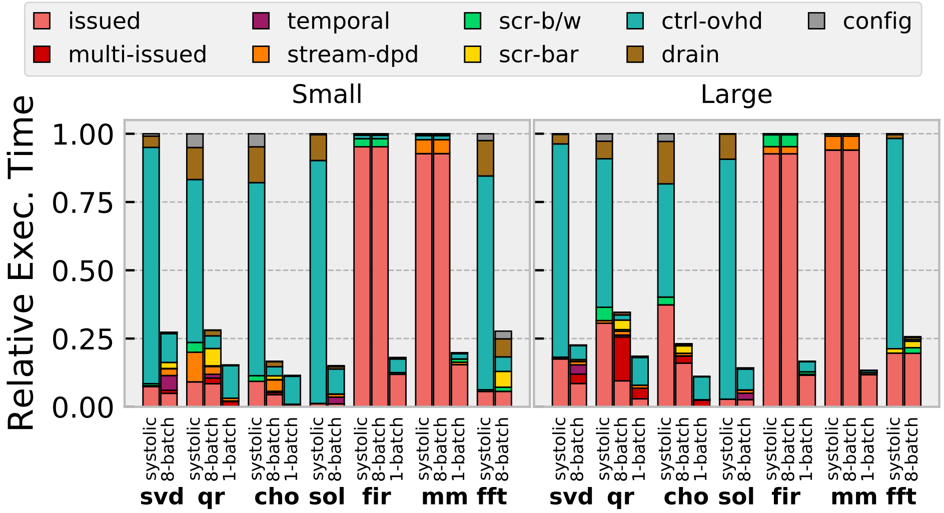


Figure 5.11: REVEL's Cycle-level bottlenecks

Comparing against ASIC Table 5.6 shows performance-normalized area overhead over ASIC analytical models. REVEL is mean 2.0× power. This is mostly due to the control logic (ports, bus, etc.) and reconfigurable networks. It is 0.55× the area of the combined ASIC. This is optimistic for ASICs in that it assumes perfect pipelining and no control power.

REVEL is on par with ASICs-level efficiency.

		area(mm^2)	power(mw)
Compute	Dedi. Net. (24)	0.06	71.40
Fabric	Temp. Net. (1)	0.02	14.81
	Func. Units	0.07	74.04
	Total Fabric	0.13	160.25
Control (ports/XFER/str. ctrl)		0.03	62.92
SPAD-8KB		0.06	4.64
1 Vector Lane		0.22	207.90
Control Core		0.04	19.91
REVEL		1.93	1663.3

Table 5.5: Area and Power Breakdown (28nm)

Workloads	SVD	QR	Cho.	Sol.	FIR	MM	FFT	Mean
Power Ovhd.	2.8	2.0	1.9	1.6	2.0	1.9	1.9	2.0
Area Ovhd.	3.3	2.4	2.3	2.2	2.3	2.3	2.8	2.5/0.55

Table 5.6: Power/Area overheads to ideal ASIC (iso-perf)

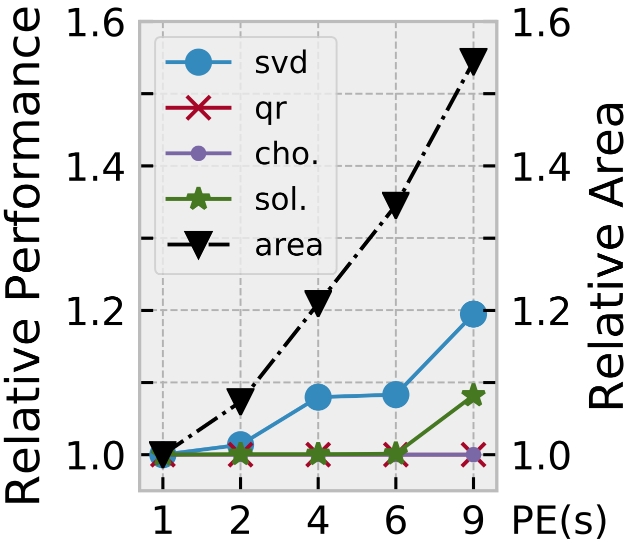


Figure 5.12: Temporal region sensitivity

5.6 Related Work

In this section, we discuss work other than that previously covered by the spatial architecture taxonomy in Section 3.2.1.

Synchronous Dataflow Variants The inductive production to consumption rates in our dataflow model is inspired by the static rates in synchronous dataflow [159] (SDF). SDF was developed as a specialization of existing dataflow models which could be statically scheduled. Cycle-static dataflow [38] extends SDF with periodically changing rates, and heterochronous dataflow [99] extends SDF to enable an FSM to step through predefined rates. None of the above were applied to spatial architectures or handle inductive dependences.

StreamIt [250] is a language and runtime with somewhat similar semantics to vanilla SDF, and was evaluated on RAW [103], a (mostly) static/shared-PE spatial architecture.

Outer-loop Parallelism Prabhakar *et al.* develops "nested-parallelism," which enables coupling of datapaths with nested parallel patterns [209]. Inductive parallelism is a general-

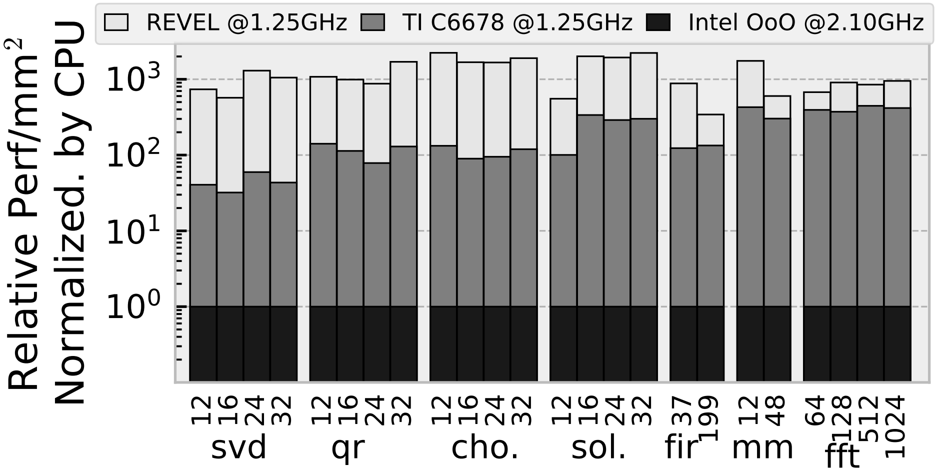


Figure 5.13: Relative performance/mm^2 normalized to CPU.

ization of nested-parallelism, and we can achieve a higher utilization due to hybrid systolic-dataflow execution.

Some CGRA compilers target nested loops [140, 160], but only parallelize the epilogue and prologue of subsequent loop nests. Recent work has made progress in pipelining imperfect nests [286], but does not parallelize across multiple region instances. CGRA Express [203] allows a CGRA to use the first row of its PEs in VLIW mode during outer loops. Concurrent execution across inner and outer regions is not attained. None of the above handle inductive dependences.

In addition, recent work [58] by Cheng *et al.* target inter-loop parallelism on a manycore system by adopting a dynamic task-parallel programming interface and communicating data through software managed scratchpads.

Flexible Vectorization Vector-threading techniques also marshal independent execution lanes for vectorized execution when useful [161, 151, 219, 139]. The RISC-V vector extension supports configurable vector-length and implicit vector masking [93]. Vector-length is limited by physical registers (REVEL's streams are arbitrary length), and inductive access is not supported, so the vector length would have to be reset on each iteration. These architectures are also not spatial, so cannot exploit pipelined instruction parallelism.

Some spatial dataflow models use predicates for control [235, 43]. These do not use streams for vector predication. dMT-CGRA [264] adds inter-thread communication for a spatial-dataflow GPU [263, 262].

DSP Accelerators Many application/domain-specific reconfigurable designs have targeted DSP algorithms. Fasthuber et. al [88] outline the basic approaches. One representative example includes LAC [205], targeted at matrix factorization. Our architecture allows more general programmability.

Stream-based ISAs and Reuse Many prior architectures have used memory-access stream primitives [220, 278, 193, 64, 272, 121, 69, 271]. To our knowledge, no prior work has incorporated inductive patterns into such streams.

5.7 Conclusion

To conclude, this work, besides being a promising accelerator that target the DSP domain, fortifies the idea of having a unified reconfigurable accelerator over a collection of application-specific designs. With moderate overhead, near-ASIC performance can be achieved while retaining flexibility. In addition, the idea of unifying the design space of specialized accelerator is well practiced. Hardware features can be integrated from a unified design space as needed by the target applications, and software/hardware co-innovations can be done by adding new components or extending existing components. This idea enables my Ph.D. core project, DSAGEN [273], which will be in detail discussed in the next Chapter.

CHAPTER 6

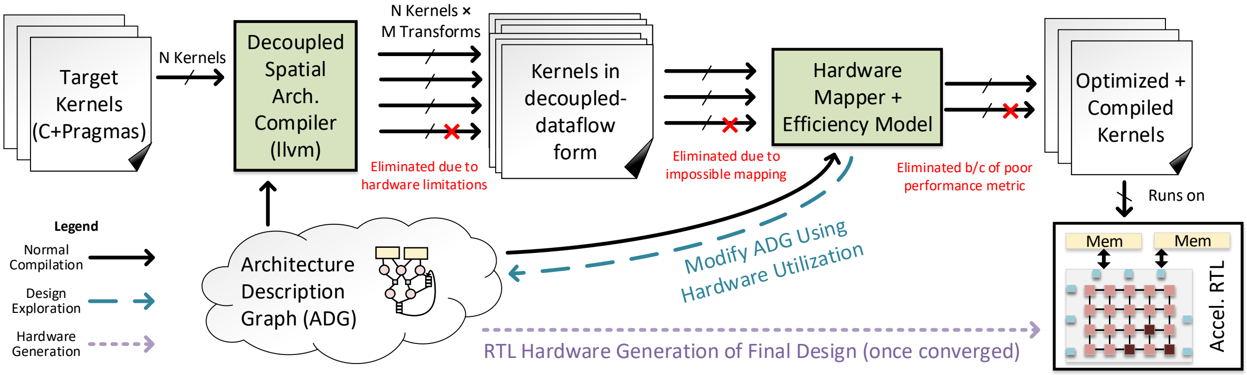
DSAGEN: Synthesizing Programmable Accelerators

In this chapter, the core project of this dissertation, DSAGEN [273], will be elaborated. This work aims at building a programmable accelerator design automation framework. The overview of this work is shown in Figure 6.1. There are three key aspects of this framework, the **design space**, the **software stack**, and the **design automation algorithm**.

The **design space** is defined by formalizing the approach discussed in Chapter 5: each component of which the software/hardware co-designed features are composed is comprised in this design space, and each design point can be represented by connecting these components. A graph is a data structure that naturally encodes the connectivity, and such a graph is called an architecture description graph (ADG).

Accelerators are designed by studying the program behaviors of interest to derive specialized hardware mechanisms, which means all the design demands are naturally encoded in the programs. Therefore, we adopt the programming interfaces and compilation flow described in Section 4.1 as our **software stack** so that a wide range of specialized accelerators with optional features can be robustly targeted. By taking advantage of the compiler's capability of exploring the set of transformations and generating multiple versions of transformed IRs, the design demands and the software/hardware affinity can easily be attained by the compiler.

Base on the design space, and the software stack, we develop a **design space exploration (DSE) algorithm** that iteratively make changes to the candidate architecture and select the IR version that best fit this underlying hardware, guided by the software/hardware



Target Kernels (C+Pragmas)
N Kernels
Decoupled Spatial Arch. Compiler (llvm)
N Kernels × M Transforms
Eliminated due to hardware limitations
Kernels in decoupled-dataflow form
Eliminated due to impossible mapping
Hardware Mapper + Efficiency Model
Eliminated b/c of poor performance metric
Optimized + Compiled Kernels
Runs on
Mem
Mem
Accel. RTL
Architecture Description Graph (ADG)
Modify ADG Using Hardware Utilization
RTL Hardware Generation of Final Design (once converged)
Legend
Normal Compilation
Design Exploration
Hardware Generation

Figure 6.1: Programmable Accelerator Design Automation Framework — DSAGEN

affinity estimated by our analytical models. Once the software/hardware pair converge, a specialized accelerator and applications well-tuned for this accelerator are generated.

In the rest of this chapter, the technical details of three key aspects, the hardware description, the transformation exploration, and the design space exploration, that together enable automated accelerator design will first be covered. Then, we will evaluate and conclude this work.

6.1 Architecture Description Graph

To automatically synthesize programmable accelerator, a very first question to answer is how the design space is designated and represented. As mentioned in Chapter 1, the decoupled-spatial architectures are attractive to us because of their performance, flexibility, and the capability of approximating prior works. Moreover, according to our prior studies, optional hardware/software codesign features can be independently integrated to these architectures by composing different components to explore tradeoffs between hardware cost and degree of specialization. These primitive architecture components and integration are discussed in Section 3.2.1.

A graph naturally encodes the connectivity among these components to a design point of a

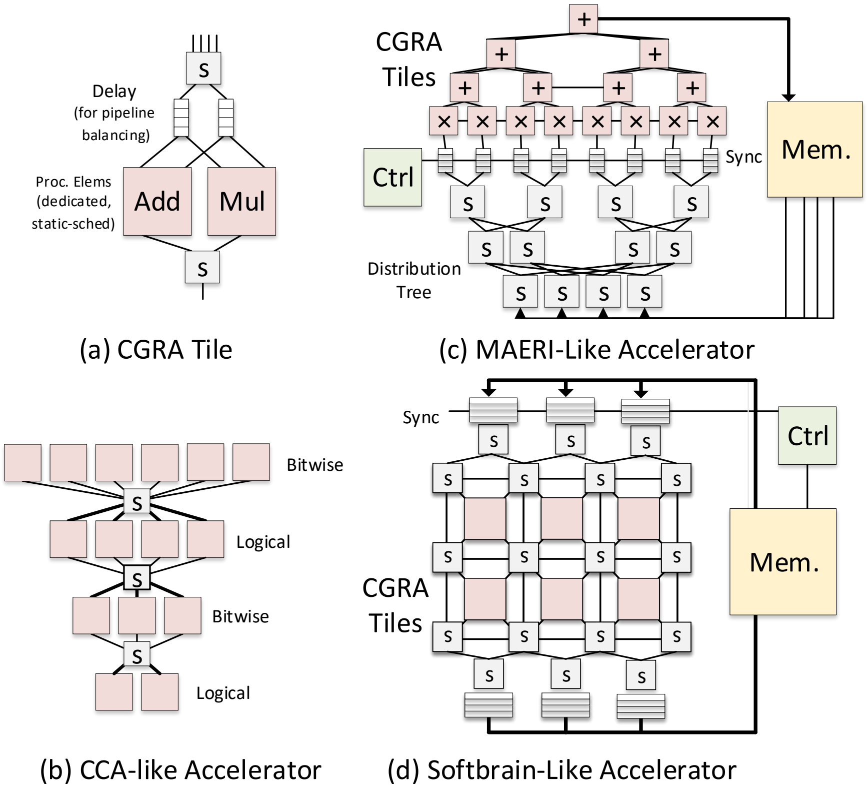
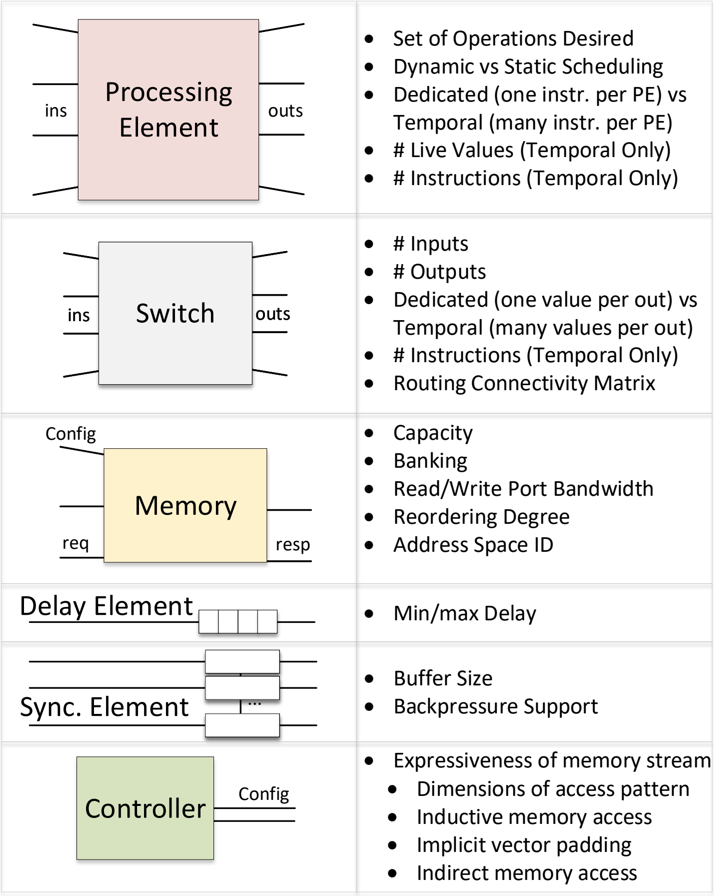


Figure 6.2: The examples of ADG approximation.

decoupled-spatial architecture. Such a graph is called architecture description graph (ADG). Figure 6.2 shows how ADG's approximate the prior reconfigurable accelerator works. Instead of tuning the parameters of a template hardware, representing the architectures in a graph gives us more flexibility on exploring the topology among the components, which enables potentially deeper specialization.

6.1.1 Parameterizing Components

Besides the connectivity, the attributes of each vertex in the ADG should also be specified. In this section, we discuss trade-offs among flexibility, programmability, and hardware cost enabled by these parameters. The components and the parameters of each component are shown in Figure 6.3. Instead of viewing the ISA we discussed in Table 3.1 as a whole, the



Processing Element	• Set of Operations Desired • Dynamic vs Static Scheduling • Dedicated (one instr. per PE) vs Temporal (many instr. per PE) • # Live Values (Temporal Only) • # Instructions (Temporal Only)
Switch	• # Inputs • # Outputs • Dedicated (one value per out) vs Temporal (many values per out) • # Instructions (Temporal Only) • Routing Connectivity Matrix
Memory	• Capacity • Banking • Read/Write Port Bandwidth • Reordering Degree • Address Space ID
Delay Element	• Min/max Delay
Sync. Element	• Buffer Size • Backpressure Support
Controller	• Expressiveness of memory stream • Dimensions of access pattern • Inductive memory access • Implicit vector padding • Indirect memory access

Figure 6.3: The hardware primitives that compose a decoupled-spatial architecture.

ISA can be modularly enabled/disabled according to the parameters of the components. As it is shown in Figure 6.4, the deeper colors of boxes indicate deeper degree of specialization (which also requires more on-chip resources), and the nested boxes indicate the dependences among the specialization.

Processing Elements (PE) can be categorized in two dimensions, the timing of execution (dynamic/static), and the size of the instruction buffer (dedicated/shared). These two dimensions together enable tradeoffs among hardware cost, compilation time efforts, and degree of specialization as discussed in prior chapters (Chapter 2 & 3).

On-chip Network is composed by switches. The key parameter of a switch is its connectivity and the capability of traffic control. Each switch has a #in×#out matrix to describe the connectivity between inputs and outputs. In a systolic PE array, the switch has no capability of traffic control, and in a dynamically scheduled array, the switches should have capability to backpressure the upstream nodes.

Decomposability The granularity of routing and computation can both be down to bytes. Each switch and arithmetic unit in PE can optionally be *decomposable* [74] down to a certain bitwidth, aligned with a power of two. This enables better flexibility of data types, but introduces more on-chip resources overhead, which was discussed in Section 2.2.

Synchronization Ports The ports not only serve as the interfaces between the memory and the spatial architecture, but also can be used to control the consumption rate of the data, and the implicit padding predications of the loop residues (refer the `ConfigPort` and `pad` in Section 3.3).

Delay FIFO's These are critical for the systolic PE execution. If the timing cannot be matched perfectly, performance degradation will be imposed to guarantee the correctness of execution [192]. The throughput of firing computation cannot exceed:

$$\text{Pipeline Utilization} \leq \frac{\text{FIFO Size}}{\text{max miss} + \text{FIFO Size}}$$

Larger delay buffers can make compilation easier, but more areas are occupied.

Memory Memories are parameterized by their capacity, number of banks, number of concurrent streams as well as their expressiveness of the streams. Typically, there are two kinds of streams, linear and indirect. The indices of linear streams are linear combinations of nested loop variables (e.g. `a[(i*n+j)*m+k]`). The indices of indirect streams are expressions contains memory operation (e.g. `a[b[i]]`). The detection and encoding of these

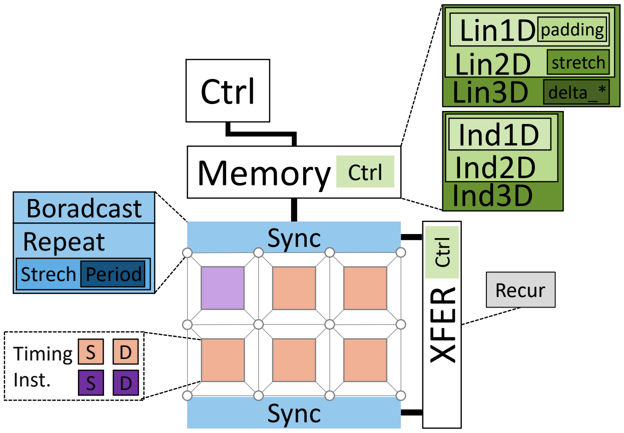


Figure 6.4: The ISA features that can be modularized

memory streams were already discussed in Section 4.3 & 3.3. The more dimensions/deeper loop nest it supports, the more complicated micro-architectural state machine it requires.

This graph serves not only as a serialized representation of the composed architecture, but also as an abstraction/specification for the compiler to understand the underlying hardware. The spatial mapper uses the information encoded in this graph to be aware of the topology of the network, and the capability of each processing element to map dataflow graph, and the compiler (refer next section for more details) inspects this graph to be aware of the modular ISA enabled to invoke each hardware-specialized transformation.

6.1.2 Limitations and Future Directions

Figure 6.2 shows the architectures can be composed of the primitives mentioned above. Beyond these architectures, there are still many architectures can or cannot be represented in ADG's. In this section, we discuss the bound of DSAGEN, including features that can be hypothetically added to DSAGEN as well as fundamental limitations.

Potential Features: When DSAGEN was developed, we had relatively limited support for compilation and architectural design space. We propose these following future directions.

- **Overlay:** The generated coarse-grain reconfigrable accelerator can be used as another level of overlay on the FPGA to save the expensive FPGA physical design and reconfiguration. This work was realized, and will be discussed in the next chapter.

- **Flexible Memory Topology:** When building this framework, we made very strong assumption on the memory-accelerator connection. It is possible to explore the topology among these coarse-grain components. This was also supported in our follow-up work, and will be discussed in the next chapter (Chapter 7), which requires extensions to the dataflow graph, and compiler to encode more domain knowledge.

- **Heterogeneous Cores:** Each host controller can only command one instance of the spatial accelerator currently. We can make the host controller connect to multiple instances of the accelerators.

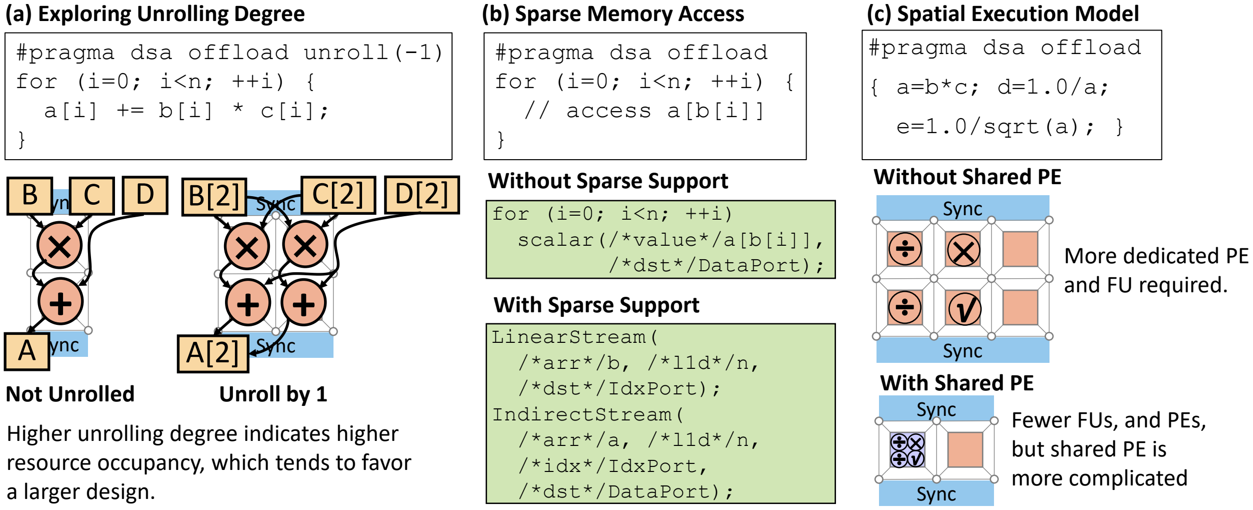
Fundamental Limitations:

- **Memory Consistency:** Because of the concurrent nature of the memory streams, we cannot enforce the orders among every single memory request. The compiler or programmer should be responsible to the coarse-grain memory operation.

- **Speculation:** Because of the decoupled nature of our system, the control command cannot be easily rolled back when mis-speculation.

6.2 Generating Multiple Intermediate Representation

As discussed before, the accelerator design demands are naturally encoded in each program behavior of interest, which will also be referred as idioms next. The compiler is aware of these software/hardware co-design opportunities, and these opportunities are reflected by performing different sets of transformations as it is shown in Figure 6.5. Here, we discuss three typical modular transformations that reflect hardware design demands.



(a) Exploring Unrolling Degree

```
#pragma dsa offload unroll(-1)
for (i=0; i<n; ++i) {
  a[i] += b[i] * c[i];
}
```

Not Unrolled

Unroll by 1

Higher unrolling degree indicates higher resource occupancy, which tends to favor a larger design.

(b) Sparse Memory Access

```
#pragma dsa offload
for (i=0; i<n; ++i) {
  // access a[b[i]]
}
```

Without Sparse Support

```
for (i=0; i<n; ++i)
  scalar(/*value*/a[b[i]],
         /*dst*/DataPort);
```

With Sparse Support

```
LinearStream(
  /*arr*/b, /*l1d*/n,
  /*dst*/IdxPort);
IndirectStream(
  /*arr*/a, /*l1d*/n,
  /*idx*/IdxPort,
  /*dst*/DataPort);
```

(c) Spatial Execution Model

```
#pragma dsa offload
{ a=b*c; d=1.0/a;
  e=1.0/sqrt(a); }
```

Without Shared PE

More dedicated PE and FU required.

With Shared PE

Fewer FUs, and PEs, but shared PE is more complicated

Figure 6.5: Compiler transformation to reflect design demands.

Unrolling Degree mainly reflects the demands of the computing resource. The higher unrolling degree the compiler adopts, the more hardware resources are required as it is shown in Figure 6.5(a).

Sparse Memory Access The compiler can generate code with the same semantics using different memory stream intrinsics. In the case shown in Figure 6.5(b), data will be fed scalar by scalar from a for-loop if the sparse memory engine is not available, which pressures the host controller to issue more instructions, while keeping the underlying hardware simple. On the other hand, if the sparse unit is required, the compiler can generate stream intrinsics uses those intrinsics.

Spatial Execution Model Section 2.1 already characterizes different spatial execution model, and REVEL discussed in Chapter 5 already demonstrates the promise of adopting suitable execution model for different code regions. The compiler tries to map the instructions to either shared PEs or dedicated PEs to reflect the demands of integrating different execution models.

Different design demands may lead to different software/hardware affinity. To rapidly evaluate how efficiently the underlying hardware is specialized for these transformations, several analytical models are developed. This will be explained in the next section.

6.3 Design Space Exploration

Our framework can perform automated codesign between the input programs and the hardware, selecting the best set of transformations along with a fine-grain selection of hardware features based on iterative search. The basic iterative approach to codesign as follows:

1. Start with the initial ADG and input applications.

2. The compiler generates multiple versions of transformations for different hypothetical sets of hardware features, and resource occupation.

3. At each iteration:

   - Make a number of random modifications on the ADG: a number of components are added or removed with random connectivity without exceeding the power and area budget.
   - Map all the versions of the applications onto the ADG.
   - Estimate the performance of the versions can be properly mapped onto the candidate hardware based on an analytical model.
   - Select the version with the best performance of each application and estimate the objective function.
   - If the objective function improves, continue with the new ADG.

4. Repeat the iterations until the objective function until the objective (perf^2/mm^2) converges.

To enable efficient design space exploration, we need to optimize the most time-consuming steps in the design space exploration, the compilation, and the software/hardware evaluation. In the rest of this chapter, we will discuss the insights and solutions to these challenges.

6.3.1 Spatial Mapping Repair

Because of the iterative nature of the spatial mapping algorithm, it becomes one of the most timing consuming steps in the design space exploration, especially when the hardware resources and datapath are highly tailored for the applications. The transformed applications stay stationary across each iteration of the design space exploration, and the modified ADG is not totally different from what it was in the previous iteration. Therefore, instead of remapping the applications onto the candidate hardware from scratch, we adopt a mapping repair algorithm. After the ADG is modified, only the portion mapped to the removed hardware resources is affected by the modification. Therefore, we only need to find new mapping for this portion. Compared with fully remapping, this partial repair inherits a highly iterated solution and make progress on this, so it will potentially yield better mapping with fewer additional iterations. Figure 6.6 shows how schedule repair saves the effort over rescheduling.

6.3.2 Analytical Estimation

It is not practical to synthesize the candidate hardware and run each application on it, considering the time consumed. We need an accurate and efficient way to predict performance-cost tradeoff of the hardware-software pair.

Performance Estimation As aforementioned, a main source of spatial architecture acceleration is the ILP achieve by the software pipelining, and the instruction execution is driven by the data availability. Therefore, we use the ILP as the metric of the estimated

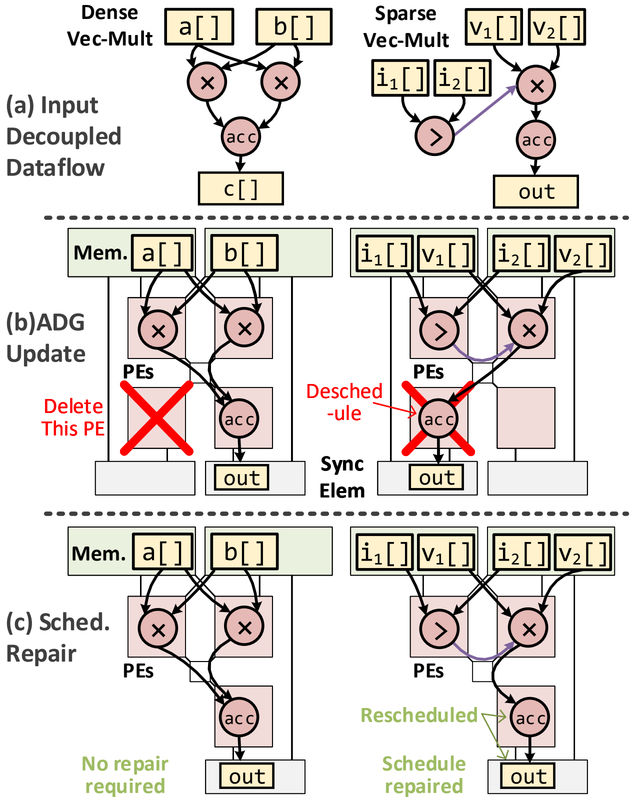


Figure 6.6: An example of DSE step with schedule repair

performance. To estimate the ILP, we only need to estimate the rate of data availability by analyzing each data source:

$$\text{perf} = \#\text{inst} \times \text{data rate}$$

For a stream from memory, the data rate can be computed:

$$\text{data rate} = \frac{\sum \text{data traffic}}{\text{memory bandwidth}}$$

where data traffic is the number of bytes required to sustain the data path of the spatial architecture.

For a recurrence stream, the data rate can be computed:

$$\text{data rate} = \min(1, \frac{\text{length}}{\text{latency}})$$

where length is the length of the stream, and the latency is the number of cycles from the input port to the result port. If we have enough instances fired to the spatial architecture to hide the latency of the datapath, the data rate is considered good.

When there are multiple offloaded regions, we use LLVM's block frequency module to estimate the relative execution rate of each offloaded region and use this as a weight to normalize the performance of each application.

Power/Area Estimation We estimate the power and area by simply adding up the power/area of each component predicted power/area interpolated models. Each component is synthesized alone with different parameters, and then we use these synthesized results to calculate a set of proper parameters for the models.

6.3.3 Hardware Generation

When the design space exploration is finalized, our framework can generate the hardware. Besides the RTL implementation, our framework also generates the ISA.

Feature-Oriented Interfaces As aforementioned, each hardware features can be integrated independently, so our framework may inspect the ADG to determine the hardware features demanded. For exmaple, if indirect memory is required in the generated hardware, address generation unit with indirect capability will integrated.

Bitstream Encoding Each component of the spatial architecture has its own local registers to store the bitstream that encodes the programmable information: A switch's bitstream encodes the routing table. A PE's bitstream encodes the instruction opcodes, execution timing (for static PE's only), and instruction tags (for shared PE's only). A synchronization element's bitstream encodes the cycles of delay. The spatial architecture is configured by loading the bitstream into these registers.

Config. Path Generation The ADG can describe arbitrary topologies, so a key question to answer is how the bitstream is loaded along with this topology. We define the problem as finding one or more paths that covers all the nodes in the ADG, and the longest path should be as short as possible. We use an approximate algorithm to solve this question. We first use a spanning-tree like algorithm to get multiple initial paths. Then we iteratively apply a heuristic: cut a node from the longest path and connect it to any nearby shorter paths. Iteratively repeat this process until the length converges.

6.4 Methodology

Compiler We build our compiler by customizing clang so that it can parse and encode the pragmas in metadata of an LLVM module. Then a customized LLVM pass is implemented to analyze and rewrite the code by taking advantage of this additional metadata. These transformed IR will generate assembly code and be assembled and linked by GNU binary toolchain.

Simulation We use a single-issue RISC-V core as our control host. We simulate it by extending gem5 RISC-V in-order core, and integrating a cycle-accurate spatial architecture simulator.

Target Accelerators We chose five accelerators to instantiate (approximately), to stress different hardware features:

- **Softbrain [193]** is instantiated using a mesh of static-scheduled/dedicated PEs and switches and a single non-banked scratchpad memory.

- **MAERI** [**154**] is approximated similarly to Softbrain, but with its novel tree-based topology.
- **Triggered Instructions** [**200**] is approximated with a mesh of dynamic-scheduled/temporal PEs. Our designs assume a group of PEs shares access to a decoupled scratchpad.
- **SPU** [**74**] is similar but has dynamic-scheduled/dedicated PEs, and banked scratchpad.
- **REVEL** [**276**] composes static-scheduled and dynamic-scheduled PEs in one mesh, and allows communication through synchronization elements.

In our experiments, we assume that accelerators are integrated to a high-bandwidth L2 cache (75 GB/s).

Synthesis We build a Scala-embedded DSL to generate the RTL implementation of the decoupled-spatial architecture, and synthesize the design in Synopsys DC 28nm library.

6.5 Evaluation

DSAGEN adopts the compiler discussed in Chapter 4, which was already evaluated, so we only evaluate DSAGEN's design space explorer and hardware generator. The major takeaways are:

- According to our estimation, the design space explorer is able to save 42% power and area over the initial hardware.
- The automated DSE generates hardware with mean 1.3× perf^2/mm^2 comparing with prior programmable accelerators across multiple sets of workloads.

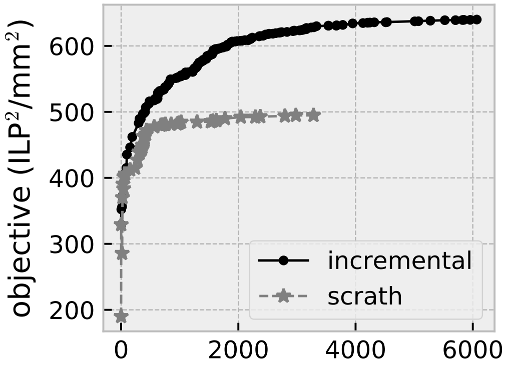


Figure 6.7: Repair versus Re-Mapping

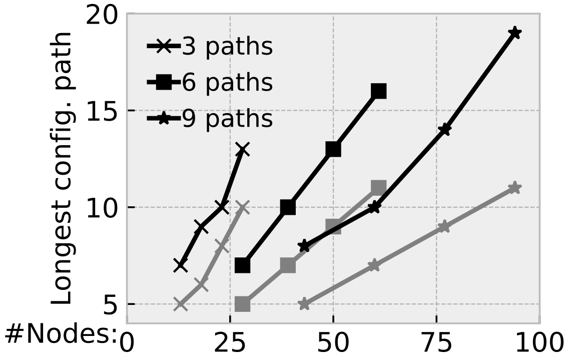


Figure 6.8: The Length of Configuration Path (gray: ideal, black: generated)

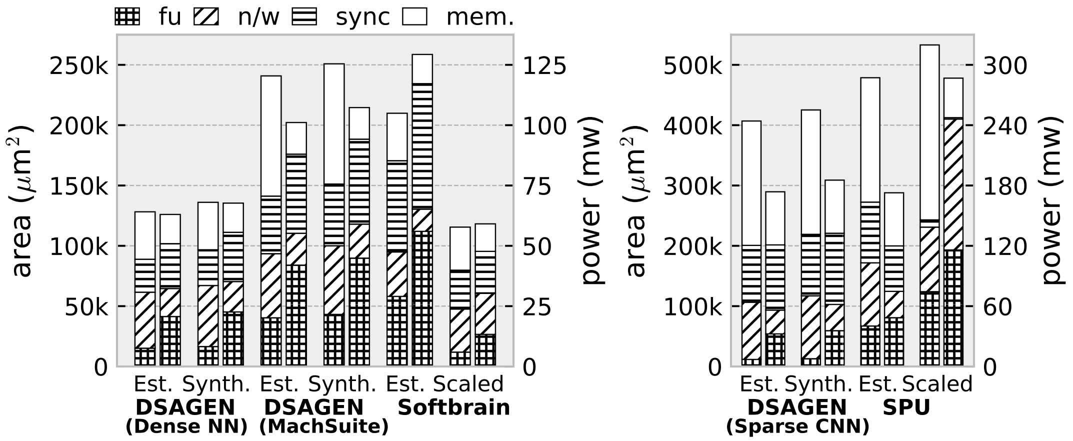


Figure 6.9: Comparing generated hardware to prior accelerators.

6.5.1 Design Space Exploration

The goal of our design space exploration is to demonstrate the ability to automatically tune the fine-grain hardware/software features and architecture topology. We evaluate three different sets of workloads:

- **MachSuite:** This set represents a variety of workloads with different needs and some irregularity. This allows us to compare DSAGEN's design (DSAGEN$_{MachSuite}$) against a hand-designed accelerator for these kinds of workloads: Softbrain [193].

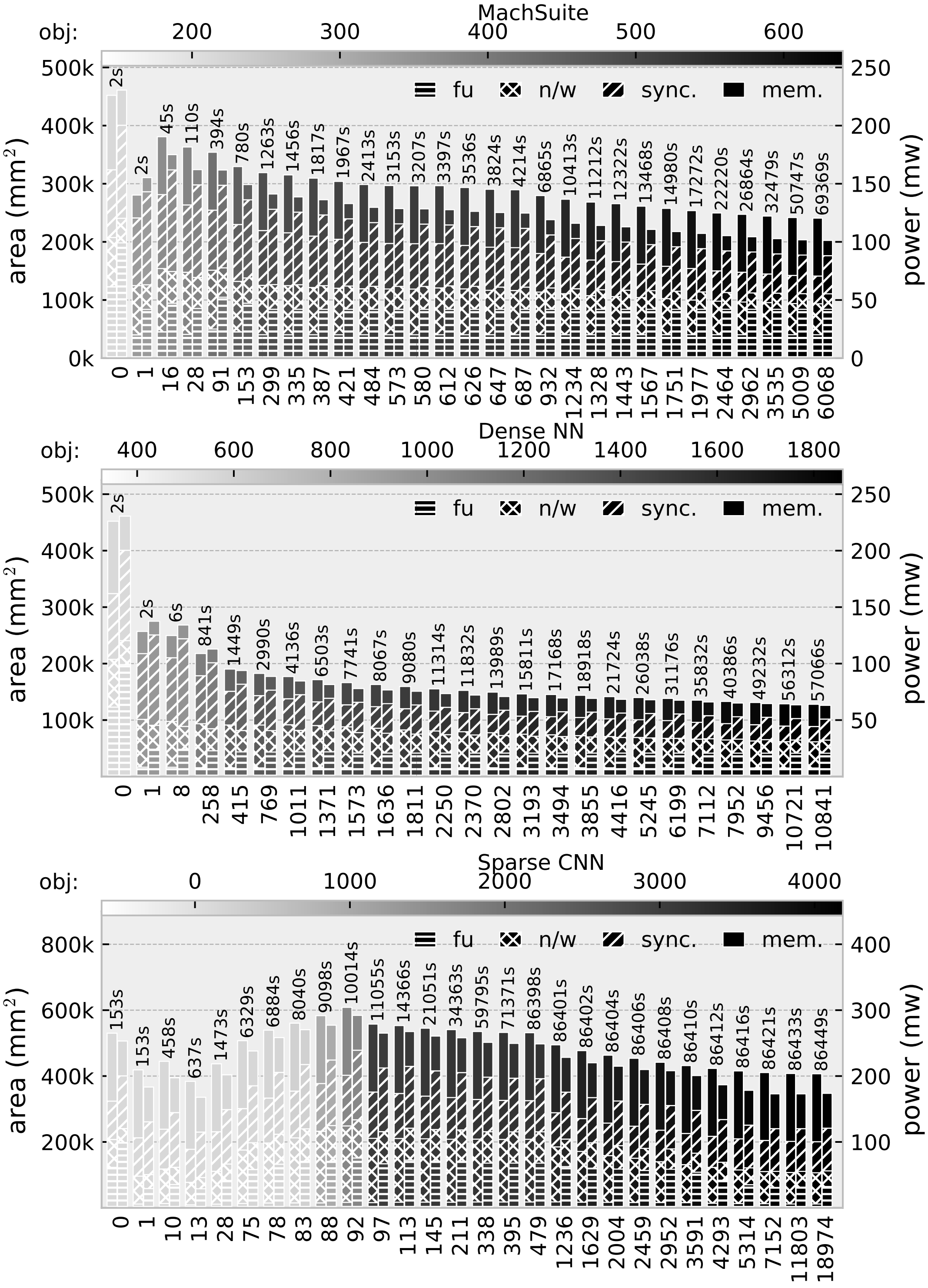


Figure 6.10: Automated Design Space Exploration

- **Dense neural networks:** We evaluate convolution, pooling, and classifier kernels, which have regular access and control. This allows us to compare the generated design (DSAGEN$_{DenseNN}$) against not only Softbrain, but also a domain-specific accelerator: DianNao [55].

- **Sparse convolutional neural network:** This is a single workload: outer-product multiply and re-sparsify. It has regular computation but data-dependent memory access. We compare DSAGEN$_{\text{Sparse CNN}}$ against SCNN [201] (a fixed accelerator) and SPU [74] (a programmable accelerator for sparse workloads).

We perform three DSE runs starting from the same initial hardware, a 5×4 mesh with full capability, including control flow, FU decomposability, and an indirect memory controller. The design space explorer estimates the performance, power, and area using the model discussed in Section 6.3, and the objective function is perf^2/mm^2. The explorer runs up to 200 scheduling iterations to initialize or repair the mapping after changing the hardware. The algorithm will exit after 750 iterations without objective improvement.

Figure 6.10 shows how the area (left bar), power (right bar), and overall objective (color intensity) evolve during design space exploration. The first two iterations initialize the exploration: after the datapaths are mapped to the initial hardware in the first iteration, the redundant features, including known unneeded functional units and address generation capability are removed. Because of the objective function, achieving better performance has higher priority than saving resources. Therefore, in these three DSE runs, the estimated performance is enhanced, and then the explorer trims redundant resources. It is hard to map sparse CNN's datapaths onto the initial hardware within a few iterations, so the explorer in the early iterations adds some redundant compute and routing resources to ease the difficulties of mapping. For MachSuite, the memory bandwidth is the bottleneck, so the explorer adds more initially. Subsequently the explorer focuses on enhancing the reuseability of on-chip network across multiple workloads, and minimizing the synchronization element depth.

Overall, our design space explorer saves mean 42% of the area and achieve mean 12× objective improvement over the initial hardware across the three selected sets of workloads.

Model Validation We validate our power/area regression model by comparing the numbers against synthesis. The results are shown in Figure 6.9. The bold label is the corresponding hardware, which is either a DSE generated hardware or existing reconfigurable accelerator. “Est.”, “Synth”, and “Scaled” stand for “estimated by the regression model”, “obtained by synthesis”, and “obtained from prior paper by technology scaling” respectively.

For the generated hardware, the estimate is 4-7% smaller than the synthesis area/power. While the model was tuned by synthesizing each component alone, extra structures are required to meet timing for the whole fabric. Our estimated model shows a somewhat large discrepancy between estimated and scaled area/power of Softbrain and SPU, which is partly due to microarchitecture[1] and technology/scaling differences. Further, some overhead may be due to having to provide more general protocols for modularity.

To validate the performance model, we simulate the generated hardware with the compiled programs after DSE. The model has mean performance error of 7%, with maximum error of 30%. The maximum error occurred in stencil-3d, because our model does not yet capture the performance impact of excessive control instructions.

Quality of the Generated Hardware Figure 6.9 shows comparison of DSAGEN designs with corresponding less-specialized programmable accelerators (Softbrain and SPU). According to our regression model’s estimation, $\text{DSAGEN}_{\text{DenseNN}}$ and $\text{DSAGEN}_{\text{Sparse CNN}}$ saves 64% and 18% area comparing against SPU and Softbrain for respective workloads. While $\text{DSAGEN}_{\text{MachSuite}}$ introduces 1.2× area overhead comparing with Softbrain, it also

[1] Softbrain’s design [193] assumed delay structures could be eliminated by the compiler, which prior work [192] found not to be true.

provides 1.2× speedup (favorable given the objective function).

We also compare against scaled domain-specific accelerators, DianNao and SCNN, for reference; this is not particularly accurate due to technology differences. $\text{DSAGEN}_{\text{DenseNN}}$ has overhead of 2.4× area and 2.6× power over scaled DianNao. $\text{DSAGEN}_{\text{SparseNN}}$ is 1.3× area and power over SCNN. We believe the overhead is mainly from reconfigurability. While these accelerators use a specific network (eg. tree in DianNao), DSAGEN's irregular network does not converge perfectly to these specific, perhaps optimal, topologies. There is still future work to be done to improve the design space exploration.

Schedule Repair We compare two different strategies, traditional scheduling (map entire dataflow every iteration) and our schedule repair approach. During DSE, after each iteration of the hardware update, both perform up to 200 scheduling iterations. The result is shown in Figure 6.7 for the MachSuite workloads. At the early stages, both strategies have a very close objective, because there are abundant resources on the hardware and scheduling is simple. Remapping the whole schedule can still succeed within 200 iterations. When the hardware resources become tight, the traditional scheduler cannot succeed on these more efficient designs, because it has to re-discover the entire mapping. Overall, schedule repair leads to a 1.3× better objective for DSE.

Configuration Path Improving the configuration time can aid performance of short program regions. Configuration time is dominated by the longest configuration path. We evaluate the path generator by giving it multiple mesh spatial architectures (2×2 to 5×5 PEs) under the constraint of having 3, 6, and 9 configuration paths, and the result is shown in Figure 6.8. The dashed lines are the ideal lengths (for a network with n nodes, p paths, the longest path cannot be shorter than $\lceil \frac{n}{p} \rceil$), and the solid lines are the actual lengths. The path generator only introduces mean 1.4× overhead versus the ideal.

6.6 Related Work

DSE for General Purpose Processors Custom fit processors [92] is a framework to build application-specialized VLIW designs. Somewhat related proposals target customized VLIW or superscalar pipelines through some codesign process [120, 117, 28, 184, 71, 72]. Another related work is for general purpose processors called Liberty [255], which uses a microarchitecture specification to generate simulators and perform DSE. Similar frameworks include Expression [110], UPFAST [197] and LISA [206]. None of these support spatial architectures.

Network Synthesis Network synthesis techniques enable customized network topologies based on workload properties. One example is SUNMAP [186], which performs network topology synthesis, and similar techniques have been developed for irregular network topologies [208, 265]. DRNoC [152] and Connect [199] are network generators tailored for FPGAs. DRNoC is particularly relevant, as it generates a network based on the application's task graph. These NoC generators only addresses network design without considering computation. Other works map applications onto potentially irregular NoCs [185, 124], but do not perform codesign search.

Accelerator Design Frameworks CGRA-ME [62] is a design framework for static-scheduled CGRAs. It uses a C programming strategy, and includes fast power and area models [191]. The framework has a generic spatial scheduler for any topology based on integer linear programming [266, 63]; this is too slow for DSE. Several works explore the design space for CGRAs, including ADRES [41] and the KressArray Xplorer [114]. Kim et al. develop a design space exploration framework tailored for DSP applications [141]. EGRA [26] is another template-based CGRA which supports compound functional units (we do too through composition). RADISH is a CGRA generator which uses genetic algorithms to search for compound PEs based on a corpus of applications [277]. Suh et al.

propose a CGRA with heterogeneous FUs, amenable to DSE [242]. APEX [175] explores the topology using a subgraph mining algorithm while keeping each processing element homoegneous. AutoSA [268, 301] generates programmable accelerators using systolic-array-like architecture as compute substrate, and their design spaces are limited to applications with static dataflow timing.

μIR [232] is an IR and framework for designing application-specific accelerators that exposes microarchitecture features as first-order primitives.

Key Differences: None of the above 1. have a design space including multiple execution models (e.g. dynamic+static scheduling, dedicated+temporal PEs), and 2. perform topology search to specialize the hardware datapath to a set of programs. To the best of our knowledge, DSAGEN is the first one that achieves these two.

Compilation for Heterogeneous Systems Prior work develops high-level abstraction for programming heterogeneous systems [53, 211, 146], but mainly focus on domain-specific languages instead of general purposed programming. Our framework can be integrated with existing high-level compilers in each one of these systems. Doing so would enable DAEGEN to serve as the middle layer for enabling many accelerators to targeted with the same compiler infrastructure.

The Spatial [146] compiler uses DSE to map parallel programs to FPGAs and the Plasticine [210] CGRA, with an optimizer called HyperMapper [187]. It is fundamentally orthogonal as it is targeting the compilation problem and not DSE of the architecture itself.

We were also inspired by the decoupled-spatial execution model to develop a unified compilation flow for the emerging tensorization idiom [274]. Because of the spatial architecture enables a programmable datapath, the offloaded computation can be arbitrary. However, for the tensorized instructions, the semantics of these instructions are fixed. This work adopts a reversed methodology comparing with the compilation for decoupled-spatial architectures. The computation and the memory accesses are decoupled and analyzed respectively to determine applicability and code organization.

6.7 Conclusion

To broaden the potential of acceleration, this work develops an approach and framework, DSAGEN, for programmable accelerator design automation. In this paradigm, an accelerator can be developed by composing simple spatial architecture primitives, and also be generated through automated codesign. Codesign works because the compiler can understand how best to use the simple primitives that are composed in an architecture description graph. Modular compiler transformations can robustly target accelerators with different ISA features, parameterizations, and topologies. Further, only traditional languages are required with relatively little programmer intervention.

More broadly, the field of computer architecture has historically grappled with what should be the layers of abstraction from hardware to software to enable efficient designs. A fixed ISA has been both the typical assumption and a persistent burden. This work suggests that the ISA does not need to be the hardware/software abstraction which designers rely on, at least for the domain of accelerators. Instead, a modular accelerator description can serve that purpose, and enable much greater flexibility to explore deeply specialized designs. This framework has been used for many use research projects [40].

This approach can also be extended and used to improve and revolutionize the existing FPGA programming, which will be discussed in the next chapter.

CHAPTER 7

OverGen: Improving FPGA Usability through Domain-specific Overlay Generation

In this chapter, a work [166] aims at reforming the existing FPGA programming paradigm is presented. By deploying a specialized programmable accelerator as another level of overlay on FPGA, under a unified high-level programming interface, orders-of-magnitude faster reconfiguration and compilation over the existing FPGA programming approach is achieved.

7.1 Motivation

FPGAs have been proven to be a very powerful computing platform. However, the existing FPGA programming paradigms impose significant challenge to developers. Developers have to manage excessive low-level details when writing register transfer language (RTL), including but not limited to timing, concurrency, and heterogeneous on-chip resources. An alternate is high-level synthesis (HLS). HLS tools can generate high-performance designs, by relying on hints provided by the developers. Hints are often encoded in a very complicated pragma system. State-of-the-art frameworks (e.g. AutoDSE [239]) explore these hints on behalf of the programmer. While highly effective, HLS limits programmer productivity because of high compilation/synthesis times. Moreover, HLS-generated FPGA designs are specific to either a single application or an application domain. When switching across different applications, loading bitstream takes significant time, taking more than a second to reconfigure modern FPGAs [225, 212].

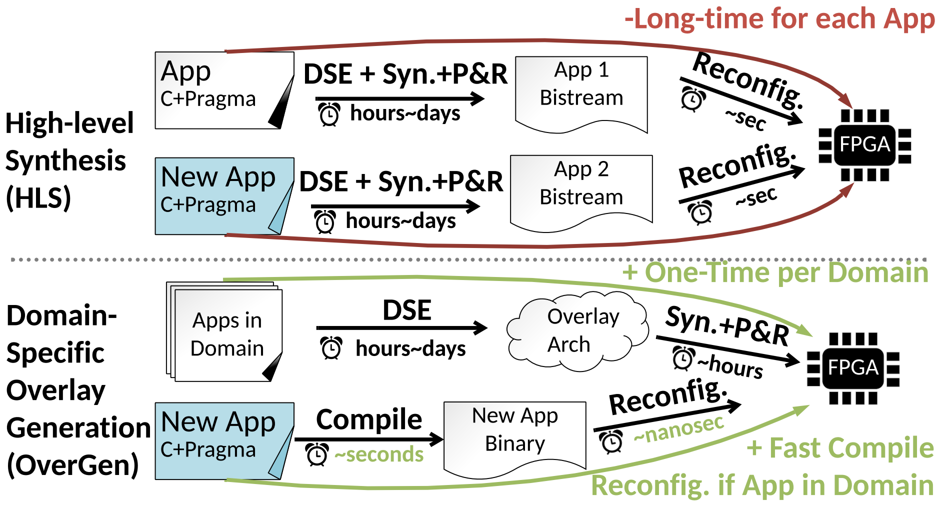


Figure 7.1: Overlay Generation Compared to HLS

Alternatively, FPGA overlays map a coarser grain architecture (e.g. CPUs [143, 156, 230, 227], GPUs [142, 21, 83, 24], CGRAs [204, 36, 19]) on top of the FPGA's fine grain abstractions. While overlays reduce compilation/synthesis time and are more general, they experience quite high overheads due to the abstraction gap between general purpose architectures and the low-level fine-grain abstractions exposed by FPGAs. Overlays can be customized with domain-specific extensions [3, 168], but this approach is highly time-consuming.

Vision and Requirements Our vision is to use an HLS-like approach where the generated hardware is tuned to input applications, but which targets a highly-flexible overlay architecture instead of a fixed-function pipeline. Figure 7.1 gives the basic idea, where a set of applications are fed to a design-space exploration (DSE) step to determine the ISA and resource provisioning in the overlay, and compiling a new application (and reconfiguring) is extremely fast. Ideally, small application changes would not require FPGA re-synthesis. We envision four requirements for overlay generation to be successful: 1. the overlay design space must include both system parameters and a broad accelerator design space, 2. it must balance generality versus specialization, depending on the degree of diversity in input applications, 3. the memory system itself should be highly specializable to the application, and 4. it has to get competitive performance with traditional HLS within reasonable DSE time.

Approach For the first two requirements, we leverage prior work on flexible multicore system generators (e.g. [23, 32]) and spatial architecture synthesis (e.g. [273, 248, 277, 33, 69, 246, 150, 30, 232]). Multicore system generators enable simple scaling in terms of cores, cache and network [23]. My prior study [273, 276] has proven the power of the decoupled-spatial paradigm, which is able to get deeply specialized for a wide range of applications while retaining flexibility with a high-level programming interface under moderate hardware overhead. Moreover, these decoupled-spatial architectures can be automatically generated to be specialized for the given applications, which well fits the goal of being a counterpart for HLS for FPGA programming.

To enable even deeper specialization, our primary insight is that data-reuse structures (e.g. DMA engines, scratchpads) must be incorporated into the spatial architecture design space to better ultize the FPGA memory system (requirement 3) – i.e. enabling a custom topology connecting reuse structures to compute structures. For the DSE to make good decisions, this requires the compiler to analyze and expose data-reuse analysis to the spatial-scheduler intermediate representation. We refer to this technique as spatial-memory exploration, which will be explained in further sections.

Finally, for requirement 4, we notice that significant time is spent on recompiling workloads as the hardware definition changes. We develop novel techniques for modifying the hardware while preserving the validity of previous compilations. We call these schedule-preserving transformations.

Implementation and Implications This work is called OverGen, which integrates two open-source frameworks, the DSAGEN [273] spatial architecture generation framework and the ChipYard [23] SoC generator, and extends these with support for FPGA resource modeling at the system level, novel hardware design space extensions, and novel algorithms for DSE-time reduction.

While much of this work is about the integration of previous ideas and existing frameworks (with some novel extensions), the results are profound: The evaluation suggests that domain-specific spatial overlays, and the OverGen framework specifically, have the potential to challenge HLS as the defacto FPGA design methodology. Our approach preserves a programmer-friendly interface with short compilation and reconfiguration times, and has competitive performance across many domains compared to the state-of-the-art HLS framework AutoDSE [239]. Across workload suites of DSP, Machsuite, and Vitis Vision, OverGen achieves geomean speedups of 1.21×, 1.13×, 1.25× speedups over baseline AutoDSE without kernel tuning, and it still reaches comparable performance, 0.71×, 0.37×, 0.65× respectively, with manual kernel tuning for AutoDSE. Our approach also enables overlays that support single or multiple workloads by *automatically* reasoning about the cross-workload flexibility.

In the rest of this chapter, the novel extensions on DSAGEN we made to enable this revolutionizing FPGA programming paradigm will be elaborated, including the extended design space, software interfaces, as well as the design space exploration.

7.2 Extension Overview

Here we overview OverGen's extensions span compilation, design space exploration, and resource modeling, and then overview the design space and key tradeoffs.

7.2.1 Overview

Compilation Figure 7.2 shows the overview of OverGen. We begin with the compilation flow, which takes the system-level ADG (sysADG) as input. The sysADG defines the spatial accelerator and system design spec, and is created during overlay generation (described later). The programming interface of OverGen is multithreaded C with aforementioned pragmas (details in Section 7.4.5). The LLVM-based compiler will attempt to create the highest-performance dataflow graph for the spatial accelerator using its knowledge of the available hardware features in the sysADG. The compiler extracts the memory access and

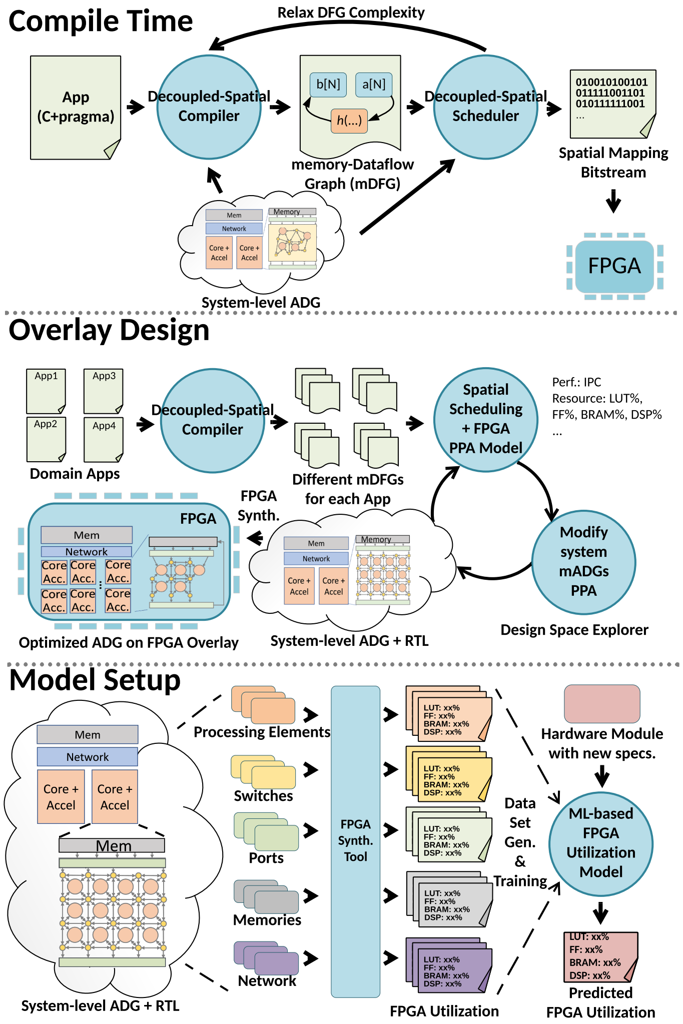


Figure 7.2: Overview of OverGen Framework

computation from the program to construct a Memory-enhanced Dataflow Graph (mDFG); the mDFG is enhanced with information about array size, suitability for mapping arrays to scratchpads, and data reuse of each stream. The program represented as an mDFG is then mapped onto the ADG by the spatial scheduler, using the reuse information to make informed decisions. The mDFG could fail to map to the hardware; if so, the compiler will "relax" the DFG complexity by using less aggressive transformations (e.g. reduce unrolling degree [275]).

Overlay Generation The input to the overlay generation is a set of workloads which forms the domain of interest. It is too inefficient to redo the compilation with each step of DSE. As we discussed before, the applications naturally encode the design demands of the generated hardware. To better utilize the underlying FPGA memory resource, the compiler generates a set of different mDFGs representing program versions that could be useful. These extracted mDFGs are incrementally recompiled during DSE.

Compiled mDFGs are used to guide spatial-accelerator synthesis: all mDFGs are scheduled to an ADG, and the ADG is iteratively updated to maximize the objective (mean performance of the best-performing mDFG for each workload). There are four innovations over prior work: 1. the system and spatial accelerator are co-designed; 2. reuse and array information enables reasoning about memory and cache allocation at the spatial level, and 3. DSE balances FPGA resource utilization, and 4. the mDFG resource utilization guides ADG transformations with schedule-preserving transformations, described in Section 7.4.2.

Finally, the chosen sysADG will then be lowered to synthesizable RTL for the FPGA, in part leveraging hardware generators from DSAGEN [273] and ChipYard [23]. DSAGEN's microarchitecture implementation is enhanced to enable pipelining on FPGAs with tight cycle-time constraints.

Model Setup Our FPGA resource utilization model is based on per-hardware element models. Elements with a few parameters (e.g. core) can be exhaustively synthesized. For elements with many parameters, we use a machine-learning (ML) based model, trained from synthesizing a representative design space, which is proven to be accurate and effective [280]. Leveraging learned models means that this framework can more easily be ported to other FPGAs.

7.2.2 Overlay Design Space

Accelerator Design Space This accelerator design space is very similar to the design space discussed in the last chapter. By connecting different hardware components, including but not limited to *processing element*, *memories*, and *recurrence bus*, a decoupled-spatial architecture can be composed. Similarly, each component can have different parameters to enable a tradeoff among *cost*, *flexibility*, and *performance*. Refer prior Section 6.1 for more details.

System Design Space This is one of key differences from DSAGEN. The target overlay is a homogenous multi-tile (i.e. multicore) where each tile contains an instance of the spatial accelerator, associated with a light-weight control core where DSAGEN can only synthesize single-core systems. Because we target highly-acceleratable workloads, the control cores are kept simple (single issue, small private cache), and are only provisioned for managing accelerator execution. The control cores and accelerators share access to a shared L2 cache over a crossbar-based NoC. Overall we explore the number of tiles, NoC bandwidth, L2 banks (for controlling L2 bandwidth), and L2 capacity.

7.2.3 Key tradeoffs

OverGen opens a variety of tradeoffs that were previously difficult to explore and would have required manual effort. On the system side,

Big tiles vs. More tiles Many acceleratable workloads benefit from vectorization, while others are difficult to vectorize due to irregularity or loop dependencies. This leads to a tradeoff where some domains prefer more small accelerators (less pipeline/vector parallelism) or fewer large accelerators (more pipeline/vector parallelism).

L2 cache size vs. Scratchpad capacity Some workloads have regular access to private data that can map to scratchpads, while less regular codes often benefit from hardware managed caches. Each domain requires a tailored allocation.

Balancing Bandwidths The overall design space has essentially three levels of memory hierarchy, from shared cache to spatially distributed scratchpads, and reuse in the computation units. Allocating bandwidths across these levels requires understanding the compute bandwidth and data reuse possible in the chosen workloads — these decisions are tightly coupled with accelerator size and number of tiles.

Compute Density vs. Generality If the goal of the overlay is to support either many workloads or dissimilar workloads, a more general overlay is required. This tradeoff can be made by constructing a flexible datapath at the cost of more resources, thus affecting all of the above tradeoffs.

7.3 Spatial Memory Exploration

7.3.1 Motivating Spatial Memory DSE

Prior spatial architecture synthesis algorithms assume that all memory elements (scratchpads/DMAs) can communicate with all computation elements. While this simplifies the design space and spatial scheduling, it also prevents the DSE from exploring the best way to connect memories and processing elements together. Figure 7.3(a) shows an example design where memory stream engines communicate over essentially a crossbar to the spatial compute units. Figure 7.3(b) shows the potential of a system that allows spatial memories, where these engines have local communication with a smaller subset of elements. Similarly, extending this design space enables the possibility of deciding between multiple smaller scratchpads or a single unified scratchpad.

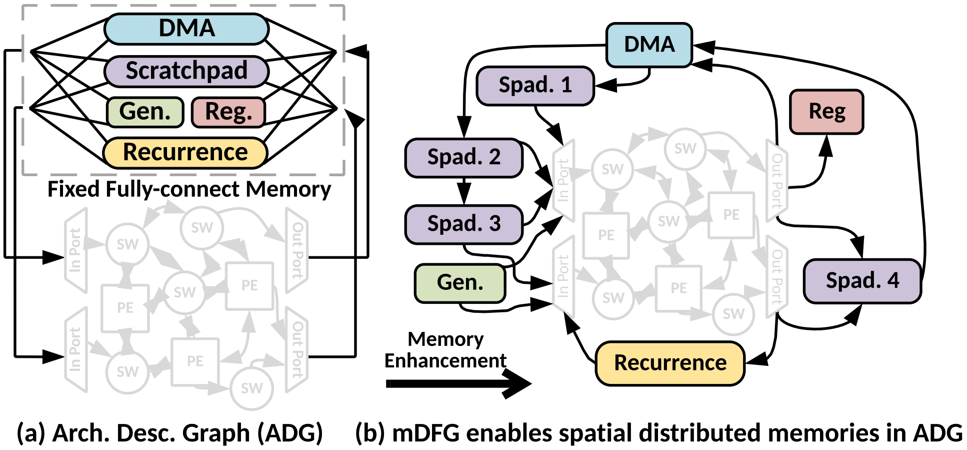
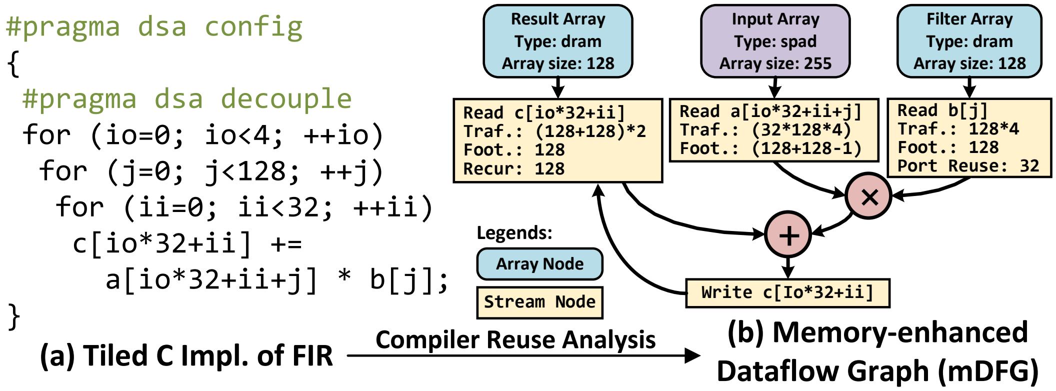


Figure 7.3: Spatial Memory Enhancement for ADG

Making these decisions with existing DFG abstractions is difficult, as they lack two key pieces of information: 1. the relationship between access patterns and *data structures*, and 2. the size and reuse of these data structures. Together, these can enable reasoning about the validity and performance of spatial memory optimizations.

Memory-enhanced DFGs (mDFG) We enhance DFGs with data structure and reuse information by introducing *array nodes*, creating what we call a memory-enhanced DFG (mDFG). Array nodes have edges to streams that consume or produce those arrays, and we include reuse properties on streams. An example is in Figure 7.4 for a simplified version of `FIR`. Here, the input array `a` is stored in scratchpad for higher bandwidth requirement and reuse. The size parameter describes the total size allocated in either DRAM or scratchpad. If it is in scratchpad, the additional space of double-buffering is included. Also, streams are annotated with additional information for computing the reuse factor, including data traffic, data footprint, *stationary reuse*, and *recurrent reuse* (see the "Reuse Analysis" paragraph).

There is now sufficient information in the mDFG to decide which scratchpad to use — i.e., if data can be routed between the scratchpad node in the ADG and PE nodes that consume this data, and if there is enough remaining space in the scratchpad. If there is ever a limited capacity, the reuse information can help determine *which* array node in the



```
#pragma dsa config
{
 #pragma dsa decouple
 for (io=0; io<4; ++io)
  for (j=0; j<128; ++j)
   for (ii=0; ii<32; ++ii)
    c[io*32+ii] +=
      a[io*32+ii+j] * b[j];
}
```

Figure 7.4: Memory Reuse Enhancement for DFG

mDFG should be mapped to a scratchpad node; for example, if an array has a stream with *stationary* reuse at the port, the benefit of exploiting reuse at the scratchpad level could be less than another array without stationary reuse. Note that the reuse information in the mDFG will also be used in the DSE for making system-level design decisions (Section 7.4).

7.3.2 Software Support for Spatial Memory

To implement spatial memories, we extract array and reuse information from the program, and embed this in the mDFG to utilize during spatial scheduling.

Array Node Extraction As it was discussed in Chapter 3, all memory operations under the `stream` pragma are "restricted" (alias free), so we can extract the arrays involved in the dataflow graph by analyzing the pointer expressions. Specifically, we extract all the array pointers that are transitively used by all the decoupled memory operations. Consider the example in Figure 7.4(a): `a`, `b`, and `c` are extracted as array nodes. An array node has three attributes: pointer, footprint, data traffic, and memory reuse.

Reuse Analysis Being aware of memory behaviors that can be captured by hardware specializations helps both compiler optimization and DSE. Our compiler recognizes these patterns and annotates them on the associated stream nodes. Next, we discuss three typical reuse patterns, *general*, *stationary* and *recurrent* through the example in Figure 7.4(a) and (b).

General Reuse refers to when a memory stream repeatedly accesses a set of data within a program region. Scratchpad is often favored to exploit this reuse, provided there is sufficient capacity. Reuse can be identified by finding a discrepancy between data footprint (array or tile size) and traffic (number of uses). Consider the operand `a[i*32+ii+j]` from the innermost loop; the compiler recursively analyzes and joins the memory boundaries touched by each loop, and finally computes that 255 elements are in the memory footprint. To compute the data traffic, the compiler notes that every loop variable is involved in this pointer expression, which means a different element is accessed in each iteration. Thus, the data traffic of this operand is computed by multiplying all loop trip counts, i.e. $32 \times 128 \times 32 = 16384$. This indicates that each element is reused an average of $\frac{16384}{255}$ times. Indirect memory access, e.g. `a[b[·]]`, can also be analyzed similarly. To simplify, we assume: 1. `b[·]` is linear and can be analyzed by the above techniques; 2. no memory access will overflow, and the indirect memory access is a uniform distribution over array `a`. Therefore, data traffic is calculated by multiplying loop trip counts, and the data footprint is the size of array `a`.

Stationary Reuse refers to an operand repeatedly reused across the innermost loop so that this operand can be *stationary* in the compute substrate (e.g. the port FIFO). Consider the `b[j]` operand: Because the innermost loop `ii` does not involve the pointer expression, this value is reused across loop `ii` 32 times. Even though `b[j]` also has *general reuse*, it does not provide as much value to map to scratchpad, because much of the reuse is captured as stationary reuse (i.e. in the port).

Recurrent Reuse refers to when a pair of memory streams repeatedly update a set of data. When this set of data can concurrently fit in the data path pipeline and port FIFO, this pair of streams are favored to use the recurrence stream engine to avoid memory traffic. Consider the `c[io*32+ii]`: it repeatedly reads and writes a set of memory touched by `ii` (i.e. 32 concurrent instances) along with `j` (i.e. 32 recurrences). Therefore, when there is enough on-chip buffer for these 32 concurrent instances, this pair of streams will be mapped to the recurrence stream engine.

To sum up, reuse behavior captured by scratchpad, port FIFO, and the recurrence stream engine will all be considered as the reuse factor; this factor is used to calculate the bandwidth pressure of each stream in the DSE performance model (Section 7.4.3).

mDFG Scheduling Enhancing any spatial-scheduling algorithm to support mDFGs is straightforward. The principle is to treat array nodes (the ones representing the data structure), as any other node which must be scheduled onto the ADG, but with unique scheduling constraints. Intuitively, an array node can be mapped to a memory stream engine if:

1. There is sufficient remaining space (for a scratchpad).
2. There is a legal route from producers to consumers.
3. The access pattern of all streams for the array node is supported by the stream engine (e.g. indirect access).

The stream engines in our implementation allow more than one array each (as they support multiple concurrent streams), provided there is sufficient capacity. The tradeoff is that the bandwidth must be shared between any associated streams. Thus, even if it is legal to map more than one array to a scratchpad, it is sometimes beneficial to avoid sharing by using a different scratchpad or even just placing the array node onto a DMA stream engine; this can help maximize the utilization of available bandwidth.

Having reuse info on streams can help resolve these choices. For example, array nodes with stationary reuse at ports (e.g., read the same value X times) provide less benefit when mapped to scratchpads than those array nodes without stationary reuse – this is because their bandwidth consumption is already reduced. These factors must be considered during spatial scheduling; thus, we modify the objective of the spatial scheduler to use the projected performance of the mDFG, which factors in reuse and bandwidth bottlenecks. Because this is a critical portion of the system-level DSE, we explain the performance model in the next section (Section 7.4.3).

7.4 Unified System & Accelerator Design Space Exploration

The goal of DSE in OverGen is to codesign the system parameters and accelerator features/topology to maximize FPGA performance on the set of input applications. Here we first give an overview, then discuss a novel technique to use prior schedules to guide spatial DSE, and finally discuss the performance and area modeling techniques.

7.4.1 Overlay Design Exploration

Logically, one iteration of the DSE involves proposing a new ADG for the hardware, recompiling all the workloads to it, and evaluating an objective (performance and FPGA resource use) to guide the next step of DSE — repeat until convergence. We use three main strategies to reduce the time for each DSE iteration.

First, we attempt to avoid recompilation as much as possible. During standard compilation, the compiler will iteratively back-off from aggressive transformations that require more resources than available (e.g. reduce the vector width and recompile). To avoid this during DSE, the compiler pre-generates different mDFGs for each program region which each use different transformations (different unrolling degrees, use a recurrence stream instead of accumulation, etc.). These different mDFGs are maintained during DSE, and ultimately only one of them needs to be used (only one has to schedule correctly to the ADG). While this increases the up-front cost for the first DSE iteration, it eliminates from-scratch recompilation during DSE.

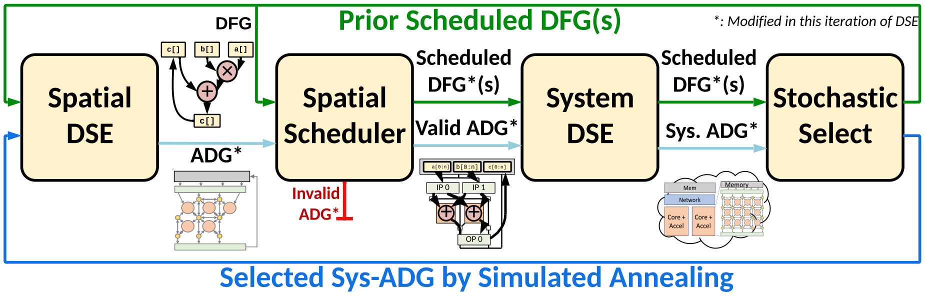


Figure 7.5: OverGen's Unified DSE Flow

Next, we also try to avoid the expensive spatial-scheduling stage of compilation by reusing the mDFG-to-ADG schedules from the prior iteration of DSE. A simple approach is to only re-schedule the portions of the DFG mapped to ADG elements that were modified (i.e. schedule repair [273]). In addition, we can use information about prior schedules to make a more informed decision about how to modify the ADG (see Section 7.4.2).

Finally, we leverage the disparity between spatial scheduling time (very high) and system-level design-space exploration (quite low). Rather than explore both ADG design (spatial DSE) and system parameters (system DSE) at the same level of the DSE, it is relatively inexpensive to nest system DSE inside of spatial DSE – i.e. run a full exploration of system parameters every time we modify the ADG. This improves the convergence of the overall DSE.

DSE Flow Summary The overall DSE flow is in Figure 7.5. At the beginning of each DSE iteration, the spatial DSE will propose a new ADG named ADG*. ADG* is constructed using a combination of random and schedule-preserving transformations (Section 7.4.2). Then, mDFGs are rescheduled onto ADG*, leveraging the prior schedules for any unchanged portions of the ADG. If any program region has no successfully scheduled mDFGs, then ADG* is abandoned, and a new iteration begins. If not, then the system-level exploration (system DSE) exhaustively searches for the best system-design parameters for ADG* (creating sysADG*) based on estimated performance and resource constraints.

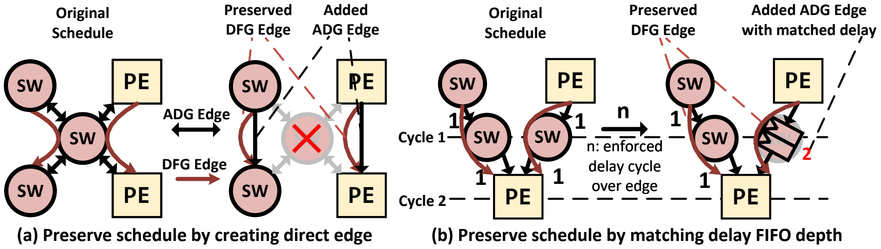

Original Schedule
SW
PE
SW
SW
PE
ADG Edge
DFG Edge
Preserved DFG Edge
Added ADG Edge
SW
PE
SW
PE
(a) Preserve schedule by creating direct edge
Original Schedule
Cycle 1
Cycle 2
SW
PE
SW
SW
PE
1
1
1
1
n
n: enforced delay cycle over edge
Preserved DFG Edge
Added ADG Edge with matched delay
SW
PE
SW
PE
1
1
2
(b) Preserve schedule by matching delay FIFO depth

Figure 7.6: Schedule-preserving Transformation Examples

The objective function favors estimated performance first (Section 7.4.3), followed by estimated resources-per-accelerator (Section 7.4.4). This secondary objective encourages the spatial DSE to prune unneeded resources in the ADG, even if it does not lead to more cores or higher performance in the current DSE iteration. The final step is to choose whether to continue with this ADG*, which is done stochastically through a simulated annealing approach.

7.4.2 Schedule Preserving Transformations

During each DSE iteration where the Spatial DSE randomly modifies the hardware, it is common that some of the compiled DFGs can become invalidated due to hardware deletions or resource reduction. While this can sometimes be rectified by repairing the schedule to use other resources, it often cannot be. In these cases, the DSE algorithm either has to use a lower-performance schedule, less-vectorized DFG. The repair itself also takes a significant amount of time. This is unfortunate, because this can even happen when deleting units that are not necessary: e.g. a switch that is only used to pass through a value without requiring flexible routing.

Thus, we introduce the concept of schedule-preserving transformations, which use prior DFG schedules to guide hardware modifications that preserve their validity. Schedule preserving transformations are defined as hardware modifications that simplify the ADG while adding back the minimum capability to support the existing schedules. Thus, in essence, schedule-preserving transformations increase hardware utilization, providing further incentives for the removal of hardware units that provide less value. Specifically, we identified three such transformations:

Node Collapsing, as shown in Figure 7.6(a), occurs when a unit which performs routing (e.g. a switch) is deleted. Here, after the routing node is deleted, any routes on existing schedules that went through the node are used to define new direct hardware connections from their source to their destination. Thus, this transformation preserves prior schedules by ensuring a valid path for routes through a deleted unit.

Edge Delay Preservation, as shown in Figure 7.6(b), preserves the pipeline depth of all operands for a PE when an intervening routing node is deleted. A balanced pipeline depth ensures that all operands arrive at the same time to avoid pipeline bubbles; these bubbles can lower the throughput of the spatial accelerator [192]. Our approach is to increase per-operand FIFO-depths in the PE (called delay-fifos) whenever this imbalance can be observed on an existing schedule.

Module-Capability Pruning prunes excess module capabilities, and associated hardware, that are not needed by mapped schedules. Without this transformation, the DSE oftentimes does not have enough incentive to remove some costly capabilities, and more frequently removes capabilities that are actually useful.

7.4.3 Performance Model with Spatial Memory

To estimate performance, we implemented a bottleneck-based analysis that captures system-level design parameters, memory bandwidth at different layers, and computational bandwidth. Specifically, the overall performance is calculated as the weighted geometric mean of the estimated IPC for each mDFG. An mDFG's IPC is calculated by multiplying the maximum instruction bandwidth (mDFG Insts) by the number of tiles and by the lowest bottleneck factor of all levels of the memory system:

$$\text{Perf} = (\text{mDFG Insts}) \cdot (\text{\# of Tiles}) \cdot \min_{L_1 \ldots L_3} \left(\frac{\text{R}_{\text{Production}}}{\text{R}_{\text{Consumption}}}\right) \tag{7.1}$$

Hardware Unit	Total Synthesized
Processing Elements	100,000
Switches	56,700
Input Port	34,412
Output Port	25,796

Table 7.1: Number of Hardware Modules Synthesized

The *mDFG Insts* factor captures vectorization degree, allowing the DSE to explore tradeoffs between higher vectorization degrees and number of tiles. Memory operations, namely load and store operations, are included within the estimated IPC to ensure that vectorization of pure data-movement DFGs is incentivized.

While counting loads and instructions estimates the ideal IPC, memory bandwidth limitations reduce the observed IPC, as memory subsystems cannot always supply enough data to fulfill computational requirements. We compute the most-bottlenecked performance reduction over L_1, L_2, and L_3, corresponding to the Scratchpad, L2 Cache, and DRAM. This bottleneck factor is calculated by dividing the production and consumption rate (as $\frac{R_{\text{Production}}}{R_{\text{Consumption}}}$ in the previous equation). These factors are calculated as follows, taking into account stream reuse factors (see Section 7.3.2):

$$\begin{aligned} \text{R}_{\text{Production}} &= BW_{L_N, L_{N+1}} \cdot (\text{\# of Banks}) \\ \text{R}_{\text{Consumption}} &= \sum_{i=1} (\frac{\text{BW}(\text{Stream}_i)}{\text{Reuse}(\text{Stream}_i)}) \cdot (\text{\# of Shared Tiles}) \end{aligned} \tag{7.2}$$

In the above equations, the production rate is computed by multiplying the bandwidth and bank count at each memory level. The consumption rate, or data needed to satisfy compute bandwidth, is the sum of compute data required by a single tile, multiplied by the number of tiles at that memory hierarchy level. The single-tile required data is computed as the summation of all stream bandwidths divided by their associated reuse rates. We describe how the bandwidth (BW) and reuse factors are computed at each level:

Scratchpad Bandwidth With scratchpads replicated across tiles, the *# of Shared Tiles* factor is one, making the bandwidth only depend on vectorization degree. Also, the bandwidth is calculated separately for the read and write port.

L2 Bandwidth As L2 Bandwidth is shared amongst tiles, the consumption rate increases with respect to tile count, requiring more banks. Accesses to L2 cache occur when a stream pattern cannot be supported by port reuse or recurrent data stream, without which the required data production rate will be increased – thus demanding more L2 Banks.

DRAM Bandwidth Similar to L2 bandwidth, the consumption rate is dependent on both reuse and tile count; however, the total FPGA's DRAM bandwidth is fixed.

7.4.4 ML-based FPGA resource model

To rapidly predict FPGA resources, the DSE leverages a machine-learning (ML) resource prediction model, which estimates resources on a component-level basis. To generate the ML model, we perform out-of-context synthesis on variations of each hardware unit, shown in Table 7.1, to train an ML-based FPGA resource model. The component-level ML model implements a 3-layer multi-layer perceptron (MLP), with an 80%/10%/10% test, train, and validation data split. As the FPGA resource model was synthesized out-of-context with no synthesis optimization passes being performed, our model behaves pessimistically – the projected design point is larger than the actual post-PnR result.

7.4.5 Limitations & Future works

Threading Interface The current pthread-like programming interface assumes a one-to-one mapping of threads to tiles, where threads run to completion uninterrupted. Also, the performance models assume that all tiles are parallelizing the same code region, and this is our convention when implementing kernels. We also do not manage the interaction between host and FPGA in terms of offloading or data movement. A more sophisticated programming interface, task model (e.g. [188, 75, 73, 245]), and analytical models could significantly expand usability.

Processing Elements Our current implementation of processing elements only supports a dedicated instruction execution model; in contrast, the use of *shared* PEs (either static [178, 173] or dynamically scheduled [276, 200, 76]) can potentially support kernels with larger code regions and get higher utilization for kernels with more complex control flow.

Compilation Support Although our processing elements already support a predication-based control lookup table for conditional execution, our compiler has only limited support for converting arbitrary control flow to predication based dataflow execution. A more general dataflow control flow model (e.g. [101, 136]) is future work. Meanwhile, our compiler only supports data parallel loop unrolling when exploring DFG resource occupation (i.e. DFG size). When it comes to exploiting overlapping data reuse between subsequent loop iterations, we still require manual unrolling to take advantage. This can be improved by integrating prior work on reuse distance analysis [68]. Also, our reuse analysis relies on strong assumptions on compilation-time determined loop trip count and array shape. One of our future directions is to support dynamical array shape and loops.

7.5 Methodology

Benchmarks We selected 19 workloads from different domains: 9 from Xilinx Vitis computer vision library, 5 from the *digital signal processing (DSP)* domain targeted by REVEL [276], and 5 from MachSuite [216] for commonly-accelerated workloads. The data size and data type are shown in Table 7.2.

	Workload	Size	Type	#ivp	#ovp	#arr	#m,a,d
DSP	cholesky	48^2	`f64`	7	3	2	5,4,2
	fft	2^{12}	`f32x2`	3	1	2	4,8,0
	fir	$2^{10} \times 199$	`f64`	4	2	2	4,4,0
	solver	48^2	`f64`	4	2	2	4,4,1
	mm	32^3	`f64`	4	3	3	4,4,0
MachSuite	stencil-3d	$34^3 \times 8$	`i64`	7	1	2	4,12,0
	crs	494×4	`f64`	6	5	6	1,0
	gemm	64^2	`i64`	4	2	3	8,8,0
	stencil-2d	$66^2 \times 3^2$	`i64`	3	1	2	9,11,0
	ellpack	494×4	`f64`	4	3	4	4,4,0
Vision	channel-ext	$128^2{\times}4$	`i16`	1	1	2	0,0,0
	bgr2grey	$128^2{\times}4$	`i16`	3	1	2	16,32,4
	blur	$128^2{\times}4$	`i16`	3	1	2	0,52,8
	accumulate	$128^2{\times}4$	`i16`	2	1	2	0,16,0
	acc-sqr	$128^2{\times}4$	`i16`	2	1	2	16,16,0
	vecmax	$128^2{\times}4$	`i16`	2	1	3	0,16,0
	acc-weight	$128^2{\times}4$	`i16`	5	1	2	32,16,4
	convert-bit	$128^2{\times}4$	`i16`	3	1	2	0,32,0
	derivative	$130^2{\times}4$	`i16`	3	1	2	16,32,4

Table 7.2: Workload specification: size, data type, input/output ports, and multiply, add, div ops in the best DFG.

Baseline We evaluate OverGen in terms of speedup, compilation, DSE time and device reprogram time. We compare against the state-of-the-art HLS technology, AutoDSE [239], as our baseline by using Merlin Compiler (2020.3) and Xilinx Vivado (2020.2). Because AutoDSE benefits significantly from manual kernel tuning, we evaluate both against non-tuned and tuned code versions for AutoDSE.

Compiler support We augment the open-source DSAGEN [273] compiler with spatial memory support. An extended Clang and LLVM compiler transform the pragma-annotated program into RISC-V assembly, and the RISC-V GNU toolchain is modified for binary generation.

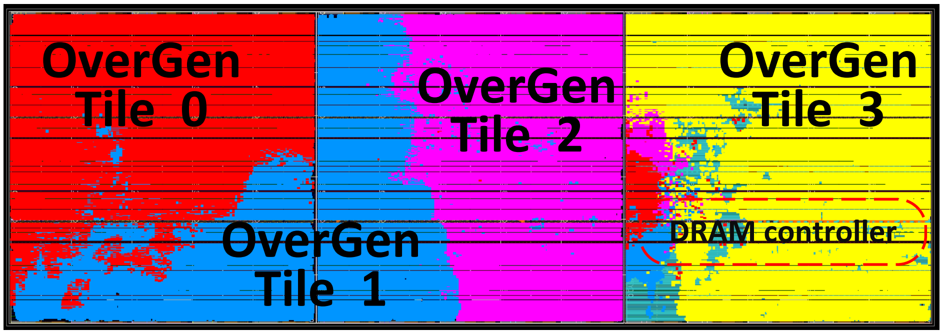


Figure 7.7: Quad-Core OverGen FPGA Floorplan

Hardware Generation & Verification OverGen augments the Chisel-based DSAGEN hardware generator [273] by extending it to full system-level with a modular spatial memory system as described in Section 7.3. After obtaining RTL from hardware generation, we further verify the functional completeness as a full system with RISC-V binaries on RTL cycle-level by using Synopsys VCS before FPGA verification.

System-Level Integration & Experiment Platform Each accelerator is integrated into ChipYard [23] as an RoCC accelerator to a small RISC-V Core (Rocket Core). All designs use an 8-way associative directory-based inclusive L2. The generated RTL is further synthesized to Xilinx VCU118 Evaluation board by using Vivado 2021.2. All data for each kernel begin offchip and are loaded from FPGA DRAM.

Because of FPGA implementation difficulties, we were not able to run on our FPGA when multiple DRAM channels were enabled. Thus, we use a single DRAM channel for most experiments, and study the effect of multiple DRAM channels separately using VCS RTL simulation (Eval. Q7).

Figure 7.7 shows the floorplan of a Quad-tile General OverGen design at 92.87MHz, including the DRAM controller's location. The critical path is around the L2 MSHR logic, and optimizing is beyond the scope of this work.

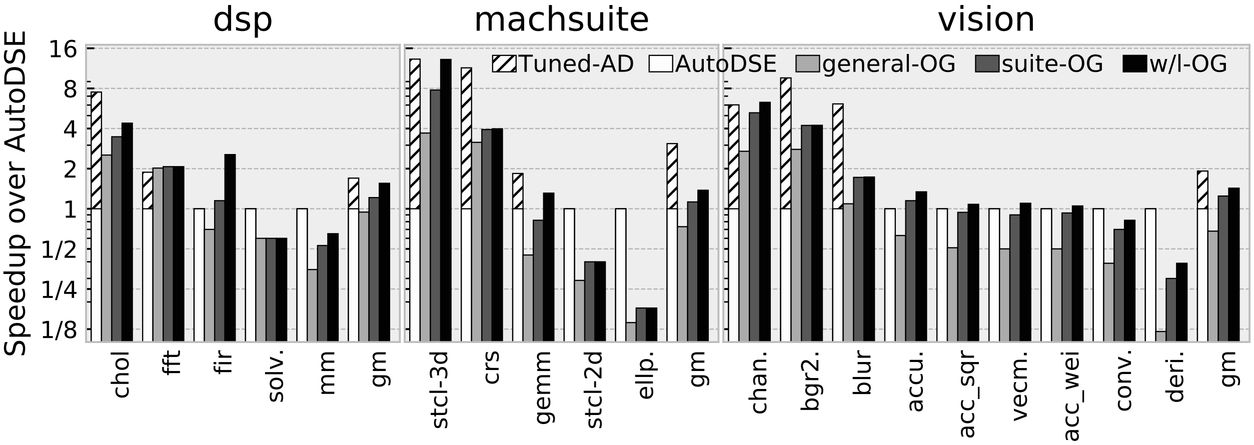


Figure 7.8: Overall Performance Comparison

7.6 Evaluation

The goal of our evaluation is to provide perspective on the opportunities of synthesized spatial overlays as compared to state-of-the-art automated HLS (AutoDSE). This section is organized around 8 key questions, with the takeaways being:

- OverGen is able to generate reconfigurable designs that can outperform baseline AutoDSE (without kernel tuning) by mean 1.2×, even though the generated designs are more flexible.
- HLS benefits more heavily from kernel tuning, while OverGen's execution model and compiler can handle many code patterns natively without software effort.
- New applications within the same domain can be easily deployed on an existing overlay with only modest performance degradation, due to overlay flexibility.

Q1 How performant are generated overlays?

Figure 7.8 shows the overall performance of OverGen across all workloads, normalized to AutoDSE without kernel tuning. We demonstrate three different kinds of overlays:

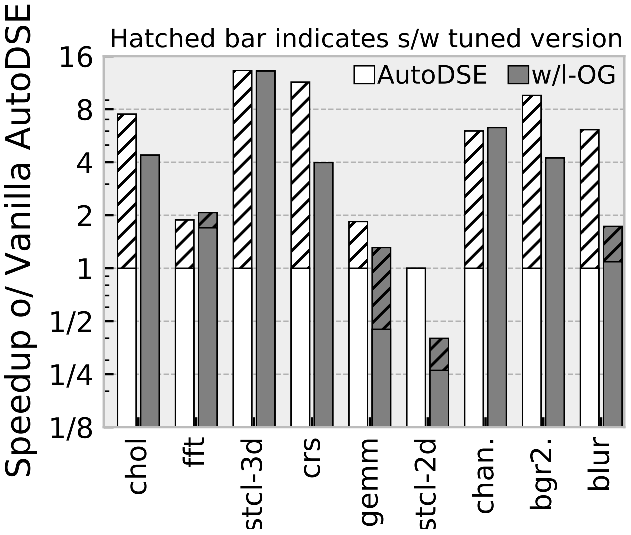


Figure 7.9: Effect of tuned kernels

- **General Overlay** (second bar): A single hand-designed mesh-based accelerator overlay targeting all workloads with maximum vectorization width (512 bit).
- **Suite Overlay** (third bar): An overlay specialized to each workload suite. Table 7.3 shows the specs of each.
- **Workload Overlay** (fourth bar): An overlay specialized only to a single workload.

We first compare against AutoDSE without manual kernel tuning. The general overlay achieves comparable performance to AutoDSE on the DSP suite and MachSuite, and mean 68% of the performance on vision suite. This is because it can only fit at most 4 general tiles, due to the high overhead of the general overlay's datapath and FUs (about 52% in LUT). The per-suite specialized overlays outperform baseline AutoDSE by a mean 1.2×, primarily due to having 2-3× more tiles (i.e. due to specialized network, FUs, and memories). Additionally, the DSP overlay uses two scratchpads to increase bandwidth without requiring more expensive wider accelerator datapaths. The per-workload specialized designs can outperform AutoDSE without kernel tuning by mean 1.45× for similar reasons; the relative improvement over suite-specialized is modest, especially for Vitis, due to the strong

	Spec.	Mach.	Vitis	DSP	General
System	Tile Count	10	13	7	4
	L2 #Bank	16	16	8	4
	NoC B/W (Byte)	64	64	64	32
Accelerator	PEs	20	16	10	24
	Switches	17	11	27	35
	Avg. Radix	2.9	2.61	2.85	4.69
	Int $+/\times/\div$	16/14/0	16/15/13	0/0/0	24/24/24
	Flt. $+/\times/\div/\sqrt{x}$	4/4/0/0	0/0/0/0	6/6/5/2	24/24/24/24
	Spad. Cap. (KB)	64	-	8, 32	32
	Spad. B/W (B/cyc)	32	-	32, 32	32
	Spad. Indirect?	Yes	-	No, No	Yes
	GEN/REC/REG	0/0/0	0/0/0	0/1/0	1/1/1
	In Ports B/W (B)	160	112	152	224
	Out Ports B/W (B)	96	48	104	160

Table 7.3: Specification of Suite Specific Overlays

similarity between workloads.

Compared to AutoDSE with manual kernel tuning, OverGen is able to achieve 0.71×, 0.37×, 0.65× of performance for DSP, Machsuite and Vision respectively, while still maintaining workload-flexibility. This is sensible, as hardware structures for preserving generality and programmability reduce the maximum resource efficiency; Q2 goes into depth on why kernel tuning is more critical for AutoDSE.

While most suite overlays were at least half the performance of the AutoDSE designs, there were a few outliers. Both `stencil-2d` and `derivative` both apply aggressive reuse optimization through a sliding window, which can be well specialized by line buffer architecture on HLS [211]. For `ellpack`, we have to load a vector to the scratchpads of all cores, but we currently lack broadcast support from DRAM to scratchpad, which wastes significant bandwidth; incorporating stream-based multicast [226] would be helpful.

Q2 Impact of kernel tuning across frameworks?

We studied 9 workloads that benefit from kernel tuning, as shown in Figure 7.9. There are 7 workloads where AutoDSE (and its underlying HLS technology) does not handle some code patterns well, leading to lower performance because of increased initiation interval (II: number of cycles between pipeline compute instances). In general, these patterns are more easily supported on OverGen's ISA/compiler. To substantiate this, we manually transform these 7 workloads to improve their II for AutoDSE, and we found 4 opportunities for kernel tuning in OverGen.

AutoDSE Kernel Tuning: We find that two main manual transformations are useful in these workloads: eliminating variable loop trip counts, and strength reduction for strided access patterns. Table 7.4 shows the II's before and after these transformations, and the hatched bar in Figure 7.8 and Figure 7.9 shows the tuned workloads' performance. Note that all other workloads achieve II=1, and OverGen *always* achieves II=1. We next discuss each transformation and the affected workloads.

Causes	Var. Loop TC			Inefficient Strided Access			
Workload	chol.	crs	fft	bgr2.	blur	chan.	stcl-3d
Untuned II	10	4	2	9	6	8	6
Tuned II	5	2	1	1	1	1	1

Table 7.4: HLS Initiation Interval (II) Optimization

Variable Loop Trip Count: HLS prefers a perfect loop nest with fixed trip-count [211], but `cholesky`, `fft`, and `crs` all have variable trip counts or imperfect loop bodies. To transform these programs, we replace variable trip counts with a fixed maximum, and push outer-loop computation into the inner loop. We then guard the conditional execution with if-statements within the inner loop. OverGen supports variable trip-count streams natively (using REVEL's ISA [276]).

Inefficient Strided Access: AutoDSE's toolchain has trouble efficiently performing strided memory access with small strides (including accesses that appear strided when observing only the innermost dimension of the access pattern). Such patterns can limit AutoDSE's ability to exploit memory parallelism, either at the BRAM level with multiple ports, or at the DRAM level with memory request coalescing. To help the underlying HLS tools understand the access pattern better, the solution is to perform a strength reduction on any strided accesses using the innermost induction variable (e.g. instead of using `i * 4`, increment `i` by `4` in each iteration). OverGen's compiler natively supports strided streams and coalescing adjacent streams.

Prebuilt Database: AutoDSE has a pre-built database that records the best explorer configuration of AutoDSE for common workloads. `gemm` is optimized using this database.

OverGen Kernel Tuning: These software behaviors of interest are more easily captured by the OverGen compiler, so only 4 workloads benefit from source code transformation on OverGen. For `fft`, we peel the last several iterations, so that strided scalar access can be coalesced to fully utilize the memory bandwidth [273, 275]. For `gemm`, to minimize I/O traffic into the accelerator and improve reuse, we unroll across two inner-loop dimensions (similar to tensorization [274]). For `stencil-2d` and `blur`, our compiler has limited support for exploiting reuse from overlapped data access between subsequent iterations. Therefore, we manually unrolled the iterations to reuse the overlapped data.

Overall, while kernel tuning is a helpful avenue for performance improvement in AutoDSE's HLS-based approach, it also more often requires programmer effort to get competitive performance than OverGen for this set of workloads.

Q3 How fast is OverGen's DSE?

Figure 7.10 shows the DSE and synthesis time comparison between AutoDSE (first bars in each suite) and the suite-wise OverGen overlay (right-most hatched bar). Comparing AutoDSE's combined time of synthesizing each application, our DSE constructs a more general accelerator while using only 47% of the time.

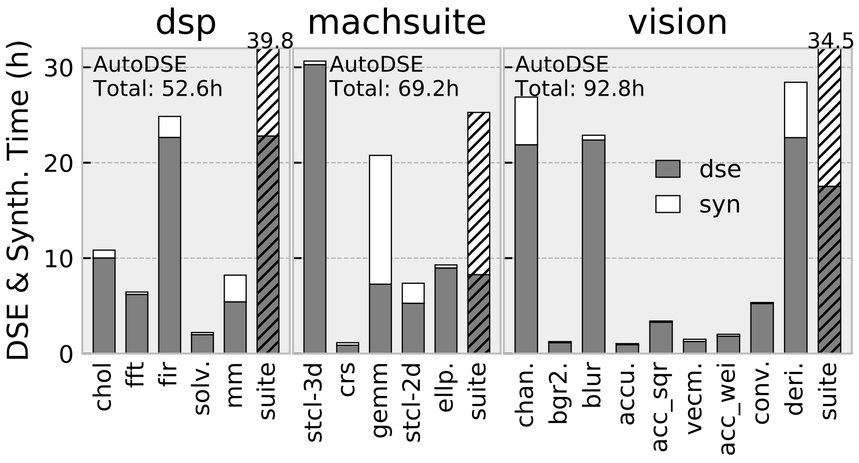


Figure 7.10: DSE and synthesis time comparison.

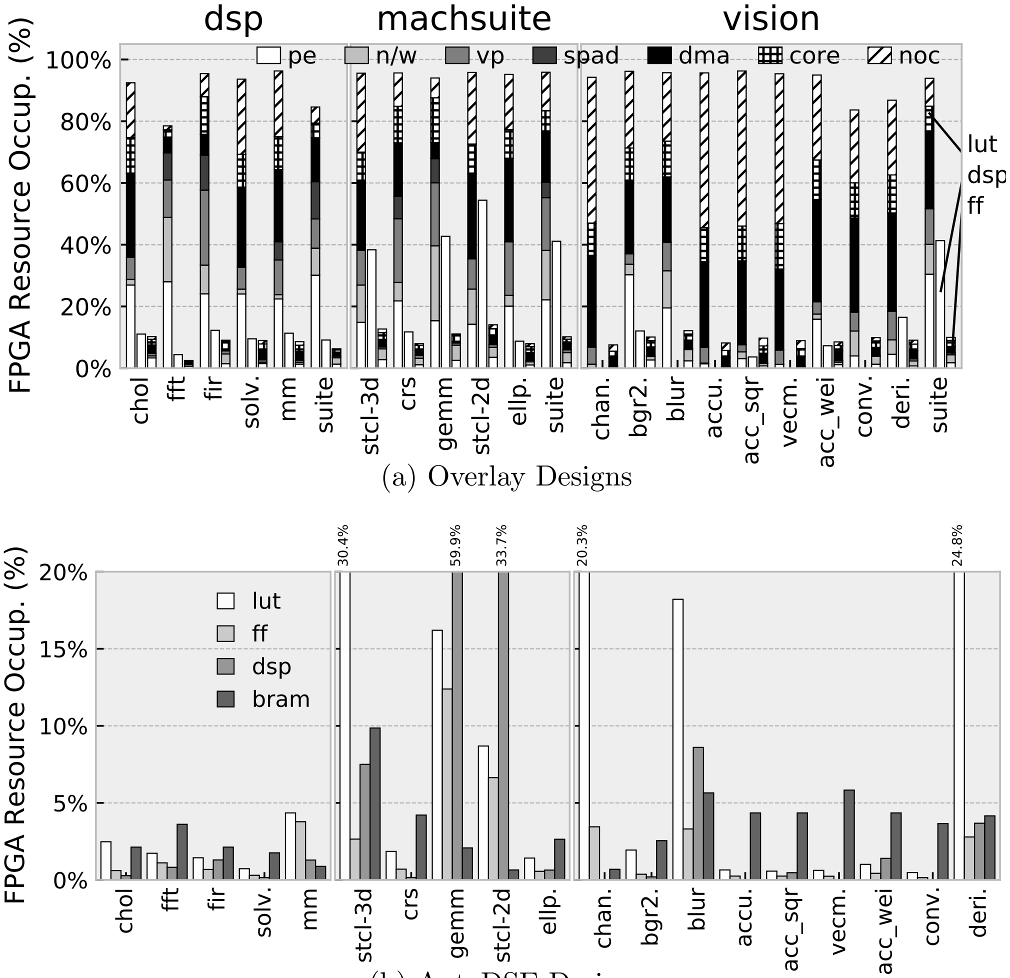
Q4 What are the limiting FPGA resources?

Figure 7.11 shows the resource breakdown of each component, normalized by the total FPGA resources available for both overlay and AutoDSE designs (with kernel tuning). All the generated overlay designs (both per-workload and suite) consume from 81% to 97% of LUTs, which is the limiting factor. Because OverGen favors some potential generality for future workloads, the DSE greedily consumes as many resources as possible, even if there is no parallelism or when we are memory bandwidth bound. One of the biggest components in terms of LUTs is the NoC, due to its crossbar-based implementation (prior work observed similar overheads [69]). AutoDSE tends to consume fewer resources as it favors utilizing less hardware when memory bound or parallelism bound, as generality is not a goal.

Q5 Can additional workloads be mapped to an overlay?

We perform a "leave-one-out" experiment to study the overlay flexibility. Specifically, we generate an overlay for all but one workload in a suite, then try to map the remaining workload. If that workload can map with relatively high performance, that indicates a more robust design.

The results are shown for MachSuite in Figure 7.12. Most of the workloads can be mapped



(a) Overlay Designs

(b) AutoDSE Design

Figure 7.11: FPGA Resource Breakdown

to the corresponding leave-one-out accelerator, with mean 49.5% performance degradation. Performance loss is caused by datapath specialization, which prevents the optimal spatial mapping; generally, a less-vectorized version is used, which has commensurately less performance. The modest performance loss may be acceptable to an FPGA programmer making incremental changes. We imagine that the compiler could inform the user when a significant performance improvement is expected, to signal when to perform DSE again.

We use the same setup to evaluate the compile/reconfiguration time, as compilation time is most meaningful on an overlay that was not specifically designed for that workload. Comparing against AutoDSE-based HLS, our spatial overlay compilation is 10000× faster. Also,

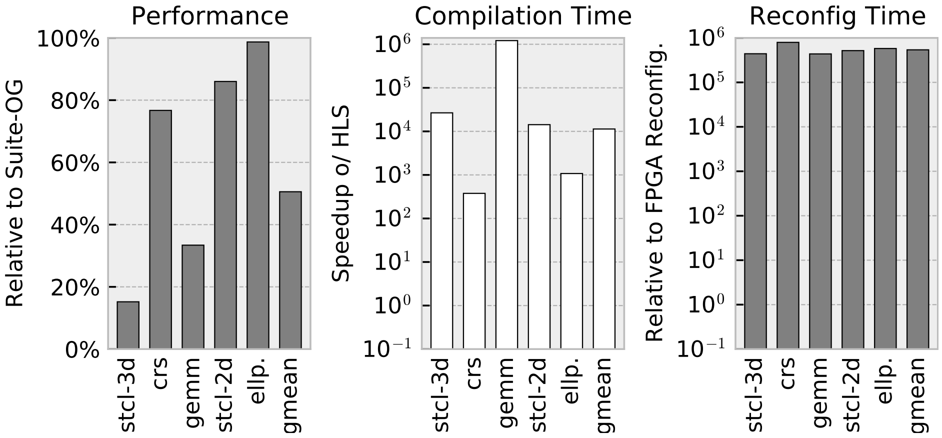


Figure 7.12: "Leave-one-out" Flexibility Evaluation

reconfiguration is much faster by mean 54000×. This is useful if the desired FPGA functionality changes rapidly, enabling efficient temporal multiplexing at very fine time scales.

Q6 How does overlay-generality affect performance?

OverGen can be used to generate increasingly general designs by incrementally adding more target workloads. Figure 7.13 shows the results of such an experiment, where we incrementally add workloads and rerun the DSE to analyze how the number of tiles and resource usage changes. We witness the overall datapath (PE + Port + network) use per tile increases as new workloads are added to the target set, because the datapath becomes more general. To compensate, the number of tiles decreases from 15 to 10. Because some of the workloads are memory bound, it only costs mean 8% performance to support all workloads in this suite.

Q7 How do more DRAM channels affect performance?

Figure 7.14 shows the performance with varying DRAM channel count, normalized to single-channel DRAM for each design. For AutoDSE, most MachSuite kernels can benefit from multiple DRAMs by mean 25%. Element-wise memory-intensive workloads like `mm`,

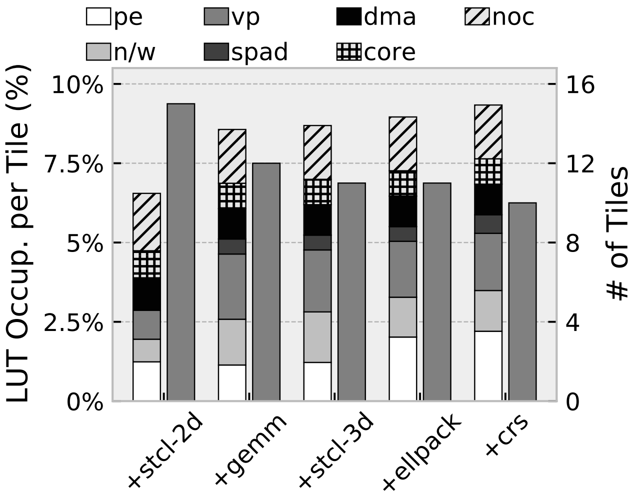


Figure 7.13: Incremental Design Optimization.

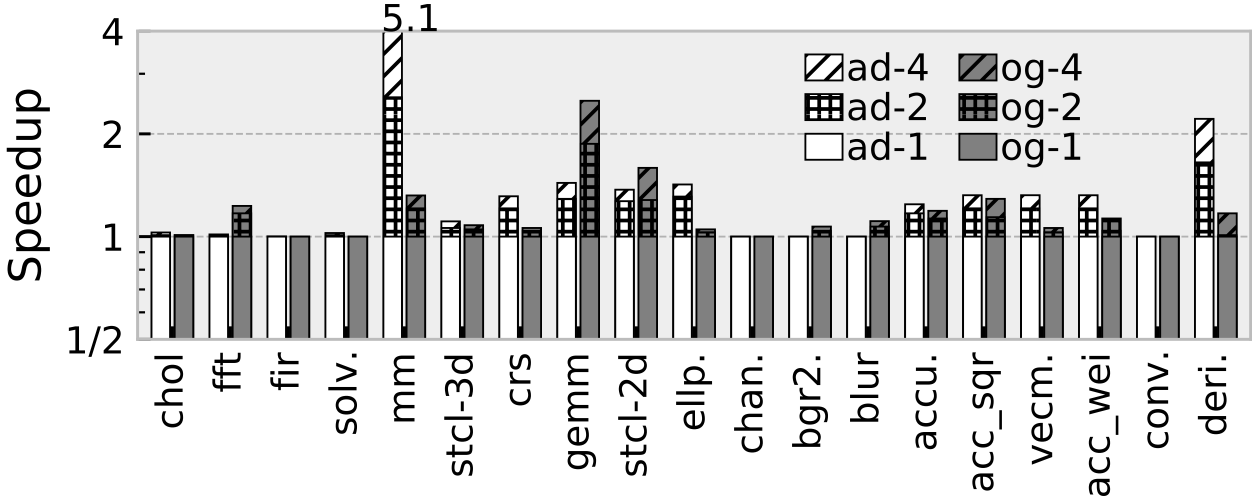


Figure 7.14: Effects of DRAM channels

gemm[1], vecm., accu., acc_sqr, acc_wei and deri. can also benefit from multiple DRAM channels. The OverGen Workload Overlays see benefits on a similar set of workloads by mean 19%.

Q8 Do schedule-preserving transforms improve DSE?

Figure 7.15 compares the DSE algorithm with and without schedule-preserving transformations. Here the x-axis is time in hours, and the y-axis is the DSE's estimated IPC for the

[1] gemm is a tiled (blocked) implementation of matrix multiply, mm is not

whole FPGA. Schedule-preserving transformations help the DSE converge faster to designs that are more-specialized to the workload datapath topologies. Overall, DSE time is reduced by mean 15%, and the estimated IPC is improved by 1.09× (running on the FPGA confirms 1.08x speedup).

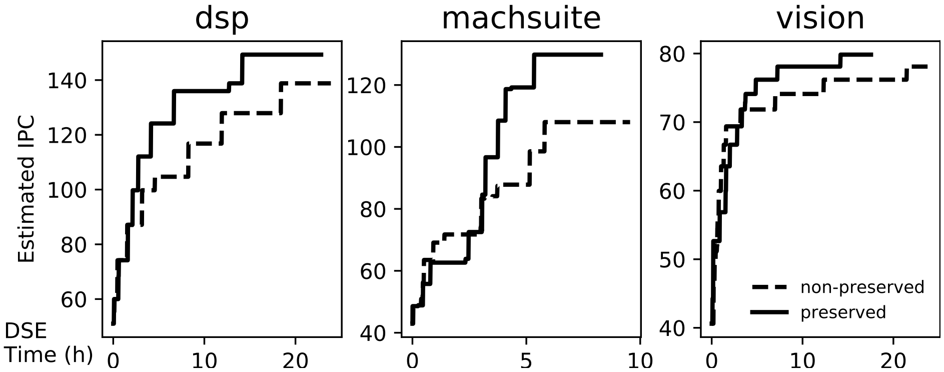


Figure 7.15: The effects of schedule-preserving transforms.

7.7 Related Works

Overlay Architectures We highlight significant and recent overlay approaches; Li et al. provide an in-depth survey [162], and a recent ACM community article by Yuze et al. also in detail discussed how FPGAs target DSAs [61].

Soft CPU: FPGA vendors provide soft processor implementations, e.g. Xilinx MicroBlaze [4] and Intel Nios II [5], and there are also many open source works, e.g. SPREE [284], iDEA [48, 47], and OpenRISC [198]. Some alternatives provide higher-performance microarchitectures, such as multi-issue (e.g. Leon3 [95], FPGA-Nehalem [227]), multi-thread (e.g. Octavo [156], CUSTARD [80], MT-MB [182]), multicore with scalable networks (e.g. Heracles [143], Kumar et al. [153]), vector operations (e.g. SIMD-Octavo [155], MXP [230]), as well as VLIW (e.g. TILT [215]).

Soft GPGPU: FlexGrip [24] and MIAOW [31] are single compute-unit (CU) overlays based on Nvidia and AMD GPU architectures, respectively. FGPU [21] was able to synthesize multiple CUs on a single FPGA board, with a follow-up work specialized for persistent deep learning [168].

Reconfigurable Architectures: QUKU [233] is an early example of a 2D-mesh style CGRA overlay. reMORPH is another 2D mesh-based overlay that is built around the FPGA's DSP blocks as primitives [204]. VDR [46] is a CGRA overlay which can map short program traces for JIT-based compilation. The DySER heterogeneous core/CGRA architecture was also mapped to FPGA [37, 118]. ZUMA is an example of an FPGA-on-FPGA overlay [42].

Customizable Overlays: Interestingly, some overlays allow architecture customization. For example, CREMA [97, 122] and Quickdough [164, 126] leverage templates to customize PEs for each application and speedup the design process. CGRA-ME [62, 191, 63, 266] and AHA [150, 174, 30] further introduced architecture description languages for arbitrary topologies and DSE with CGRA mapper involvement. Mocarabe [251] introduces the communication cost as a first-class citizen in the compiler to obtain a design with high frequency while still meeting the targeted II. SCRATCH [83] is a GPU-based overlay based on MIAOW [31], which automatically identifies the application-specific demands regarding the instruction set and computing unit capability, and generates a trimmed down GPU design.

Key Difference to prior Overlays As compared to these prior frameworks, our overlay-synthesis approach attempts to perform application specialization automatically and across many aspects of the overlay architecture (instructions/topology/execution model/provisioning).

FPGA Programming While the overlay approach improves the programmability by providing another layer of abstraction, there are also efforts to directly tackle this problem with new programming languages with lower-level abstractions. As an example, Dahila [190] generates predictable HLS designs by incorporating time-sensitive affine types into the language. On the other hand, Reticle [257] proposes an intermediate representation and low-level assembly that explicitly expresses special resources on FPGAs, e.g. LUTs and DSPs. Spatial [146] is a language designed for implementing accelerators based on parallel patterns. Although these techniques improve the programmability, they do not tackle reconfiguration overheads. Just-in-time compilation frameworks can also reduce the burden of FPGA synthesis [228, 35].

A recent approach integrates separate compilation into an FPGA design flow to enable better usability [202, 279]. These works leverage faster compilation/reconfiguration to subregions of the FPGA, and enable linking through a packet-switched network. The RapidStream framework [108, 107, 60] also partitions a large design for parallel implementation and final re-assembly, but instead uses customized point-to-point and pipelined channels to address the high area and limited bandwidth of packet-switched NoC's in prior work.

7.8 Conclusion

While FPGAs have proven to be extremely effective computational accelerators, their usability is not ideal. The heart of the problem is the limited design space of existing HLS tools, which is inflexible and requires frequent re-synthesis. In this work, we develop and evaluate the idea of an alternate HLS paradigm where a highly-flexible overlay is the target architecture. Surprisingly, even though the generated designs are programmable, the overall performance is on-par with state-of-the-art HLS tools.

Yet there is much more to be explored, and OverGen should be seen as a proof-of-concept for the potential of multicore spatial overlays. Many aspects of the design space can be further specialized to the chosen applications, leveraging the extreme flexibility of FPGAs. Examples include the NoC topology [297], NoC protocol [67, 285], cache policies [138], coherence protocol [302], and synchronization [98] to name a few. One broad, underexplored aspect is *heterogeneity*: including heterogeneous cores, caches, networks, and memories. While our current framework assumes pure single-program parallelization, real systems (e.g. mobile SoCs [116], datacenters, VR [123] and even brain computer interfaces [135]) often require heterogeneous mixes of workloads with different throughput and latency requirements on the same fabric — this opens up vast potential for these different forms of architecture and microarchitecture heterogeneity. Supporting heterogeneity is challenging both because it adds another dimension to design-space exploration, and because it requires novel system support in virtualization and runtime management of heterogeneous resources.

Overall, we see spatial overlay synthesis as a potentially disruptive approach for FPGA HLS, and OverGen as springboard for future spatial architecture research.

CHAPTER 8

UNIT: Unifying Tensorized Instruction Compilation

In this chapter, an extensible framework embedded in a tensor domain-specific language that compiles tensorized instructions will be presented. In reaction to the increasing computing demand on the tensor operations, general-purpose hardware vendors (e.g. Intel, ARM, and NVIDIA) have extended their instruction sets to develop specialized instructions for these operations. These instructions specialize new program behavior that can hardly be captured by prior compilation techniques. Though we claimed to have a general-purpose compiler for specialized accelerators, there is still a gap between generality and high specialization. By taking advantage of the domain knowledge provided by a tensor domain-specific language, a productive and extensible compilation flow for these instructions is developed.

8.1 Motivation

Dense tensor operations like matrix multiplication (Matmul) and convolution (Conv) have long been the workhorses in many domains, including deep learning workloads [79]. The popularity of deep learning means that aggressively optimizing these operations has a high payoff. Essentially, Matmul and Conv are a series of multiply-accumulate (MAC) operations, which perform accumulation over a number of elementwise multiplications.

To capture the reduction behavior and perform it more efficiently, recent general-purpose processors offer native tensor operation specialized instructions (hereinafter referred to as *tensorized instructions*), like Intel VNNI [12], Nvidia Tensor Core [15], and ARM DOT [11].

Unlike the conventional SIMD instructions, after performing elementwise arithmetic operations, these instructions introduce a "horizontal computation" to accumulate elementwise results. Further, tensorized instructions are often mixed-precision, meaning that elementwise operations use less precise and lower bitwidth operands (e.g., `fp16` and `int8`), while accumulation occurs with higher bitwidth, where it is needed. This offers a good balance between data width and precision that is generally sufficient for deep learning workloads [176, 125], and enables the use of quantized data types.

Mixed-precision is difficult to express in a single SIMD instruction, because the output vector width is different than the input vector width. In most ISAs this paradigm requires multiple SIMD instructions to express. In a tensorized instruction, by definition there are fewer outputs, so allocating more bitwidth to them for the output vector is natural. In addition, tensorized instructions sometimes reuse the same inputs multiple times, which reduces the required register file bandwidth. Overall, tensorized instructions offer significant advantages over SIMD for executing MACs.

While promising, the absence of appropriate compilation techniques limit c the applicability of these tensorized instructions. Conventional SIMD instructions are vector instructions, so industry standard compilers only try parallelizing the innermost loops. In addition, it is difficult for the high-level language programmer to express the compute flow in a tensorization-friendly way and hint the compiler to try tensorization upon a loop nest, because the dependency of reduction is more complicated and error-prone.

In practice, there are normally two options to leverage tensorized instructions. One way is to call the vendor-provided libraries such as Intel oneDNN [16], Nvidia cuBLAS and cuDNN [14], which provides highly optimized performance in some pre-defined single kernels using tensorized instructions [112, 281]. However, it also brings inflexibility when it comes to new workloads or when further performance exploitation is desired. The other option is to manually write assembly intrinsics, which sets a high bar to ordinary developers and hence lacks productivity. Some prior works tried to solve this problem by developing a compiler [240, 249] for each instruction. This requires too much effort when there are many tensorized instructions, both within and across hardware platforms.

Our Goal: Although different processors may provide different tensorized instructions, in the context of deep learning workloads, we observe that these instructions essentially handle a similar compute pattern, i.e., elementwise multiplication and then horizontal accumulation. They primarily differ in the number of elementwise computation lanes and the accepting data types. Therefore, we aim to develop a unified approach to compile these tensorized instructions on multiple platforms to optimize the tensor operations in deep learning workloads. Our techniques are extensible to the tensorized instructions with other data types and operations as well.

Challenges: There are several challenges to attain a unified compilation pipeline:

- *Instructions Integration:* Instead of building a new specialized compiler for each new instruction, it is desirable to create a unified and extensible compilation flow;
- *Detecting the applicability:* Given a tensorized instruction, a first question is whether and how this instruction can be applied to the target tensor operation, which may require loop reorganization to make it applicable;
- *Code rewriting:* When applicable, the compiler must determine how the loops involved should be rewritten by the tensorized instruction, and how the loops should be rearranged to achieve high performance.

Our Insight: We envision that the key to addressing these three challenges is to have a unified semantics abstraction for tensorized instructions so that the analysis and transformation can also be unified.

This paper presents UNIT, an end-to-end compilation pipeline to surmount the above three challenges. UNIT takes the tensorized instructions (e.g., Intel VNNI instructions on CPUs, or Nvidia Tensor Core instructions on GPUs) and a deep learning model as input, lowers the tensor operations of the model into loop-based IRs to identify the tensorizable components, and inserts the tensorized instructions by transforming and rewriting the loop. It achieves high performance for tensor operations, and consequently, model inference. To the best of our knowledge, this is the first work to tackle tensorized instruction compilation and optimization with a unified solution. UNIT not only achieves high performance for single tensor operations, but also provides desirable model inference latency in practice.

Key Results: According to our evaluation, UNIT is expressive enough to target many tensorized instructions on multiple hardware platforms, including Intel VNNI, Nvidia Tensor Core, and ARM DOT. The generated programs for end-to-end model inference are 1.3× and 1.75× faster than the solutions backed up by Intel oneDNN and Nvidia cuDNN on CPU and GPU, respectively. In addition, UNIT can be extended to new tensorized instructions with moderate effort. Although we designed UNIT to target Intel CPUs and Nvidia GPUs, on an ARM Cortex A-72 CPU with DOT instructions, UNIT achieves up to 1.13× speedup against a carefully manual tuned solution.

8.2 Background

UNIT is an end-to-end compilation pipeline capable of automatically mapping tensorized instructions to the deep learning tensor operations. It defines the tensorized instruction's semantics using a suitable intermediate representation (IR) and inserts them in proper places of the program of tensor operations. In this section, we give an overview of popular mixed precision tensorized instructions, followed by the limitations of existing solutions in automatic mapping of these tensorized instructions. Finally, we discuss the background of tensor domain specific language and the multi-level intermediate representation.

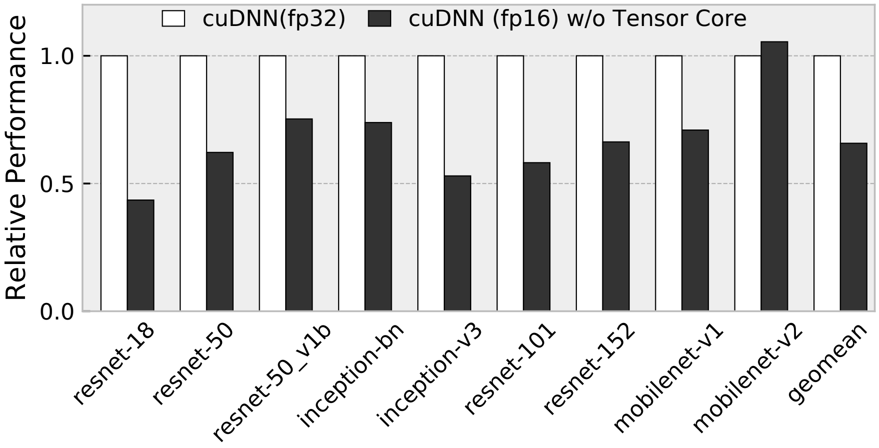


Figure 8.1: Performance comparison on Nvidia V100-SXM2 between `fp32` and `fp16` without mixed precision instruction support.

8.2.1 Mixed Precision Tensorized Instructions

Deep learning is computationally expensive, requiring substantial compute and memory resources. As deep learning becomes more pervasive, researchers are designing both software and hardware techniques to reduce the compute and memory burden. A widely adopted approach in this context is using mixed precision for expensive operations, e.g., convolution or dense operations [176, 125]. In practice, this means representing 32-bit floating point (`fp32`) operands with a lower bitwidth datatype - 16-bit floating point numbers (`fp16`) or 8/16-bit integer numbers (`int8`, `int16`). To keep the accuracy in check, it is helpful to accumulate the results in higher precision (`fp32` or `int32`). This type of mixed precision computation is often called *quantization* for integer values [125]. In this paper, we will always use *mixed precision* for brevity.

While using mixed precision data types reduces memory footprint, it might not necessarily lead to performance improvement. To investigate this, we conducted an experiment to compare the performance of Nvidia cuDNN performance for `fp16` and `fp32` in the absence of Nvidia mixed precision tensorized instructions (Tensor Core). As shown in Figure 8.1, we observe that blindly using mixed precision leads to substantial slowdown because of the

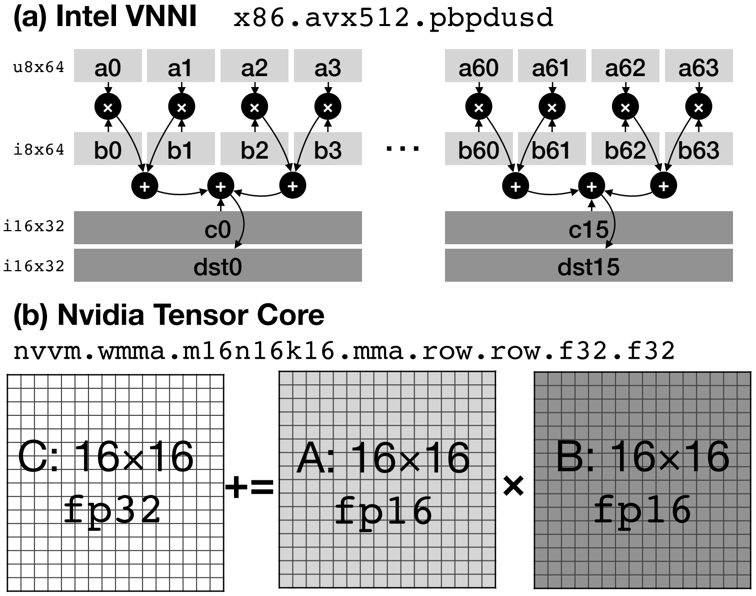


Figure 8.2: The semantics of Intel VNNI and Nvidia Tensor Core. The text beside is the name of the corresponding LLVM intrinsic.

overhead of casting between two data types.

Therefore, mainstream hardware vendors (Intel, ARM and Nvidia) have introduced mixed precision tensorized instructions to achieve better performance. These instructions add mixed precision arithmetic support where operands are of lower precision while the accumulation happens in higher precision, potentially leading to 2× - 4× speedup. The most popular examples of these tensorized instructions are Intel VNNI, ARM DOT and Nvidia Tensor Core. We will discuss the semantics of these operations in Section 8.3.

Hardware vendors have a long history of adding new instructions to accelerate important applications. However, the mixed precision tensorized instructions introduce a unique idiom - horizontal accumulation. These tensorized instructions typically conduct a sequence of elementwise multiplications governed by a memory access pattern, followed by a horizontal accumulation. The accumulation is termed horizontal because all values to be accumulated are present in the same vector register. For example, as it is shown in Figure 8.2(a), Intel VNNI executes a dot product of two vectors, each having 4 `int8` elements, while performing the accumulation in `int32`. We observe a similar pattern, though with different numbers of entries and data types, for Nvidia Tensor Core (in Figure 8.2(b)) and ARM DOT instructions (this is omitted, because it is similar to VNNI).

8.2.2 Limitations of Existing Solutions

Though tensorized instructions seem promising, their adoption pace is limited because of the absence of an automatic technique that can detect and use these instructions seamlessly. Currently, their usage in the deep learning domain is limited to hardware vendor libraries like Intel oneDNN and Nvidia cuDNN, which may provide high performance for the pre-defined operations but are inflexible as discussed in Section 8.1.

Similarly, conventional loop vectorizers find it hard to exploit the profitability of these tensorized instructions, as they are not designed to work with the horizontal reduction idiom. Conventional loop vectorizers in general-purpose compilers like GCC and LLVM mainly focus on either analyzing the innermost loop body or combining instructions in the unrolled loop bodies. When it comes to the horizontal reduction idiom, these compilers often reorder the computation and generate epilogue reduction, preventing us from using the tensorized instructions.

There have been some recent works in compiling programs to leverage tensorized instructions. PolyDL [249] generates CPU programs for convolution kernels in neural networks that call a GEMM micro-kernel using Intel VNNI instructions. Bhaskaracharya et al. [240] generate CUDA programs for matrix computation leveraging Nvidia Tensor Core. However, these works are limited to one platform and its specific instruction, which lacks generalizability. A generic solution to handle tensorized instructions from multiple platforms together is still missing.

8.2.3 Multi-Level Intermediate Representation

Compilers often have multiple levels of intermediate representation (IR) to express the program; each level is designed to enable different analyses and transformations. In this section, we describe the background of a tensor domain specific language (DSL) and the multi-level IR.

8.2.3.1 Graph-Level IR

Deep learning compilers like TVM [53], Glow [224], and XLA [10] adopt a graph-level IR to represent a deep learning model as a directed acyclic graph (DAG) of operations. This graph-level IR is useful for inter-tensor-operation optimization, like tensor shape padding, operation fusion, and choosing the proper data layout [167]. Our tensorized analysis relies on tensor padding so that loops can be tiled by the number of lanes of the instruction perfectly. However, this IR has little knowledge about the implementation of each tensor operation. When compiling a graph-level IR, each node of the DAG will be dispatched to its implementation in tensor DSL as explained next.

8.2.3.2 Tensor DSL

Tensor domain-specific languages, like Halide [214], TVM [53], and Tensor Comprehension [256], have been developed to productively and portably express tensor programs while enabling efficient performance tuning. As shown in Figure 8.3 and Figure 8.4, programs written in tensor DSLs follow this paradigm: Users first declare the tensors and the loop variables, and then the computation is described by expressions involving the declared tensors and loop variables. These DSLs also provide interfaces to split, reorder, and annotate loops without affecting the computation semantics for performance tuning.

All the information gathered from the tensor DSL frontend will be stored in a *tensor Op* data structure, including the declared tensors, loop variables, expressions, and loop manipulation.

8.2.3.3 Tensor IR

Each *tensor Op* is then lowered to *Tensor IR*, which is an imperative program IR with additional constraints: All the loops are canonical (starting from 0, and increased by 1 each time), and all the array operations are restricted (i.e., an element cannot be accessed by two different pointers). These two properties enable making strong assumptions for analysis and transformation. Our work conducts analysis on the *tensor Op* data structure level and then performs transformation on the tensor IR. Although the tensor IR provides essentially identical information for analysis, as discussed above, it is easier to reorganize the loops via the *tensor Op* data structure.

8.2.3.4 Low-Level IR

The tensor IR is lowered to a general-purposed low-level IR like LLVM, after all the specialized analysis and transformations on the tensor IR are done, to get ready for assembly code generation.

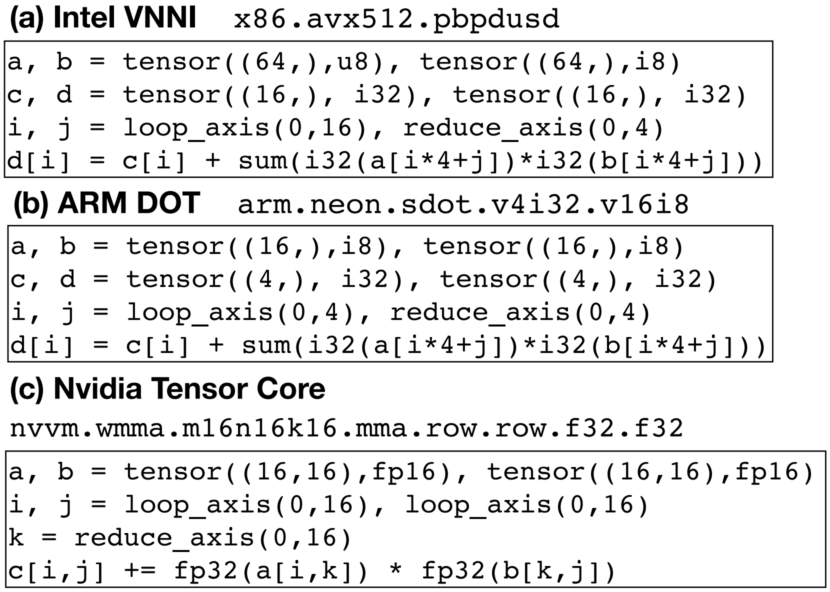
8.3 Unified Tensorization

Our goal is to automatically *tensorize*[1] mixed-precision deep learning tensor operations across a variety of hardware platforms. We resolve the challenges discussed in Section 8.1 by presenting UNIT with the following techniques:

1. *Tensorized Instruction in Tensor DSL:* To abstract the diverse tensorized instructions on different hardware platforms, we leverage the existing tensor DSL to represent their semantics.

2. *Applicability Inspection:* To determine if and how a tensorized instruction can be

[1] We coin the word to mean rewrite and optimize a given code by the tensorized instruction.



(a) Intel VNNI `x86.avx512.pbpdusd`

```
a, b = tensor((64,),u8), tensor((64,),i8)
c, d = tensor((16,), i32), tensor((16,), i32)
i, j = loop_axis(0,16), reduce_axis(0,4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```

(b) ARM DOT `arm.neon.sdot.v4i32.v16i8`

```
a, b = tensor((16,),i8), tensor((16,),i8)
c, d = tensor((4,), i32), tensor((4,), i32)
i, j = loop_axis(0,4), reduce_axis(0,4)
d[i] = c[i] + sum(i32(a[i*4+j])*i32(b[i*4+j]))
```

(c) Nvidia Tensor Core

`nvvm.wmma.m16n16k16.mma.row.row.f32.f32`

```
a, b = tensor((16,16),fp16), tensor((16,16),fp16)
i, j = loop_axis(0,16), loop_axis(0,16)
k = reduce_axis(0,16)
c[i,j] += fp32(a[i,k]) * fp32(b[k,j])
```

Figure 8.3: Tensorized instructions as abstracted in the tensor DSL.

applied to a tensor operation, we developed an analysis pass in the *Inspector* component of UNIT, which analyzes the *tensor Op* data structure of both the instruction and the operation. The result of analysis will guide the loop reorganization and instruction injection.

3. *Code Rewriter:* Once the tensorized instruction is determined applicable, the Rewriter reorganizes the loop nests in accordance with the Inspector so that the innermost loop nests resemble the tensorized instruction and are ready to be replaced. Finally, it sets up the tuning space for the remaining loop nests to exploit high performance.

These components of UNIT together enable a unified compilation flow to simplify the mapping of tensorized instructions across a variety of hardware platforms. In the rest of this section, the details of each of the above steps will be discussed.

8.3.1 Semantics Abstraction - Tensor DSL

In order to unify the compilation of tensorized instructions from different platforms and keep the system open to integrate new instructions, the first question to answer is how to have a unified description of the semantics of tensorized instructions. As explained in Section 8.2, we employ ubiquitous tensor DSL and tensor IR to solve the abstraction problem. All mixed precision tensorized instructions perform some elementwise operations for vectors, followed by a horizontal reduction. Each tensorized instruction, therefore, can be regarded as a small tensor operation program written in the tensor DSL.

Figure 8.3(a) shows how an Intel VNNI instruction is described in the tensor DSL. Three source operands of Intel VNNI are 512-bit registers. Two of them are 64 lanes of unsigned 8-bit integers (`uint8`) and signed 8-bit integers (`int8`), and the other one is 16 lanes of signed 32-bit integers (`int32`), which correspond to the tensors `a`, `b`, `c` we defined. The arithmetic behavior is defined by the loop variables and the expression of `d[i]`. Here we annotate that loop `i` is data parallel, since these 16 elements are independent from each other; loop `j` is reduction since for every independent element it sums up 4 elements along with this loop. A similar loop pattern appears in the other tensor operations shown in Figure 8.4. The description of ARM DOT, shown in Figure 8.3(b), is similar to Intel VNNI, with a different number of lanes and data types.

Nvidia Tensor Core, on the other hand, performs a 16^3 square matrix multiplication as shown in Figure 8.3(c). Comparing with (a) and (b), a key difference is that it requires the accumulator register to be the same as the addition register (note the `+=`). This is due to the data type opaqueness of the Tensor Core instruction, which prevents us from giving arbitrary initial values for the accumulators.

We describe the semantics of each tensorized instruction in tensor DSL. The deep learning compiler pipeline parses the operation into *tensor Op*, which preserves tensor information like the expression tree, the loop trip count, and the array buffers. This information is essential for the analysis and transformation passes in Inspector and Rewriter.

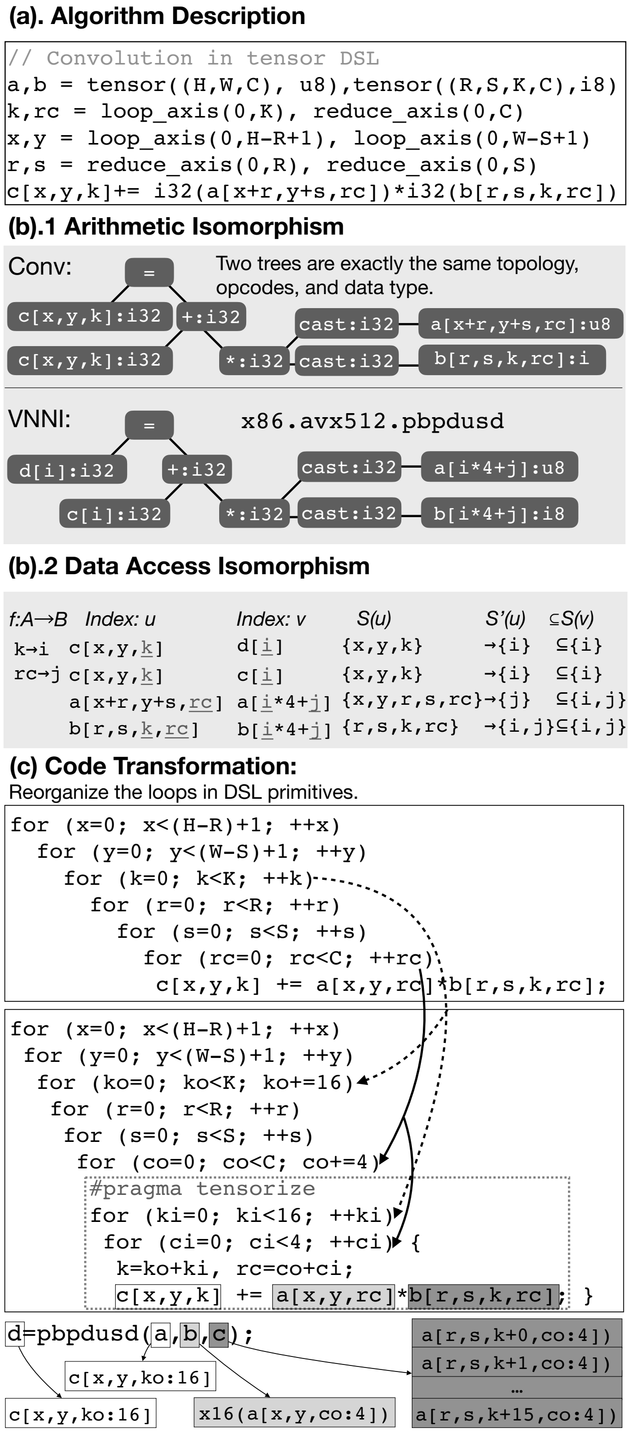
8.3.2 Applicability Detection - Inspector

To determine if a tensorized instruction can be applied to a tensor operation, the Inspector pass uses a two-step approach. It first determines if (part of) the tensor operation program and the instruction can be arithmetically equivalent by checking a form of isomorphism between their associated expression trees. After that, it inspects the data access pattern to confirm the assembly operands can be prepared so as to guide the Rewriter transformation.

8.3.2.1 Compute Isomorphism

Algorithm 1 shows the algorithm we adopt to determine the isomorphism of two expression trees. It recursively traverses both trees and matches the data type and opcode of each pair of nodes. Figure 8.4(b).1 shows that the two trees of convolution and `pbpdusd` (an Intel VNNI instruction) are in exactly the same topology and data type, so these two programs are arithmetically isomorphic.

This analysis also finds a mapping from the operands in the tensor program to the operands in the tensorized instruction. As we explained, tensor operands in the tensorized instruction are the abstraction for registers. Therefore, a register cannot correspond to multiple data sources. This property still requires further checks, which will be explained in the next section.



(a). Algorithm Description

```
// Convolution in tensor DSL
a,b = tensor((H,W,C), u8),tensor((R,S,K,C),i8)
k,rc = loop_axis(0,K), reduce_axis(0,C)
x,y = loop_axis(0,H-R+1), loop_axis(0,W-S+1)
r,s = reduce_axis(0,R), reduce_axis(0,S)
c[x,y,k]+= i32(a[x+r,y+s,rc])*i32(b[r,s,k,rc])
```

(b).1 Arithmetic Isomorphism

Two trees are exactly the same topology, opcodes, and data type.

(b).2 Data Access Isomorphism

f:A→B	*Index: u*	*Index: v*	*S(u)*	*S'(u)*	*⊆S(v)*
k→i	c[x,y,k]	d[i]	{x,y,k}	→{i}	⊆{i}
rc→j	c[x,y,k]	c[i]	{x,y,k}	→{i}	⊆{i}
	a[x+r,y+s,rc]	a[i*4+j]	{x,y,r,s,rc}	→{j}	⊆{i,j}
	b[r,s,k,rc]	b[i*4+j]	{r,s,k,rc}	→{i,j}	⊆{i,j}

(c) Code Transformation:

Reorganize the loops in DSL primitives.

```
for (x=0; x<(H-R)+1; ++x)
  for (y=0; y<(W-S)+1; ++y)
    for (k=0; k<K; ++k)
      for (r=0; r<R; ++r)
        for (s=0; s<S; ++s)
          for (rc=0; rc<C; ++rc)
           c[x,y,k] += a[x,y,rc]*b[r,s,k,rc];
```

```
for (x=0; x<(H-R)+1; ++x)
 for (y=0; y<(W-S)+1; ++y)
  for (ko=0; ko<K; ko+=16)
   for (r=0; r<R; ++r)
    for (s=0; s<S; ++s)
     for (co=0; co<C; co+=4)
      #pragma tensorize
      for (ki=0; ki<16; ++ki)
       for (ci=0; ci<4; ++ci) {
        k=ko+ki, rc=co+ci;
        c[x,y,k] += a[x,y,rc]*b[r,s,k,rc]; }
```

```
d=pbpdusd(a,b,c);
```

Figure 8.4: An example of applying Intel VNNI to Conv using UNIT.

```
function INSPECT(a,b)
    if a.type=b.type then
        if isleaf(a)$\wedge$isleaf(b) then
            if a is not bound then
                bind[a]:=b
            else if bind[a]$\neq$b then
                return False
            end if
            return True
        else if isarith(a), isarith(b) then
            cond:=a.opcode=b.opcode
            cond:=cond$\wedge$Inspect(a.lhs, b.lhs)
            cond:=cond$\wedge$Inspect(a.rhs, b.rhs)
            return cond
        end if
    end if
    return False
end function
```

Algorithm 1: Determine the isomorphism between expression trees. *a* is for the instruction, and *b* is for the operation.

8.3.2.2 Array Access Isomorphism

Once compute isomorphism is determined, the next concern is how the data are fed to this instruction. The enforcement explained in the last subsection already determines each register operand only corresponds to one array in the tensor operation. On top of this, we need to determine each element in the operand tensor corresponds to only one memory address in the tensor program when mapping to the tensorized instruction. To map a tensor program to a tensorized instruction, we need to know which loop levels are tensorized. We enumerate the loop levels to be tensorized, and these loop levels will be mapped to loops in the tensorized instruction. Note that only loops with the same annotation (data parallel or reduction) can be mapped to each other. Then we check if this enumerated mapping is feasible, by scanning each pair of operand correspondence determined in the last paragraph. If the operand in the tensor program is a constant, we just skip it[2]. If the operand is a memory operation, we inspect the index expressions of both memory operations in the operation and instruction. We define:

- A is the set of loop variables to be mapped to the tensorized instruction.
- B is the set of loop variables of the tensorized instruction.
- $f : A \mapsto B$ is the mapping we enumerate.
- $S(u) := \{x | x \text{ is loop variable in the index expression } u\}$
- $S'(u) := \{f(x) | x \in S(u) \cap A\}$

A mapping is considered feasible, if every pair of memory operation's index expressions (u, v), where u is from the operation and v is from the instruction, holds $S'(u) \subseteq S(v)$. Figure 8.4(b).2 shows an example of inspection. If $S'(u)$ is a subset of $S(v)$, this means the data loaded by the tensor operation should be broadcast along with the loop variables that do not exist in $S(v)$ to fill all the register lanes. If not, this means each register lane corresponds to multiple memory addresses under this mapping, which is not realistic for code generation, so we should try another enumeration.

[2] If it is a constant, the correspondence was already checked in the last section. This register corresponds to this constant.

If there are multiple feasible mappings, we leave this as a dimension of code tuning space. Once this mapping is determined, it will guide the further loop transformation and code generation.

8.3.3 Code Transformation - Rewriter

There are three phases in the code transformation: loop reorganization, tensorized instruction replacement, and tuning.

8.3.3.1 Loop Reorganization

As discussed in Subsection 8.3.2, the inspector selects the loop levels to be executed by the given instruction. To get poised for code generation, as shown in Figure 8.4(c), we need to tile these loops and reorder them to the innermost loop levels so that those innermost loops perform exactly the same semantics as the instruction. As we explained, tensor DSL provides the capability to reorganize the loops nests easily.

8.3.3.2 Tensorized Instruction Replacement

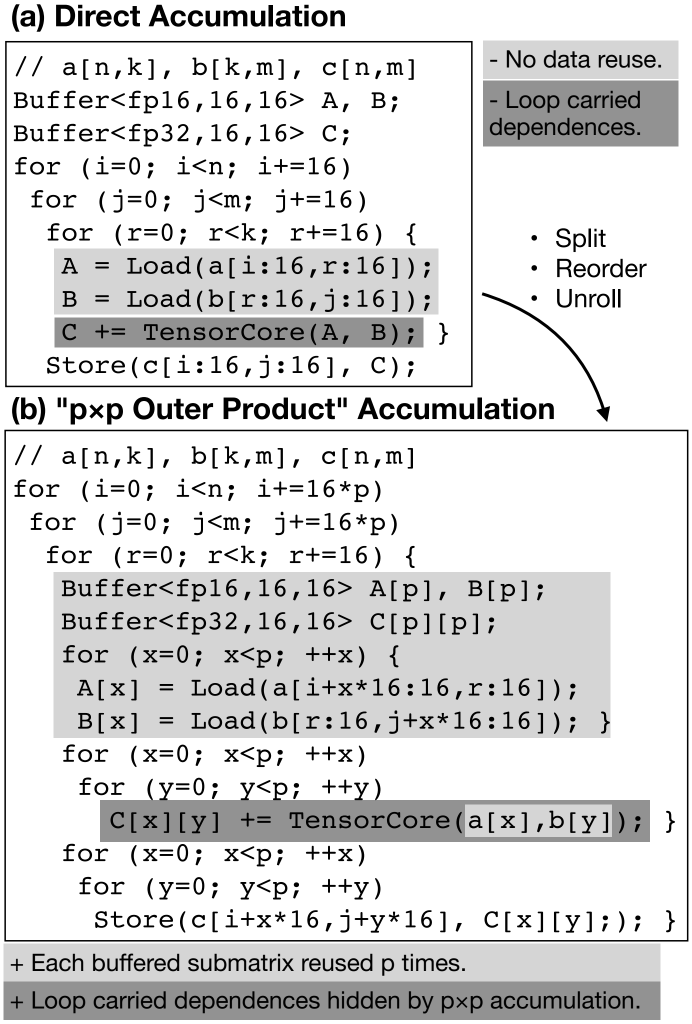
After identifying the code region to be replaced by a tensorized instruction, the code generator should prepare each operand of this instruction. It is difficult to fully automate the operand preparation for different platforms because of their diverse execution models and assembly formats. Therefore, we formalize a unified programming interface to compiler developers to manually specify the rule of operand generation. In this interface, each loop variable to be replaced, and their coefficients in the index expression are exposed. For example, as shown in Figure 8.4(c), by analyzing the strides and trip count of `ki`, and `ci`, the array access `c[x,y,c]` will be transformed to a 16-lane vector; `a[x,y,rc]` will be vectorized along with `c` by 4, and broadcast along with `ki` by 16; `b[r,s,k,c]` will be vectorized along with `ci` by 4, and unrolled and concatenated along with `ki`.

8.3.3.3 Tuner

All the other loop levels that are not involved in instruction rewriting can be reorganized to tune the performance. Here, we develop strategies to optimize the performance of tensor programs on both CPU and GPU. The generic philosophy is to exploit both fine- and coarse-grained parallelism. We also developed specialized strategies because of the different execution models and memory hierarchy.

CPU Tuning: On CPU, data-parallel loops are distributed to multiple threads to achieve coarse-grained parallelism. On the other hand, the loop-carried dependence in reduction loops introduces RAW hazards in the execution pipeline. To avoid this penalty, and achieve instruction-level parallelism, we reorder and unroll a small degree of data parallel loops below the innermost reduction loop.

The tuning space of CPU involves two dimensions, the degree of unrolling and parallelization. We enumerate these two parameters and profile the execution time to search for the best one. If the unrolling degree is too small, there will not be enough independent instructions to fill in the idle penalty cycles caused by RAW hazards. If it is too large, it will cause I-cache misses. Similarly, the number of threads can neither be too few or too many. If it is too few, the computing cores would have insufficient utilization and memory latency would not be hidden. Too many threads introduce context switching overhead. We rely on the tuning process to look for the best combination.



(a) Direct Accumulation

```
// a[n,k], b[k,m], c[n,m]
Buffer<fp16,16,16> A, B;
Buffer<fp32,16,16> C;
for (i=0; i<n; i+=16)
 for (j=0; j<m; j+=16)
  for (r=0; r<k; r+=16) {
   A = Load(a[i:16,r:16]);
   B = Load(b[r:16,j:16]);
   C += TensorCore(A, B); }
  Store(c[i:16,j:16], C);
```

(b) "p×p Outer Product" Accumulation

```
// a[n,k], b[k,m], c[n,m]
for (i=0; i<n; i+=16*p)
 for (j=0; j<m; j+=16*p)
  for (r=0; r<k; r+=16) {
   Buffer<fp16,16,16> A[p], B[p];
   Buffer<fp32,16,16> C[p][p];
   for (x=0; x<p; ++x) {
    A[x] = Load(a[i+x*16:16,r:16]);
    B[x] = Load(b[r:16,j+x*16:16]); }
   for (x=0; x<p; ++x)
    for (y=0; y<p; ++y)
     C[x][y] += TensorCore(a[x],b[y]); }
   for (x=0; x<p; ++x)
    for (y=0; y<p; ++y)
     Store(c[i+x*16,j+y*16], C[x][y];); }
```

+ Each buffered submatrix reused p times.
+ Loop carried dependences hidden by p×p accumulation.

Figure 8.5: Accumulating a p×p "square window" avoids loop-carried data dependences, and reuses buffered submatrices.

GPU Tuning: On GPU, coarse-grained parallelism is achieved by distributing the data parallel loops across the streaming multiprocessors. Similar to CPU, fine-grained parallelism is also achieved by reordering and unrolling a small degree of data parallel loops to avoid the pipeline penalty caused by loop-carried dependences. Moreover, on GPU, data reuse is explicitly managed by the software. Therefore, as it is shown in Figure 8.5, we adopt an outer-product style matrix multiply accumulation to reuse the buffered submatrices.

Besides the generic optimization, we also developed optimization mechanisms specialized for DNN kernels. Among popular DNN models, there are many layers with relatively small width and height and deep channels. We apply *dimension fusion* to layers with small width and height – these two dimensions are fused into one to save the redundant padding. In addition, we apply *split reduction* to layers with deep channels. For a reduction loop with large trip count, we can split it and parallelize each split segment on `threadIdx`. After all the segments are done, we synchronize the threads and reduce the splitted segments in the shared memory.

8.4 Implementation

In this section, we will discuss technical details in our implementation. UNIT is implemented by extending Apache TVM [53], a full-stack deep learning compiler, with tensorized instruction support. We leverage TVM's tensor DSL, tensor Op, tensor IR infrastructure, and the tuning infrastructure mechanisms [54, 167] to generate high performance kernels. In addition, implementing UNIT on top of TVM enables end-to-end model inference with other optimizations such as operator fusion, in addition to tensorization.

8.4.1 Inspector

The inspector pass is implemented by analyzing TVM's `ComputeOp` data structure. This matches the expression tree of both the instruction and program and enumerates mappings between the loop variables. We enumerate the loops from the tensor's innermost dimension to outermost dimension, and greedily return the first eligible one because of the better potential data locality for inner dimensions. The enumerated mapping provides us with the correspondence of loop variables between the instructions and the tensor operations.

8.4.2 Rewriter

These following steps will be performed by the rewriter:

1. According to the loop correspondence analyzed by the inspector, we reorganize the loops to be tensorized by tiling these loops by the trip counts of the corresponding loops in the instruction, and reorder them to be the innermost loops. These loops will be annotated by a `tensorize` pragma to hint the instruction injection.

2. Based on the strategies discussed in Section 8.3.3, we reorganize the loops above not involved in instruction rewriting to tune the performance.

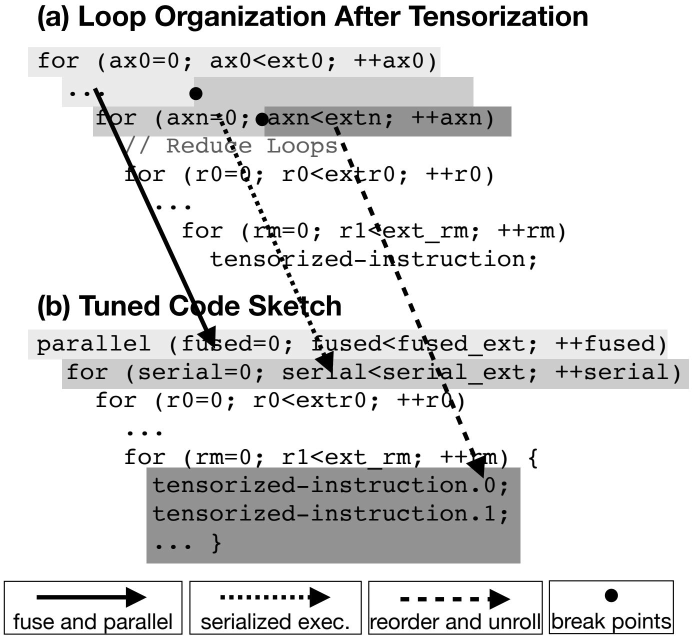
3. We lower the manipulated loop nest to the tensor IR, and replace the loop body annotated with the `tensorize` pragma with the target instructions, as shown in Figure 8.4(c).

Steps 1 and 2 are achieved by invoking TVM scheduling primitives on the tensor DSL level, and step 3 is a tensor IR transformation pass.

Next, we discuss the implementation of the tuning strategies discussed in the last section.

CPU Tuning: The code sketch of tuned CPU code is shown in Figure 8.6. To implement the tuning we discussed in Section 8.3.3, we enumerate two breaking points on the data parallel loop nest, which define how the loop levels are parallelized and unrolled. A breaking point is defined by a *loop level* and *tiling factor*, giving more flexibility to the division. Loops before the first breaking point, will be fused and parallelized. Loops between these two points will be executed in serialized order. Loops after the second breaking point will be reordered to the innermost and unrolled.

GPU Tuning: As it is discussed in the last paragraph of Section 8.3.3, both coarse-grained and fine-grained parallelism optimizations are applied on data-parallel loops, so there is a tradeoff between them: data reuse is increased by increasing the unrolling degree (each buffered submatrix is reused `p` times), but the coarse-grained parallelism is decreased. Also,



(a) Loop Organization After Tensorization

```
for (ax0=0; ax0<ext0; ++ax0)
  ...
    for (axn=0; axn<extn; ++axn)
      // Reduce Loops
      for (r0=0; r0<extr0; ++r0)
        ...
          for (rm=0; r1<ext_rm; ++rm)
            tensorized-instruction;
```

(b) Tuned Code Sketch

```
parallel (fused=0; fused<fused_ext; ++fused)
  for (serial=0; serial<serial_ext; ++serial)
    for (r0=0; r0<extr0; ++r0)
      ...
      for (rm=0; r1<ext_rm; ++rm) {
        tensorized-instruction.0;
        tensorized-instruction.1;
        ... }
```

Figure 8.6: The code sketch of CPU tuning.

a large unrolling degree may overwhelm the register resources. Therefore, the key to generic optimization is to choose a proper unrolling degree.

On the other hand, greedily applying each specialized optimization does not always improve the performance. Though dimension fusion may save the memory traffic, it also introduces software overhead on data rearrangement. Similarly, though splitting the reduction loop introduces more parallelism, it also introduces thread synchronization overhead and register pressure. We enumerate each parameter, including the degree of reduction parallelization and whether to fuse the width and height dimensions, and then apply these transformations to the program and profile the performance to determine which transformation leads to the best performance.

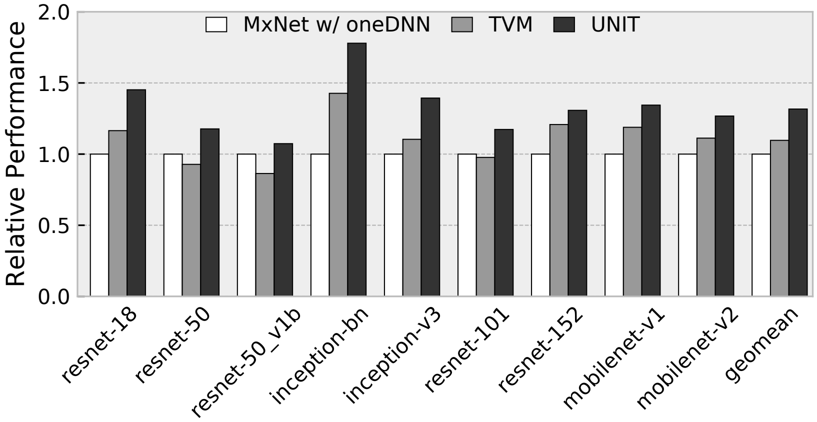


Figure 8.7: Quantized network inference (bs=1) accelerated by Intel VNNI.

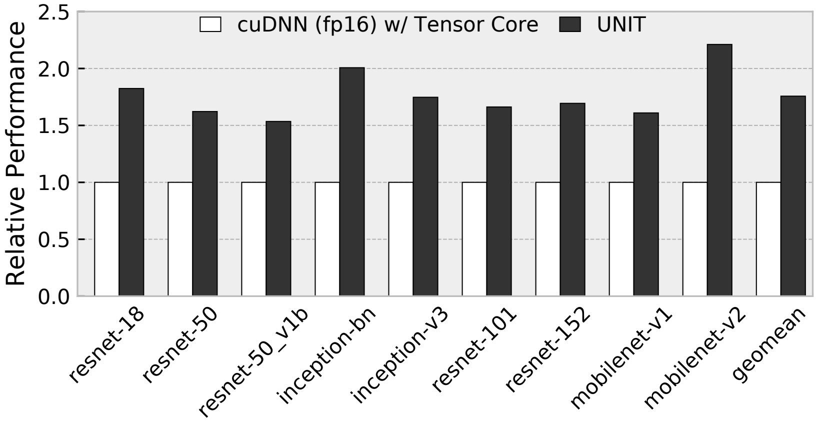


Figure 8.8: Mixed precision network inference (bs=1) accelerated by Tensor Core.

8.5 Methodology

8.5.1 Target Hardware Platforms

We assess UNIT on three hardware platforms:

Intel x86 CPU: We use Amazon EC2 C5.12xlarge instance as our x86 platform with 24-core Intel Xeon Platinum 8275CL CPU @3.00GHz (μarch: Cascade Lake) and 96GB memory.

ARM CPU: We use Amazon EC2 M6g.8xlarge instance as our ARM platform with AWS Graviton2 which features 32-core ARM Cortex-A72 CPU @2.30GHz and 128GB memory.

Nvidia GPU: We use Amazon EC2 P3.2xlarge instance with 16GB host memory as our GPU platform with Nvidia Tesla V100 SXM2 GPU.

8.5.2 Software Frameworks

Code Generation: All programs implemented in Apache TVM are emitted to LLVM IR for code generation. We choose LLVM-10 as our backend, and to be compatible, we use CUDA-10.0 as the NVPTX linker and runtime.

Baseline: We use vendor-provided libraries for baseline performance of operators whenever possible. Specifically, Intel oneDNN v1.6.1 and Nvidia cuDNN 7.6.5 are used as our CPU and GPU baselines, respectively. For end-to-end model inference, we looked for the best available solutions with those libraries, which was MXNet integrated with oneDNN for CPU and TVM integrated with cuDNN for GPU. Another set of baselines is the manually written implementation. To this end, we use the existing TVM solutions for Intel and ARM CPUs, which involve heavy engineering effort to carefully write intrinsics to use Intel VNNI and ARM DOT instructions. We did not find a manually written Tensor Core implementation that covers our evaluated workloads.

8.5.3 Workloads

DNN Models: All DNN models are from the MXNet Model Zoo and converted to TVM's graph IR, Relay [222], for quantization [127], layout transformation, and data padding. All these models adopt `NCHW[x]c` data layout [167] for the data and `KCRS[y]k[x]c` for the kernel. Here `N` denotes the batch size, `C` denotes the input channels, `H` and `W` are the width and height of the input image, and `[x]c` denotes that the original `C` is split by `x`. Similarly, `K` denotes the number of output channels, `R` and `S` are the height and width of the kernel, and `[y]k` denotes the original dimension `K` is split by `y`. `[x]` equals to the number of lanes of the instruction output, and `[y]` equals to the width of reduction.

In the evaluation, we target the `N=1` cases, because it is hard to optimize but critical for inference use cases. Comparing with batched cases where `N>1`, we cannot reuse the kernel tensor across samples, or exploit the parallelism brought by the data-parallel batching dimension.

8.6 Evaluation

Our evaluation of UNIT attempts to answer these questions:

1. What is the performance of the end-to-end deep learning model inference powered by *tensorized* instructions?
2. How does each optimization technique that UNIT uses impact the performance?
3. Can UNIT be extended to support new hardware platforms and tensor operations?

8.6.1 End-to-End Performance

In this subsection, we show the UNIT end-to-end effectiveness on Intel x86 and Nvidia GPU processors for tensorizing mixed precision instructions. For Intel x86 experiments, we use MXNet integrated with Intel oneDNN (referred to as MXNet-oneDNN) as the baseline. Another comparison of ours is TVM with manually written schedules using Intel's VNNI instruction. The findings of this experiment are shown in Figure 8.7.

We observe that UNIT achieves significant speedup compared to MXNet-oneDNN. Note that Intel oneDNN has access to manually written schedules that have been aggressively optimized and tuned by domain experts. We also observe that TVM overall achieves better performance than MXNet-oneDNN, but has suboptimal performance on resnet50 and resnet50b, which were heavily tuned by oneDNN engineers. On the other hand, UNIT outperforms both baselines, by 1.3× over MXNet-oneDNN and by 1.18× over TVM.

Next, we test the efficacy of UNIT on utilizing Nvidia Tensor Core instructions for Nvidia GPUs. For the baseline, we integrate TVM with cuDNN, which has access to manually written aggressively tuned Tensor Core schedules. The findings of this experiment are shown in Figure 8.8. We observe that UNIT consistently achieves better performance than cuDNN with a mean speedup of 1.75× and up to 2.2×.

8.6.2 Optimization Implications

In this subsection, we focus on the convolution operators of the DNN models to perform an in-depth analysis of the impact of different optimization techniques used by UNIT's Rewriter. This is essentially an ablation study, showing how important different parts of UNIT are. There are 148 different convolution workloads (i.e., convolution with different feature map sizes, kernel sizes, strides, etc.) in the models, out of which we choose 16 representative convolution layers. These kernels cover diverse input shapes and strides. Other workloads behave similarly in the ablation study. We summarize the characteristics, namely, convolution attributes, like shapes, strides, etc., of the selected workloads in Table 8.1.

Intel x86 servers: As we discussed in Section 8.3.3, we have two breaking points in CPU scheduling. The loop nests before the first breaking point are parallelized and the loop nests after the second breaking point are unrolled, while the ones in between the breaking point are executed serially. As loop nests can either be parallelized or unrolled (remaining one is serialized), we have a search space represented by the tuning pairs. Rewriter tunes this search space to generate a high-performance kernel. In this experiment, we incrementally measure the performance improvements brought by parallelizing, unrolling and tuning. The findings of this experiment are shown in Figure 8.9, normalizing the speedup to Intel oneDNN execution latency.

First we fuse outer loop nests such that the loop bound of the fused loop nest is < 3000, and measure the latency of the resulting kernel (shown by *Parallel*). Then, we take the remaining loop nests, and tile and unroll them such the unrolling factor is < 8, and measure this performance (shown by *+Unroll*). Finally, instead of setting the limits as 3000 and 8, we tune the search space and measure performance (shown by *+Tune*), getting the final latency UNIT achieves. We observe that Parallel and Unroll together is responsible for most of the speedup. The additional speedup introduced by Tuning is quite small. It turns out that more than half of the kernels get the optimal performance on the first tuning pair (i.e.

3000 and 8), and more than 95% of the kernels get the optimal performance within the first 8 tuning pairs.

CPU does poorly on workloads #1 and #4, because their output shapes (OH/OW) can neither be perfectly tiled nor fully unrolled. Inherited from TVM, loop residues are handled the by guarding it with a likely clause, which results in an if-branch that harms the performance.

Nvidia GPU servers: As discussed in Section 8.3.3, we employ three optimizations on GPU: generic coarse- and fine-grained parallelism, fusing width and height to save memory bandwidth, and parallelizing the reduction dimension. In this subsection, we study the impact of these optimizations on the performance. We show the findings in Figure 8.10, normalizing the speedup to Nvidia cuDNN.

According to our evaluation, any unrolling degree (`p` in Figure 8.5) larger than 2 may overwhelm the registers, so we use `p=2` to apply the generic optimization. The generic optimization already beat cuDNN in most cases (shown by *Generic*). Then, depending on the height and width values, Rewriter fuses the height and width dimensions to save memory bandwidth (shown by *+FuseDim*). Then, we split the reduction dimension K by 64 and measure the performance (*+SplitK*). Finally, we let Rewriter to choose the sizes for these 3 optimizations and measure performance (shown by *+Tune*).

We observe that SplitK leads to the maximal speedup, as it leads to significant parallelism and keeps the Tensor Cores busy. More than 70% of the kernels can get high performance by employing fusion and parallelizing the reduction dimension. Similar to CPUs, the additional speedup by tuning is small.

UNIT cannot outperform cuDNN on #1 and #15, because the strided data accesses lead to less data locality. However, since these adversarial cases (both CPU and GPU) only occupy a very small portion among all these models, we can still outperform vendor-provided libraries because of the generality of our optimization.

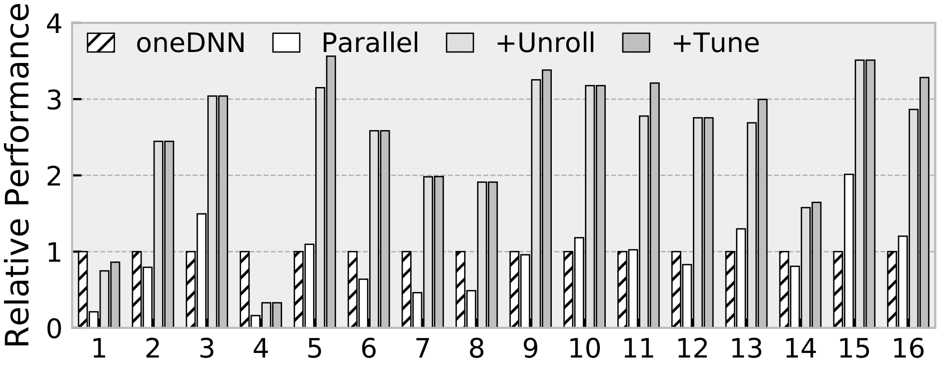


Figure 8.9: The performance impact of the code space exploration.

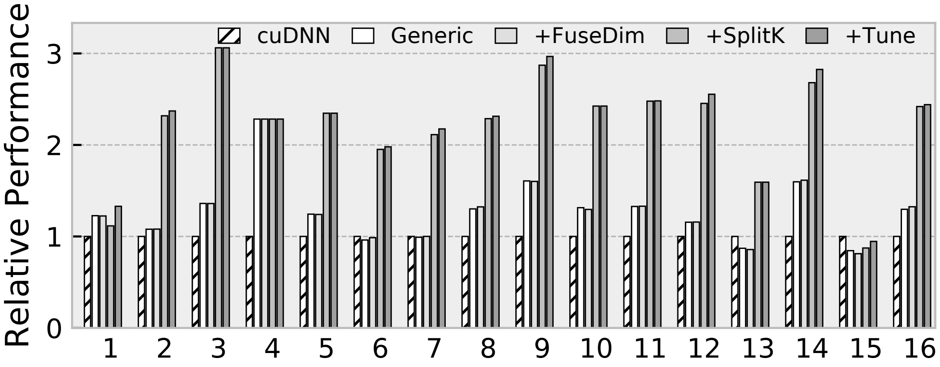


Figure 8.10: The performance impact of the code space exploration.

8.6.3 Extensibility

We evaluate the extensibility of UNIT in two aspects: to new hardware platforms and to new deep learning tensor operations. We observe that by just representing the semantics of the new tensorized instruction in tensor DSL, UNIT can easily extend to new tensorized instructions and tensor operations.

New Hardware Platforms: To demonstrate the capability of extending to new hardware platforms, we apply UNIT to an ARM CPU supporting the ARM DOT instruction. To the best of our knowledge, there is a lack of a deep learning framework with well-integrated ARM backend library support. In the absence of a framework baseline, we choose TVM compiling to ARM Neon assembly as the baseline (shown by TVM-NEON). Additionally, we find that TVM has manually-written schedules using ARM DOT instructions, which forms our second

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
C	288	160	1056	80	128	192	256	1024	128	576	96	1024	576	64	64	608
IHW	35	9	7	73	16	16	16	14	16	14	16	14	14	29	56	14
K	384	224	192	192	128	192	256	512	160	192	128	256	128	96	128	192
R=S	3	3	1	3	3	3	3	1	3	1	3	1	1	3	1	1
Stride	2	1	1	1	1	1	1	1	1	1	1	1	1	1	2	1
OHW	17	7	7	71	14	14	14	14	14	14	14	14	14	27	28	14

Table 8.1: Characteristics of the selected convolution layers.

comparison baseline (shown by TVM-Manual). Note that in contrast to UNIT's automatic approach, this is a manually written schedule requiring intense engineering efforts. Finally, we represent the semantics of ARM DOT instruction in UNIT's tensor DSL and use UNIT to compile the models. The findings of this experiment are shown in Figure 8.11, showing normalized speedup compared to the TVM-Neon baseline. The results show that UNIT consistently outperforms both TVM-NEON and TVM-Manual, proving UNIT's effectiveness in extending to new hardware platforms.

3D Convolution: We test UNIT on 3D convolution operation for mapping Intel VNNI tensorized instructions. Note that this does not require any changes from UNIT perspective; we are just giving a new input (tensor-level IR for conv3d) to UNIT. To evaluate this extensibility, we take all the 2D convolutions from Resnet18 and manually convert them to 3D convolutions. We then apply UNIT on these kernels and show the speedup compared to oneDNN baseline in Figure 8.12. We observe that UNIT easily extends to 3D convolution, as it has comparable performance for many convolution kernels, with an average of 1.2× speedup.

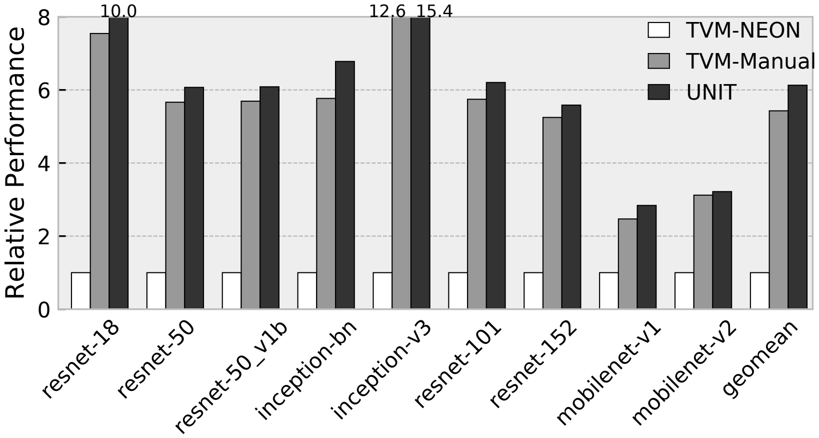


Figure 8.11: The performance of ARM on model inference.

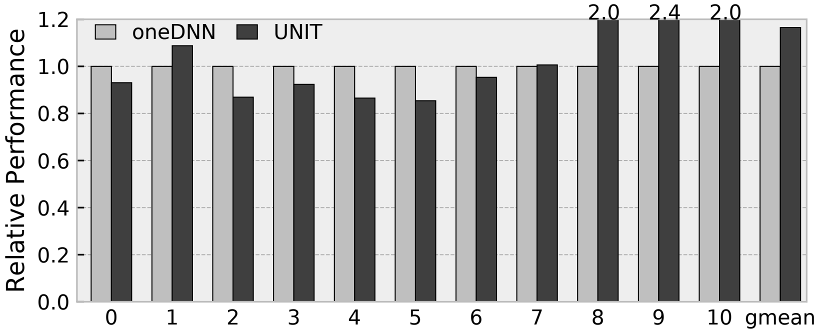


Figure 8.12: The performance of each layer on res18-3d.

8.7 Related Work

Compilation support for hardware intrinsics: There exists a large body of literature on compilation support for various hardware intrinsics [223, 148, 194, 207, 84, 158, 195, 82, 249, 240]. Existing production compilers such as GCC and LLVM implement auto-vectorization to leverage SIMD intrinsics. Prior works such as [148, 223] propose various approaches to further improve the performance of the auto-vectorizer. These approaches cannot be extended to support tensor computation intrinsics which introduce "horizontal computation" within each lane. To better exploit this programming behavior, multiple IRs [109, 89, 119] were developed for more effective compiler analysis and transformation. TVM [53] implements an extensible interface to support new hardware intrinsics that are not limited to SIMD instructions.

However, programmers need to transform the program to match the behavior of the intrinsics and declare the lowering rule for the intrinsics prior to compilation. Many prior works seek to automate software tuning, AutoTVM [54] still requires users to define the space of code organization. Ansor [298] adopts rule-based strategies to automatically define this space. However, both Ansor and AutoTVM still relies on execution-based sampling to determine the quality of tuning, while some recent works develop cost model [296, 243, 288, 134] cost model based. All these related works mentioned above focus on single kernel acceleration, and many state-of-art works have shifted their focus to large scale and distributed parallelism [90, 282, 299]. In addition, there are also recent work to improve TVM's test case coverage [165], and recent domain-specific language development no longer limit to the ML/AI domain [169].

Software/Hardware Co-designed Accelerators: By carefully study the memory/computation pattern, software/hardware co-designed accelerators [294, 293] can achieve orders-of-magnitude performance gain in the tensor operation domain.

Moreover, the analysis pass of UNIT is inspired by the decoupled-access execute (DAE) architectures [154, 210, 193, 273, 74, 276, 270]. Computation and data access are decoupled and specialized separately. The computation is offloaded onto a programmable data path and data access is encoded in hardware intrinsics and executed on a specialized address generation unit (AGU). UNIT adopts a reversed approach, it matches computation on a fixed data path, and analyzes data access fed to the data path.

Polyhedral model: Many prior works have built program analysis and transformation frameworks based on the polyhedral model for tensor programs [148, 249, 240, 82, 256, 70, 181, 259, 106]. Loop Tactics [49] is one representative work which matches the pre-defined computation patterns in the polyhedral IR and transforms the matched patterns to optimized programs. UNIT distinguishes itself from Loop Tactics in: 1) Compared with the schedule tree [260] in the polyhedral model, the tensor DSL provides more information such as loop reduction properties and operand types; 2) UNIT provides an end-to-end solution including auto-tuning to obtain the optimal performance, whereas Loop Tactics requires the optimized schedules to be provided manually.

Deep learning frameworks: UNIT is complementary to the existing deep learning frameworks. Existing frameworks such as Tensorflow [18], PyTorch [17], and MXNet [13] rely on vendor-crafted libraries to support the new tensor intrinsics. TVM [53] requires code rewriting at the user side. UNIT is able to handle new operators which might not be covered by the vendor libraries and spare the user from having to perform manual re-writing. We have demonstrated the effectiveness of the methodology of UNIT based on TVM. Similar technique can be applied to other frameworks to further boost their performance.

CHAPTER 9

Discussion

To conclude this dissertation, we first summarize lessons learned from the works discussed in prior chapters, and then discuss the open questions of this research area.

9.1 Reforming Full-Stack Accelerator Design Paradigm

The research projects presented in this dissertation primarily aim to push the boundaries of full-stack specialized accelerators. Specifically, we propose a reform of the design paradigm for specialized accelerators, where each prior software/hardware co-designed innovation should be modularized and comprised in a unified design space for future reuse. By identifying commonalities among accelerators within this design space, we can adopt a high-level mainstream programming language with moderate extensions as the unified programming interface. This approach makes porting legacy workloads to our accelerator easy, although it does impose challenges to the development of compilation techniques. An alternate approach to build software stacks for new specialized accelerators is to develop domain-specific programming languages. The tradeoff of choosing different programming interfaces for non-conventional execution models will be elaborated in the next section (Section 9.2).

Besides serving as a bridge between the software and the hardware, the applications written in this programming interface also naturally encode the demands of accelerator design. Leveraging the compiler's awareness of these demands, specialized accelerator design automation can be realized. Through our evaluation, we have found that the automatically generated accelerators exhibit comparable quality to prior handcrafted ones.

In addition to advancing full-stack specialized accelerators, this dissertation also demonstrates how techniques developed for specialized accelerators and general-purpose processors can mutually enhance each other when applied to relevant problems. For instance, ScalarEvolution, initially developed for automatically generating vector instructions for general-purpose ISAs like X86 AVX512 and ARM NEON, is utilized in Chapter 4 to analyze and encode specific memory access patterns for memory stream commands. Similarly, the compilation techniques employed to detect applicability and rewrite programs with tensorized instruction are derived from finding a graph isomorphism from the computational graph to the spatial datapath.

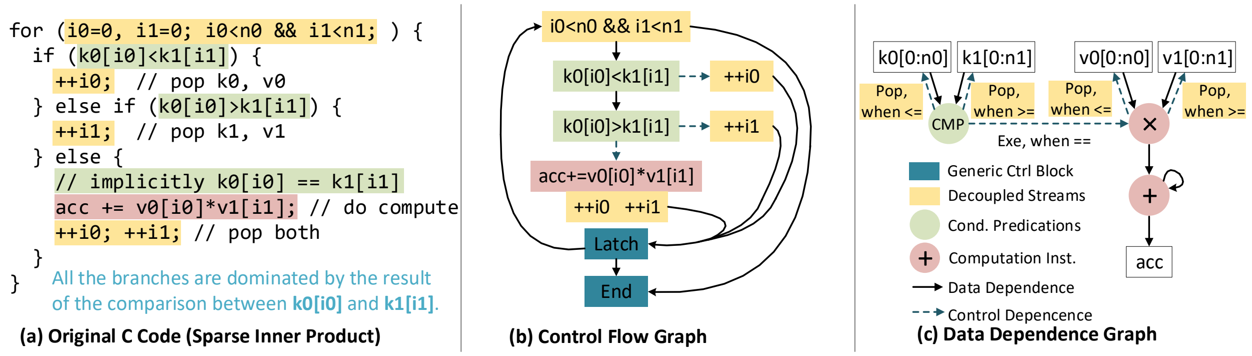
Another future direction of this work is to broaden the applicability of full-stack specialized accelerators. We envision a promising applicable area to be mobile SoCs. Over the past decade, mobile SoCs produced by companies like Apple and Qualcomm have employed numerous specialized blocks, with their numbers continually increasing. These specialized blocks consume a significant amount of on-chip power and area. Prior research [193, 276, 278, 166, 55, 273] has demonstrated that a unified programmable accelerator can achieve performance close to that of individual specialized blocks, while maintaining flexibility with only a moderate (or even better) power/area overhead compared to a combination of specialized blocks. Hence, we believe it is the opportune time to reconsider the design of specialized blocks on SoCs by adopting unified programmable accelerators. This would save significant effort in developing new accelerators for each generation, and sometimes software tuning alone can meet the performance requirements. Given the proven promise, the question arises as to why programmable accelerators are not yet widely adopted in industries. Apart from legacy issues, we argue that this is due to the insufficiency of various aspects, including operating systems and design approaches, which will be discussed in Section 9.3.

9.2 Software Stacks for Accelerators

While the hardware development itself has been well studied in many early works on programmable accelerators, the importance of the software stack has often been overlooked. This neglect of the software stack has hindered the adoption of specialized accelerators, as developers have had to program in low-level interfaces that manage excessively exposed details. The aggressively broken and reformed execution model of these accelerators has made it challenging to develop high-level programming interfaces and compilation techniques. To address this issue, there are two main approaches that have been used to develop software stacks for these accelerators: domain-specific and legacy.

Domain-Specific Language Some prior works have focused on developing domain-specific languages (DSLs) [53, 214, 7, 29, 147, 145] to program domain-specific accelerators. These DSLs encode additional domain knowledge in the programming interface, allowing compilers to better optimize applications for the underlying hardware. For example, tensor domain-specific languages [53, 214, 256] offer opportunities to automatically tune the performance of tensor applications by maximizing data locality. Additionally, by exposing fixed primitives, DSLs like Spatial [147] can generate high- performance code by taking advantage of known parallel patterns.

The tradeoff of this domain-specific approach is the learning curve and portability. Developers must learn a new programming model when targeting a new application domain, which may limit potential users. An example of this tradeoff is choosing between Tensorflow and PyTorch. While Tensorflow's IR builder interface may be counterintuitive to many ML/AI researchers, PyTorch's interface is closer to numpy and is preferred by more ML/AI researchers [8]. Moreover, because of the domain-specific programmability, many legacy applications written in legacy programming interfaces should be redeveloped. It will impose significant inconvenience on a big team to thoroughly shift the infrastructures, considering the path dependences.



```
for (i0=0, i1=0; i0<n0 && i1<n1; ) {
  if (k0[i0]<k1[i1]) {
    ++i0;  // pop k0, v0
  } else if (k0[i0]>k1[i1]) {
    ++i1;  // pop k1, v1
  } else {
    // implicitly k0[i0] == k1[i1]
    acc += v0[i0]*v1[i1]; // do compute
    ++i0; ++i1; // pop both
  }
}
```

All the branches are dominated by the result of the comparison between **k0[i0]** and **k1[i1]**.

(a) Original C Code (Sparse Inner Product)

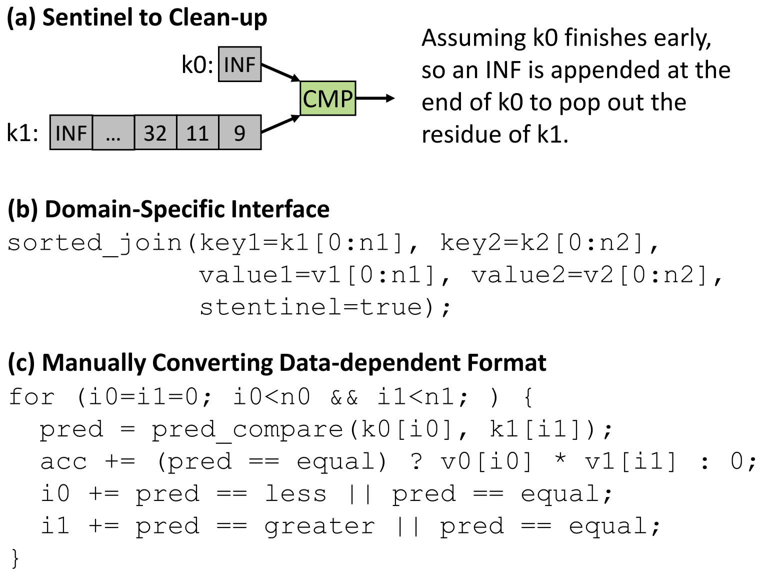
(b) Control Flow Graph

(c) Data Dependence Graph

Figure 9.1: A key-value join implemented in a general-purpose language, and compiled to a specialized accelerator.

Legacy Approach The legacy approach involves making no or moderate modifications to the existing programming interface, allowing for easier porting of applications to the newly developed software stack. Examples of this approach include CUDA and HLS. Developers can utilize their existing C-programming knowledge to rapidly develop applications or even tune the performance. This approach is also adopted in core projects discussed in this dissertation, such as DSAGEN and OverGen.

However, the legacy approach has its tradeoff in terms of the expressiveness. Most existing programming models were designed for the Von-Neumann's imperative execution, so to effectively map this programming paradigm to the accelerators with aggressively broken and reformed execution models can pose challenges for both compiler and application developers. Some parallelism and execution that was not originally encoded and exploited in conventional programming requires additional hints from the developers. For example, in high-level synthesis, the developers are required to annotate the program to explicitly express the memory banking, loop pipelining, and the data produce/consume flow. Although some state-of-art research on HLS [239, 237] seeks to automatically annotate these hints on the behalf of developers, it still takes excessive time to explore the space of code transformation.



(a) Sentinel to Clean-up

Assuming k0 finishes early, so an INF is appended at the end of k0 to pop out the residue of k1.

(b) Domain-Specific Interface

```
sorted_join(key1=k1[0:n1], key2=k2[0:n2],
            value1=v1[0:n1], value2=v2[0:n2],
            stentinel=true);
```

(c) Manually Converting Data-dependent Format

```
for (i0=i1=0; i0<n0 && i1<n1; ) {
  pred = pred_compare(k0[i0], k1[i1]);
  acc += (pred == equal) ? v0[i0] * v1[i1] : 0;
  i0 += pred == less || pred == equal;
  i1 += pred == greater || pred == equal;
}
```

Figure 9.2: Injecting sentinels for clean-up; choosing the right abstraction.

Discussion While there is promise in having a general-purpose abstraction for full-stack specialized accelerators, it is not yet perfect.

Consider the example depicted in Figure 9.1. Implementing this in a general-purpose imperative language necessitates a complex transformation from the imperative control flow in Figure 9.1(b) into the dataflow presented in Figure 9.1(c). Successfully converting this general-purpose imperative code into accelerator commands demands strong assumptions about both loop iterations and branch organizations.

As illustrated in Figure 9.1(a), there are two explicit conditions ("if" and "else if") and an implicit one (the final "else"), comparing `k0[i0]` and `k1[i1]`. These conditions dictate the progression of the index pointers of `v0[]` and `v1[]`. This intertwining memory access and computation complicates the loop iteration analysis. `ScalarEvolution` analysis is best suited for canonical loops, i.e. loops with only one inductive variable and increased by one each iteration. Thus, to analyze such a loop with multiple conditionally increased inductive variables and various termination conditions, we could only rely on very strong assumption on the loop structure to match the pattern of a key-value join template: there are three comparison conditions, which share the same operands in contiguous branches, and these two loop variables will be increase accordingly in each branch.

Beyond detecting the key-value join behavior itself, to clear the remnants of a late-terminate stream, sentinels after each array should be injected, as shown in Figure 9.2(a). The compiler can automatically generate the sentinels, but this deviates from the intended purpose of faithful translating the program. Adopting a domain-specific programming interface would significantly simplify compiler analysis and code generation, and the sentinel can be a part of the semantics. However, this places a heavy burden on compiler developers, who need to continually expand the domain-specific software stack to accommodate new programming patterns.

An alternative solution in the middle (between the arbitrary control flow and the domain-specific programming) is to retain the general-purpose imperative programming while manually embedding a predication-based code segment within the loop body, as demonstrated in Figure 9.2(c). To manage the late-terminate stream, the developer can just manually append a sentinel to the end of each array. While this approach mildly increases the workload for application developers, it notably amortizes the difficulties for compiler analysis. Nonetheless, there is still one pressing question remains unresolved: a well-structured principle should be formalized to analyze the intertwined conditional increases of loop variables and the various loop termination conditions.

9.3 Future Research and Open Questions

As the landscape of specialized accelerators continue to evolve, so must our code design approaches and understanding. In this section, several future directions may potentially usher in transformative advancements in this area.

Virtualizing Decoupled-spatial Accelerators Previous research, both within this dissertation and externally, has highlighted the potential of decoupled-spatial architectures which achieves near-ASIC performance and energy saving under a moderate overhead. While the software/hardware interfaces and compiler transformations have been well studied, system-wide support remains underexplored. This hinders the broader adoption of this innovative architectural paradigm. The distinct nature of this paradigm paves the way for ground-breaking research questions on operating system support.

For instance, mobile operating systems like Android and iOS already well manage tightly coupled specialized on-chip blocks. Unlike these dedicated ASICs, which are tailored and dedicated for specific applications, a decoupled-spatial architecture's compute resources can be shared across multiple applications. Yet, the current mobile operating systems have not been sufficiently investigated to manage application concurrency on a spatial architecture.

Though there is some capability for desktop operating systems (e.g. Windows and *nix-like systems) to manage the application concurrency on PCIe devices (i.e. loosely coupled), like GPUs. However, GPUs still adopt thread-based execution, so many techniques from CPU virtualization, like context switching, and isolation can be reused. For a spatial accelerator, as mentioned before, all the instructions are locally buffered in spatial PE's, which makes context switching more complicated than a mere adjustment of the program counter. In addition, compared with CPU's centralized register file, all the intermediate results are either buffered in PE's local register file or routed through the on-chip network, which complicates preserving the intermediate states during application switches.

The research question most akin to our needs, namely virtualizing a decoupled spatial architecture, lies in multi-tenancy FPGA execution. FPGA vendors have leaned to hardware solutions, while virtualizing FPGA for multi-tenancy execution is still an open question on academic research, and attracts significant attention [149, 170, 157]. Decoupled-spatial architectures distinguished themselves from FPGAs because of their programming interfaces, and granularity. FPGAs still rely on low-level RTL programming interfaces, and take substantial duration to reconfigure (ms order). Instead of switching the context, to ensure the performance of executing different applications, excessive on-chip resources are reserved for different application instances. On the other hand, the coarse granularity of decoupled spatial architectures requires only a brief period of cycles to reconfigure, which makes frequent fine-grain context switching a feasible venture.

Because of the uniqueness discussed above, one promising direction is to develop operating systems tailored for decoupled-spatial computing resource management.

Context-Aware Accelerator Synthesis We currently adopt the perf/mm^2 as the objective for our hardware design space exploration in our design automation discussed in Chapter 6, which is also the implication of Moore's Law, maximizing the number of on-chip active transistors. This is already proven to be effective to generate accelerators with comparable quality compared against handcrafted accelerators.

In the future, integrating further insights regarding the use context of accelerators could enhance the quality of the generated hardware. Specifically, the performance of an accelerator could be fine-tuned in terms of latency, throughput, power, and clock frequency depending on the usage scenario. For instance, in real-time systems such as self-driving vehicles or graphics rendering, latency becomes a paramount concern. In these cases, eliminating superfluous switches and FIFOs from the data path could substantially improve response times. On the other hand, for applications that resist parallelization across multiple cores, increasing the clock frequency might emerge as a viable alternative strategy.

To facilitate such optimizations, a more comprehensive pre-synthesis estimation that encompasses power, area, and frequency is essential.

Unifying Simulation and Implementation The current architectural research and development stick on such a familiar pattern: When architects introduce a novel feature, its efficacy is initially ascertained through functional simulation. If these simulated results prove promising, this feature is then handed over to the implementation team for RTL design. This approach, which we consistently employ in the design and execution of both DSAGEN and OverGen, presents coordination challenges. It necessitates seamless alignment between teams regarding performance metrics and intricate technical specifics. Ideally, there should be a unified development interface that serves both simulation and implementation phases.

By carefully studying the execution model of the RTL interface, we discern event-driven programming a promising opportunity for this goal. In this paradigm, each RTL module acts in response to specific signals, activating the logic contained within. This methodology, prevalent in UI design and distributed systems, might well be repurposed to innovate the hardware design landscape.

9.4 Conclusion

An application domain does not inherently dictate an accelerator design. Rather, the specific program idioms within these applications do. Based on this insight, this dissertation unveils the potential for automated, programmable accelerator design. Through careful crafting, each co-designed software/hardware innovation can be modularized and encapsulated within a comprehensive, universally defined design space. Moreover, the correspondences between software and hardware can be understandable by a domain-agnostic compiler.

The compiler serves a dual purpose: it is not only a bridge between software and hardware but also a navigator for hardware design space exploration. Harnessing the compiler's awareness of software/hardware synergies, alongside design requirements innately embedded within applications, can foster effective design space exploration.

Beyond the realm of automation, we posit that the true transformative potential lies in reshaping the foundational design principles of specialized accelerators. Subsequent innovations can be realized by augmenting this design space, rendering them reusable across diverse application domains. The era of initiating accelerator designs from a blank slate may be nearing its end.

REFERENCES

[1] "Aws announces aqua for amazon redshift." [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2020/12/aws-announces-aqua-for-amazon-redshift-preview/

[2] "Aws inferentia." [Online]. Available: https://aws.amazon.com/machine-learning/inferentia/

[3] "Brainwave project." [Online]. Available: https://www.microsoft.com/en-us/research/project/project-brainwave/

[4] "MicroBlaze processor reference guide."

[5] "Nios II processor reference guide."

[6] "Nvidia tensor cores." [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/

[7] "The scala programming language." [Online]. Available: https://scala-lang.org/

[8] "The state of ml frameworks in 2019." [Online]. Available: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

[9] "Tuning guide for deep learning with intel(r) avx512 and intel(r) deep learning boost on 3rd generation intel(r) xeon(r) scalable processors." [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/guide/deep-learning-with-avx512-and-dl-boost.html

[10] "Xla: Domain-specific compiler for linear algebra to optimizes tensorflow computations." [Online]. Available: https://www.tensorflow.org/performance/xla

[11] "Exploring the Arm dot product instructions," https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/exploring-the-arm-dot-product-instructions, 2017.

[12] "Introduction to Intel deep learning boost on second generation Intel Xeon scalable processors," https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-deep-learning-boost-on-second-generation-intel-xeon-scalable.html, 2019.

[13] "Apache MXNet — a flexible and efficient library for deep learning." https://mxnet.apache.org/versions/1.6/, 2020.

[14] "Nvidia CUDA® deep neural network library (cuDNN)," https://developer.nvidia.com/cudnn, 2020.

[15] "Nvidia tensor cores," https://www.nvidia.com/en-us/data-center/tensor-cores/, 2020.

[16] "oneAPI deep neural network library (oneDNN)," https://github.com/oneapi-src/oneDNN, 2020.

[17] "Pytorch," https://pytorch.org/, 2020.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026877.3026899

[19] R. B. Abdelhamid, Y. Yamaguchi, and T. Boku, "A highly-efficient and tightly-connected many-core overlay architecture," *IEEE Access*, vol. 9, pp. 65 277–65 292, 2021.

[20] M. Afarin, C. Gao, S. Rahman, N. Abu-Ghazaleh, and R. Gupta, "Commongraph: Graph analytics on evolving data," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 133–145. [Online]. Available: https://doi.org/10.1145/3575693.3575713

[21] M. Al Kadi, B. Janssen, and M. Huebner, "FGPU: An SIMT-architecture for FPGAs," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–263.

[22] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 1–13.

[23] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[24] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GP-GPU for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230–237.

[25] M. Annaratone, E. A. Arnould, T. Gross, H. T. Kung, M. S. Lam, O. Menzilcioglu, and J. A. Webb, "The Warp Computer: Architecture, Implementation, and Performance," *IEEE Transactions on Computers*, 1987.

[26] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: a coarse grained reconfigurable architectural template," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 1062–1074, 2010.

[27] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[28] M. Auguin, F. Boeri, and E. Carriere, "Automatic exploration of vliw processor architectures from a designer's experience based specification," in *Third International Workshop on Hardware/Software Codesign*, Sep 1994.

[29] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *49th DAC*, 2012.

[30] R. Bahr, C. Barrett, N. Bhagdikar, A. Carsello, R. Daly, C. Donovick, D. Durst, K. Fatahalian, K. Feng, P. Hanrahan, T. Hofstee, M. Horowitz, D. Huff, F. Kjolstad, T. Kong, Q. Liu, M. Mann, J. Melchert, A. Nayak, A. Niemetz, G. Nyengele, P. Raina, S. Richardson, R. Setaluri, J. Setter, K. Sreedhar, M. Strange, J. Thomas, C. Torng, L. Truong, N. Tsiskaridze, and K. Zhang, "Creating an agile hardware design flow," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[31] R. Balasubramanian, V. Gangadhar, Z. Guo, C.-H. Ho, C. Joseph, J. Menon, M. P. Drumond, R. Paul, S. Prasad, P. Valathol, and K. Sankaralingam, "Enabling GP-GPU low-level hardware explorations with MIAOW: An open-source RTL implementation of a GP-GPU," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 2, Jun. 2015.

[32] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang, M. Matl, and D. Wentzlaff, "Openpiton: An open source manycore research framework," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872414

[33] T. K. Bandara, D. Wijerathne, T. Mitra, and L.-S. Peh, "REVAMP: A systematic framework for heterogeneous CGRA realization," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 918–932.

[34] S. Basalama, A. Sohrabizadeh, J. Wang, L. Guo, and J. Cong, “Flexcnn: An end-to-end framework for composing cnn accelerators on fpga,” *ACM Transactions on Reconfigurable Technology and Systems*, 2022.

[35] A. Becker, S. Sirowy, and F. Vahid, “Just-in-time compilation for FPGA processor cores,” in *2011 Electronic System Level Synthesis Conference (ESLsyn)*, 2011, pp. 1–6.

[36] R. Ben Abdelhamid, Y. Yamaguchi, and T. Boku, “MITRACA: A next-gen heterogeneous architecture,” in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2019, pp. 304–311.

[37] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, “Design, integration and implementation of the DySER hardware accelerator into OpenSPARC,” in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[38] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, “Cyclo-static data flow,” in *1995 International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, May 1995, pp. 3255–3258 vol.5.

[39] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, 2011.

[40] N. Bleier, M. H. Mubarik, S. Chakraborty, S. Kishore, and R. Kumar, “Rethinking programmable earable processors,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 454–467. [Online]. Available: https://doi.org/10.1145/3470496.3527396

[41] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, “Architectural exploration of the ADRES coarse-grained reconfigurable array,” in *ARC 2007*.

[42] A. Brant and G. G. Lemieux, “ZUMA: An open FPGA overlay architecture,” in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012, pp. 93–96.

[43] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein, “Spatial Computation,” in *ASPLOS XI*.

[44] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and t. T. Team, “Scaling to the end of silicon with EDGE architectures,” *Computer*, 2004.

[45] D. S. Cali, K. Kanellopoulos, J. Lindegger, Z. Bingöl, G. S. Kalsi, Z. Zuo, C. Firtina, M. B. Cavlak, J. Kim, N. M. Ghiasi, G. Singh, J. Gómez-Luna, N. A. Alserr, M. Alser, S. Subramoney, C. Alkan, S. Ghose, and O. Mutlu, "Segram: A universal hardware accelerator for genomic sequence-to-graph and sequence-to-sequence mapping," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 638–655. [Online]. Available: https://doi.org/10.1145/3470496.3527436

[46] D. Capalija and T. S. Abdelrahman, "Towards synthesis-free JIT compilation to commodity FPGAs," in *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2011, pp. 202–205.

[47] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, "The IDEA DSP block-based soft processor for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 3, Sep. 2014.

[48] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: a DSP block based FPGA soft processor," in *2012 International Conference on Field-Programmable Technology*, 2012, pp. 151–158.

[49] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal, "Declarative loop tactics for domain-specific optimization," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 4, pp. 1–25, 2019.

[50] H. Chen, M. Zhang, K. Yang, K. Chen, A. Zomaya, Y. Wu, and X. Qian, "Achieving sub-second pairwise query over evolving graphs," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–15. [Online]. Available: https://doi.org/10.1145/3575693.3576173

[51] J. Chen and X. Qian, "Khuzdul: Efficient and scalable distributed graph pattern mining engine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 413–426. [Online]. Available: https://doi.org/10.1145/3575693.3575743

[52] T. Chen, S. Srinath, C. Batten, and G. E. Suh, "An architectural framework for accelerating dynamic parallel algorithms on reconfigurable hardware," in *MICRO*, 2018.

[53] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *13th OSDI*, 2018.

[54] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," in *Advances in Neural Information Processing Systems*, 2018, pp. 3389–3400.

[55] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," in *19th ASPLOS*. ACM, 2014.

[56] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 367–379. [Online]. Available: https://doi.org/10.1109/ISCA.2016.40

[57] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[58] L. Cheng, M. Ruttenberg, D. C. Jung, D. Richmond, M. Taylor, M. Oskin, and C. Batten, "Beyond static parallel loops: Supporting dynamic task parallelism on manycore architectures with software-managed scratchpad memories," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 46–58. [Online]. Available: https://doi.org/10.1145/3582016.3582020

[59] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 204–213.

[60] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 204–213.

[61] Y. Chi, W. Qiao, A. Sohrabizadeh, J. Wang, and J. Cong, "Democratizing domain-specific computing," *Commun. ACM*, vol. 66, no. 1, p. 74–85, dec 2022. [Online]. Available: https://doi.org/10.1145/3524108

[62] S. A. Chin, N. Sakamoto, A. Rui, J. Zhao, J. H. Kim, Y. Hara-Azumi, and J. Anderson, "CGRA-ME: a unified framework for cgra modelling and exploration," in *28th ASAP*, July 2017.

[63] S. A. Chin and J. H. Anderson, "An architecture-agnostic integer linear programming approach to cgra mapping," in *55th DAC*, 2018.

[64] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The Reconfigurable Streaming Vector Processor (RSVP)," in *36th MICRO*. IEEE, 2003.

[65] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[66] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014, pp. 1–6.

[67] J. Cong, M. Gill, Y. Hao, G. Reinman, and B. Yuan, "On-chip interconnection network for accelerator-rich architectures," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[68] J. Cong, H. Huang, C. Liu, and Y. Zou, "A reuse-aware prefetching scheme for scratchpad memory," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2011, pp. 960–965.

[69] J. Cong, H. Huang, C. Ma, B. Xiao, and P. Zhou, "A fully pipelined and dynamically composable architecture of CGRA," in *22th FCCM*, 2014.

[70] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[71] T. M. Conte and W. Mangione-Smith, "Determining cost-effective multiple issue processor designs," in *ICCD*, Oct 1993.

[72] T. M. Conte, K. N. P. Menezes, and S. W. Sathaye, "A technique to determine power-efficient, high-performance superscalar processors," in *HICSS*, 1995.

[73] V. Dadu, S. Liu, and T. Nowatzki, "Systematically understanding graph accelerator dimensions and the value of hardware flexibility," *IEEE Micro*, vol. 42, no. 4, pp. 87–96, 2022.

[74] V. Dadu and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *52nd MICRO*, 2019.

[75] V. Dadu and T. Nowatzki, "Taskstream: Accelerating task-parallel workloads by recovering program structure," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, 2022.

[76] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general-purpose acceleration: Finding structure in irregularity," *IEEE Micro*, vol. 40, no. 3, pp. 37–46, 2020.

[77] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang, "Graphh: A processing-in-memory architecture for large-scale graph processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 4, pp. 640–653, April 2019.

[78] G. Dai, Z. Zhu, T. Fu, C. Wei, B. Wang, X. Li, Y. Xie, H. Yang, and Y. Wang, "Dimmining: Pruning-efficient and parallel graph mining on near-memory-computing," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 130–145. [Online]. Available: https://doi.org/10.1145/3470496.3527388

[79] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[80] R. Dimond, O. Mencer, and W. Luk, "CUSTARD - a customisable threaded FPGA soft processor and tools," in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 1–6.

[81] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited vector extension with data streaming support," in *ISCA*, 2021.

[82] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache, "TC-CIM: Empowering tensor comprehensions for computing-in-memory," in *IMPACT 2020-10th International Workshop on Polyhedral Compilation Techniques*, 2020.

[83] P. Duarte, P. Tomas, and G. Falcao, "SCRATCH: An end-to-end application-aware soft-GPGPU architecture and trimming tool," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 165–177.

[84] A. E. Eichenberger, P. Wu, and K. O'brien, "Vectorization for simd architectures with alignment constraints," *Acm Sigplan Notices*, vol. 39, no. 6, pp. 82–93, 2004.

[85] R. Engelen, "Symbolic evaluation of chains of recurrences for loop optimization," 03 2000.

[86] Y. Fang, T. T. Hoang, M. Becchi, and A. A. Chien, "Fast support for unstructured data processing: The unified automata processor," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 533–545. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830809

[87] Y. Fang, C. Zou, A. Elmore, and A. Chien, “Udp: A programmable accelerator for extract-transform-load workloads and more,” in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[88] R. Fasthuber, F. Catthoor, P. Raghavan, and F. Naessens, *Energy-Efficient Communication Processors: Design and Implementation for Emerging Wireless Systems.* Springer Publishing Company, Incorporated, 2013.

[89] S. Feng, B. Hou, H. Jin, W. Lin, J. Shao, R. Lai, Z. Ye, L. Zheng, C. H. Yu, Y. Yu, and T. Chen, “Tensorir: An abstraction for automatic tensorized program optimization,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 804–817. [Online]. Available: https://doi.org/10.1145/3575693.3576933

[90] Y. Feng, M. Xie, Z. Tian, S. Wang, Y. Lu, and J. Shu, “Mobius: Fine tuning large-scale models on commodity gpu servers,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 489–501. [Online]. Available: https://doi.org/10.1145/3575693.3575703

[91] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[92] J. A. Fisher, P. Faraboschi, and G. Desoli, “Custom-fit processors: Letting applications define architectures,” in *MICRO*, 1996.

[93] R. Foundation, “Working draft of the proposed risc-v v vector extension,” https://github.com/riscv/riscv-v-spec.

[94] D. Fujiki, A. Subramaniyan, T. Zhang, Y. Zeng, R. Das, D. Blaauw, and S. Narayanasamy, “Genax: A genome sequencing accelerator,” in *Proceedings of the 45th Annual International Symposium on Computer Architecture.* IEEE Press, 2018, pp. 69–82.

[95] J. Gaisler and M. Isomäki, “Leon3 gr-xc3s-1500 template design,” *Copyright Gaisler Research*, pp. 1–153, 2006.

[96] Y. Gao, B. Zhang, X. Qi, and H. K.-H. So, “Dpacs: Hardware accelerated dynamic neural network pruning through algorithm-architecture co-design,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 237–251. [Online]. Available: https://doi.org/10.1145/3575693.3575728

[97] F. Garzia, W. Hussain, and J. Nurmi, “CREMA: A coarse-grain reconfigurable array with mapping adaptiveness,” in *2009 International Conference on Field Programmable Logic and Applications.* IEEE, 2009, pp. 708–712.

[98] C. Giannoula, N. Vijaykumar, N. Papadopoulou, V. Karakostas, I. Fernandez, J. Gómez-Luna, L. Orosa, N. Koziris, G. Goumas, and O. Mutlu, “Syncron: Efficient synchronization support for near-data-processing architectures,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE, 2021, pp. 263–276.

[99] A. Girault, B. Lee, and E. A. Lee, “Hierarchical finite state machines with multiple concurrency models,” *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 18, no. 6, pp. 742–760, 1999.

[100] G. Gobieski, A. O. Atli, K. Mai, B. Lucia, and N. Beckmann, “SNAFU: an ultra-low-power, energy-minimal cgra-generation framework and architecture,” in *ISCA*, 2021.

[101] G. Gobieski, S. Ghosh, M. Heule, T. Mowry, T. Nowatzki, N. Beckmann, and B. Lucia, “RipTide: a programmable, energy-minimal dataflow compiler and architecture,” in *MICRO-55: 55th Annual IEEE/ACM International Symposium on Microarchitecture*, 2022.

[102] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, “PipeRench: a coprocessor for streaming multimedia acceleration,” in *26th ISCA*, 1999.

[103] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, C. Leger, A. A. Lamb, J. Wong, H. Hoffman, D. Z. Maze, and S. Amarasinghe, “A Stream Compiler for Communication-Exposed Architectures,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 291–303.

[104] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, “DySER: unifying functionality and parallelism specialization for energy-efficient computing,” *IEEE Micro*, Sep. 2012.

[105] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, “Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 341–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2523721.2523767

[106] Z. Gu and R. Pellizzoni, “Optimizing parallel prem compilation over nested loop structures,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1249–1254. [Online]. Available: https://doi.org/10.1145/3489517.3530610

[107] L. Guo, Y. Chi, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, and J. Cong, "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 81–92.

[108] L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, Y. Chi, W. Qiao, A. Kaviani, Z. Zhang, and J. Cong, "RapidStream: parallel physical implementation of FPGA HLS designs," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022, pp. 1–12.

[109] B. Hagedorn, B. Fan, H. Chen, C. Cecka, M. Garland, and V. Grover, "Graphene: An ir for optimized tensor computations on gpus," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 302–313. [Online]. Available: https://doi.org/10.1145/3582016.3582018

[110] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "Expression: a language for architecture exploration through compiler/simulator retargetability," in *DATE*, March 1999.

[111] 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)*, Oct 2016, pp. 1–13.

[112] Q. Han, Y. Hu, F. Yu, H. Yang, B. Liu, P. Hu, R. Gong, Y. Wang, R. Wang, Z. Luan, and D. Qian, "Extremely low-bit convolution optimization for quantized neural network on modern computer architectures," in *ICPP '20: 49th International Conference on Parallel Processing - ICPP*, 2020.

[113] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[114] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array architectures," in *DAC*, Jan 2000.

[115] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*,

ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 674–687. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00062

[116] M. D. Hill and V. J. Reddi, "Accelerator-level parallelism," *Communications of the ACM*, vol. 64, no. 12, pp. 36–38, 2021.

[117] B. K. Holmer and A. M. Despain, "Viewing instruction set design as an optimization problem," in *24th MICRO*. ACM, 1991.

[118] C.-H. Hoy, V. Govindarajuz, T. Nowatzki, R. Nagaraju, Z. Marzec, P. Agarwal, C. Frericks, R. Cofell, and K. Sankaralingam, "Performance evaluation of a DySER FPGA prototype system spanning the compiler, microarchitecture, and hardware implementation," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2015, pp. 203–214.

[119] O. Hsu, M. Strange, R. Sharma, J. Won, K. Olukotun, J. S. Emer, M. A. Horowitz, and F. Kjølstad, "The sparse abstract machine," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 710–726. [Online]. Available: https://doi.org/10.1145/3582016.3582051

[120] I. J. Huang and A. M. Despain, "High level synthesis of pipelined instruction set processors and back-end compilers," in *DAC*, Jun 1992.

[121] T. Hussain, O. Palomar, O. Unsal, A. Cristal, E. Ayguadé, and M. Valero, "Advanced pattern based memory controller for fpga based hpc applications," in *2014 International Conference on High Performance Computing Simulation (HPCS)*, July 2014, pp. 287–294.

[122] W. Hussain, T. Ahonen, and J. Nurmi, "Effects of scaling a coarse-grain reconfigurable array on power and energy consumption," in *2012 International Symposium on System on Chip (SoC)*. IEEE, 2012, pp. 1–5.

[123] M. Huzaifa, R. Desai, S. Grayson, X. Jiang, Y. Jing, J. Lee, F. Lu, Y. Pang, J. Ravichandran, F. Sinclair, B. Tian, H. Yuan, J. Zhang, and S. V. Adve, "ILLIXR: enabling end-to-end extended reality research," in *IEEE International Symposium on Workload Characterization, IISWC 2021, Storrs, CT, USA, November 7-9, 2021*. IEEE, 2021, pp. 24–38.

[124] R. M. J. Hu, "Energy-aware mapping for tile-based noc architectures under performance constraints," in *ASP-DAC*, Jan 2003.

[125] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient journal," *CoRR*, vol. abs/1712.05877, 2017.

[126] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Are coarse-grained overlays ready for general purpose application acceleration on FPGAs?" in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech).* IEEE, 2016, pp. 586–593.

[127] A. Jain, S. Bhattacharya, M. Masuda, V. Sharma, and Y. Wang, "Efficient execution of quantized deep learning models: A compiler approach," *arXiv preprint arXiv:2006.10226*, 2020.

[128] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 228–241.

[129] O. Jin, Q. Xing, Y. Li, S. Deng, S. He, and G. Pan, "Mapping very large scale spiking neuron network to neuromorphic hardware," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 419–432. [Online]. Available: https://doi.org/10.1145/3582016.3582038

[130] H. Johansson *et al.*, "Polyphase decomposition of digital fractional-delay filters," *IEEE signal processing letters*, vol. 22, no. 8, pp. 1021–1025, 2015.

[131] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017.

[132] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–12.

[133] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu *et al.*, "Bluedbm: An appliance for big data analytics," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).* IEEE, 2015, pp. 1–13.

[134] S.-C. Kao, S. Subramanian, G. Agrawal, A. Yazdanbakhsh, and T. Krishna, “Flat: An optimized dataflow for mitigating attention bottlenecks,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 295–310. [Online]. Available: https://doi.org/10.1145/3575693.3575747

[135] I. Karageorgos, K. Sriram, J. Veselý, M. Wu, M. Powell, D. Borton, R. Manohar, and A. Bhattacharjee, “Hardware-software co-design for brain-computer interfaces,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 391–404.

[136] M. Karunaratne, D. Wijerathne, T. Mitra, and L.-S. Peh, “4D-CGRA: introducing branch dimension to spatio-temporal application mapping on CGRAs,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019.

[137] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel, “Rigel: An architecture and scalable programming interface for a 1000-core accelerator,” in *ISCA ’09*, pp. 140–151.

[138] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, “Ripple: Profile-guided instruction cache replacement for data center applications,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 734–747.

[139] J. Kim, S. Jiang, C. Torng, M. Wang, S. Srinath, B. Ilbeyi, K. Al-Hawaj, and C. Batten, “Using intra-core loop-task accelerators to improve the productivity and performance of task-based parallel programs,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 ’17. New York, NY, USA: ACM, 2017, pp. 759–773. [Online]. Available: http://doi.acm.org/10.1145/3123939.3136952

[140] Y. Kim, J. Lee, T. X. Mai, and Y. Paek, “Improving performance of nested loops on reconfigurable array processors,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 32, 2012.

[141] Y. Kim, R. N. Mahapatra, and K. Choi, “Design space exploration for efficient resource utilization in coarse-grained reconfigurable architecture,” *IEEE transactions on very large scale integration (VLSI) systems*, vol. 18, no. 10, pp. 1471–1482, 2009.

[142] J. Kingyens and J. G. Steffan, “The potential for a GPU-like overlay architecture for FPGAs,” *International Journal of Reconfigurable Computing*, vol. 2011, 2011.

[143] M. A. Kinsy, M. Pellauer, and S. Devadas, “Heracles: Fully synthesizable parameterized MIPS-based multicore system,” in *2011 21st International Conference on Field Programmable Logic and Applications*, 2011, pp. 356–362.

[144] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *MICRO*, 2013.

[145] D. Koeplinger, C. Delimitrou, R. Prabhakar, C. Kozyrakis, Y. Zhang, and K. Olukotun, "Automatic generation of efficient accelerators for reconfigurable hardware," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 115–127. [Online]. Available: https://doi.org/10.1109/ISCA.2016.20

[146] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *PLDI*, 2018.

[147] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 296–311. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192379

[148] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet SIMD code generation," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 127–138.

[149] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 991–1010. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/roscoe

[150] K. Koul, J. Melchert, K. Sreedhar, L. Truong, G. Nyengele, K. Zhang, Q. Liu, J. Setter, P.-H. Chen, Y. Mei, M. Strange, R. Daly, C. Donovick, A. Carsello, T. Kong, K. Feng, D. Huff, A. Nayak, R. Setaluri, J. Thomas, N. Bhagdikar, D. Durst, Z. Myers, N. Tsiskaridze, S. Richardson, R. Bahr, K. Fatahalian, P. Hanrahan, C. Barrett, M. Horowitz, C. Torng, F. Kjolstad, and P. Raina, "Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers," *ACM Trans. Embed. Comput. Syst.*, apr 2022.

[151] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic, "The vector-thread architecture," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 52–. [Online]. Available: http://dl.acm.org/citation.cfm?id=998680.1006736

[152] Y. E. Krasteva, F. Criado, E. d. l. Torre, and T. Riesgo, "A fast emulation-based noc prototyping framework," in *ReConFig*, Dec 2008.

[153] C. Kumar H B, P. Ravi, G. Modi, and N. Kapre, "120-Core MicroAptiv MIPS overlay for the Terasic DE5-NET FPGA board," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 141–146.

[154] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," *SIGPLAN Not.*, vol. 53, no. 2, pp. 461–475, Mar. 2018.

[155] C. E. Laforest and J. H. Anderson, "Microarchitectural comparison of the MXP and Octavo soft-processor FPGA overlays," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 10, no. 3, May 2017.

[156] C. E. LaForest and J. G. Steffan, "OCTAVO: An FPGA-centric processor family," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 219–228.

[157] J. Landgraf, T. Yang, W. Lin, C. J. Rossbach, and E. Schkufza, "Compiler-driven fpga virtualization with synergy," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 818–831. [Online]. Available: https://doi.org/10.1145/3445814.3446755

[158] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *Acm Sigplan Notices*, vol. 35, no. 5, pp. 145–156, 2000.

[159] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 7, no. 9, pp. 1235–1245, September 1987.

[160] J. Lee, S. Seo, H. Lee, and H. U. Sim, "Flattening-based mapping of imperfect loop nests for cgras?" in *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2014, pp. 1–10.

[161] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović, "Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000080

[162] X. Li and D. L. Maskell, "Time-multiplexed FPGA overlay architectures: A survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 5, jul 2019.

[163] Y.-C. Lin, B. Zhang, and V. Prasanna, "Hp-gnn: Generating high throughput gnn training implementation on cpu-fpga heterogeneous platform," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 123–133. [Online]. Available: https://doi.org/10.1145/3490422.3502359

[164] C. Liu, H.-C. Ng, and H. K.-H. So, "Automatic nested loop acceleration on FPGAs using soft CGRA overlay," *arXiv preprint arXiv:1509.00042*, 2015.

[165] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 530–543. [Online]. Available: https://doi.org/10.1145/3575693.3575707

[166] S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Wang, L. Guo, J. Liu, M. Zhulin, R. Mani, L. Zhang, J. Cong, and T. Nowatzki, "Overgen: Improving fpga usability through domain-specific overlay generation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 35–56.

[167] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19).* Renton, WA: USENIX Association, Jul. 2019, pp. 1025–1040. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/liu-yizhi

[168] R. Ma, J.-C. Hsu, T. Tan, E. Nurvitadhi, D. Sheffield, R. Pelt, M. Langhammer, J. Sim, A. Dasu, and D. Chiou, "Specializing FGPU for persistent deep learning," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 326–333.

[169] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A compiler for vectorizing encrypted arithmetic circuits," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 118–133. [Online]. Available: https://doi.org/10.1145/3582016.3582057

[170] M. Mandava, P. Reckamp, and D. Chen, "Nimblock: Scheduling for fine-grained fpga sharing through virtualization," in *Proceedings of the 50th International Symposium on Computer Architecture*, ser. ISCA '23, 2023.

[171] L. McMurchie and C. Ebeling, "PathFinder: a negotiation-based performance-driven router for fpgas," in *3rd FPGA*, Feb 1995.

[172] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, “Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling,” *IEE Proceedings - Computers and Digital Techniques*, vol. 150, no. 5, pp. 255–61–, Sept 2003.

[173] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, “ADRES: an architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix,” in *FPL*, 2003.

[174] J. Melchert, K. Feng, C. Donovick, R. Daly, C. W. Barrett, M. Horowitz, P. Hanrahan, and P. Raina, “Automated design space exploration of CGRA processing element architectures using frequent subgraph analysis,” *CoRR*, 2021.

[175] J. Melchert, K. Feng, C. Donovick, R. Daly, R. Sharma, C. Barrett, M. A. Horowitz, P. Hanrahan, and P. Raina, “Apex: A framework for automated processing element design space exploration using frequent subgraph analysis,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 33–45. [Online]. Available: https://doi.org/10.1145/3582016.3582070

[176] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, “Mixed precision training,” *CoRR*, vol. abs/1710.03740, 2017.

[177] F. Mintzer, “On half-band, third-band, and nth-band fir filters and their design,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 30, no. 5, pp. 734–738, Oct 1982.

[178] E. Mirsky, A. DeHon *et al.*, “MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources.” in *FCCM*, vol. 96, 1996, pp. 17–19.

[179] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, S. C. Goldstein, and M. Budiu, “Tartan: Evaluating spatial computation for whole program execution,” in *12th ASPLOS*, 2006.

[180] T. Miyamori and K. Olukotun, “REMARC (abstract): Reconfigurable multimedia array coprocessor,” in *6th FPGA*, 1998.

[181] MLIR, “Multi-level IR compiler framework,” https://mlir.llvm.org.

[182] R. Moussali, N. Ghanem, and M. A. R. Saghir, “Supporting multithreading in configurable soft processor cores,” in *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 155–159.

[183] F. Muñoz Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, "Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 252–265. [Online]. Available: https://doi.org/10.1145/3582016.3582069

[184] J. M. Mulder, R. J. Portier, A. Srivastava, and R. in't Velt, "An architecture framework for application-specific and scalable architectures," in *16th ISCA*, 1989.

[185] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," in *DATE*, vol. 2, Feb 2004.

[186] S. Murali, G. De Micheli, G. De Micheli, and G. De Micheli, "SUNMAP: a tool for automatic topology selection and generation for nocs," in *41st DAC*. ACM, 2004.

[187] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical design space exploration," 2018.

[188] Q. M. Nguyen and D. Sanchez, "Fifer: Practical acceleration of irregular applications on reconfigurable architectures," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, 2021.

[189] C. Nicol, "A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing," *WaveComputing WhitePaper*, 2017.

[190] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 393–407.

[191] K. Niu and J. H. Anderson, "Compact area and performance modelling for cgra architecture evaluation," in *FPT*, Dec 2018.

[192] T. Nowatzki, N. Ardalani, K. Sankaralingam, and J. Weng, "Hybrid optimization/heuristic instruction scheduling for programmable accelerator codesign," in *27th PACT*, 2018.

[193] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in *44th ISCA*, 2017.

[194] D. Nuzman and R. Henderson, "Multi-platform auto-vectorization," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. USA: IEEE Computer Society, 2006, p. 281–294. [Online]. Available: https://doi.org/10.1109/CGO.2006.25

[195] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for SIMD," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, M. I. Schwartzbach and T. Ball, Eds. ACM, 2006, pp. 132–143. [Online]. Available: https://doi.org/10.1145/1133981.1133997

[196] T. O. Odemuyiwa, H. Asghari-Moghaddam, M. Pellauer, K. Hegde, P.-A. Tsai, N. C. Crago, A. Jaleel, J. D. Owens, E. Solomonik, J. S. Emer, and C. W. Fletcher, "Accelerating sparse data orchestration via dynamic reflexive tiling," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 18–32. [Online]. Available: https://doi.org/10.1145/3582016.3582064

[197] S. Önder and R. Gupta, "Automatic generation of microarchitecture simulators," in *ICCL*, 1998.

[198] "Openrisc project, http://opencores.org/project,or1k."

[199] M. K. Papamichael and J. C. Hoe, "Connect: re-examining conventional wisdom for designing nocs in the context of FPGAs," in *FPGA*, 2012.

[200] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer, "Triggered Instructions: a control paradigm for spatially-programmed architectures," in *40th ISCA*, 2013.

[201] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: an accelerator for compressed-sparse convolutional neural networks," in *44th ISCA*, 2017.

[202] D. Park, Y. Xiao, N. Magnezi, and A. DeHon, "Case for fast FPGA compilation using partial reconfiguration," in *28th International Conference on Field Programmable Logic and Applications, FPL 2018, Dublin, Ireland, August 27-31, 2018.* IEEE Computer Society, 2018, pp. 235–238.

[203] Y. Park, H. Park, and S. Mahlke, "Cgra express: Accelerating execution using dynamic operation fusion," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 271–280. [Online]. Available: http://doi.acm.org/10.1145/1629395.1629433

[204] K. Paul, C. Dash, and M. S. Moghaddam, "reMORPH: a runtime reconfigurable architecture," in *2012 15th Euromicro Conference on Digital System Design.* IEEE, 2012, pp. 26–33.

[205] A. Pedram, A. Gerstlauer, and R. van de Geijn, "Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator," *IEEE Transactions on Computers*, no. 1, pp. 1–1, 2014.

[206] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "Lisa-machine description language for cycle-accurate models of programmable dsp architectures," in *DAC*, 1999.

[207] P. M. Phothilimthana, A. S. Elliott, A. Wang, A. Jangda, B. Hagedorn, H. Barthels, S. J. Kaufman, V. Grover, E. Torlak, and R. Bodik, "Swizzle inventor: data movement synthesis for GPU kernels," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 65–78.

[208] A. Pinto, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Efficient synthesis of networks on chip," in *21st ICCD*, 2003.

[209] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 651–665. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872415

[210] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, "Plasticine: A reconfigurable architecture for parallel paterns," in *44th ISCA*, 2017.

[211] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 3, p. 1–25, Aug 2017. [Online]. Available: http://dx.doi.org/10.1145/3107953

[212] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, "Fans: Fpga-accelerated near-storage sorting," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 106–114.

[213] W. Qin, S. Rajagopalan, M. Vachharajani, H. Wang, X. Zhu, D. I. August, K. Keutzer, S. Malik, and L.-S. Peh, "Design tools for application specific embedded processors," in *Proceedings of the Second International Workshop on Embedded Software, Lecture Notes in Computer Science*, 2002.

[214] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. P. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA,*

June 16-19, 2013, H. Boehm and C. Flanagan, Eds. ACM, 2013, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[215] R. Rashid, J. G. Steffan, and V. Betz, "Comparing performance, productivity and scalability of the TILT overlay processor to OpenCL HLS," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 20–27.

[216] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks, "MachSuite: benchmarks for accelerator design and customized architectures," in *IISWC*, Oct 2014.

[217] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Y. 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)*, June 2016, pp. 267–278.

[218] T. J. Repetti, J. a. P. Cerqueira, M. A. Kim, and M. Seok, "Pipelining a triggered processing element," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017, pp. 96–108. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124551

[219] S. Rivoire, R. Schultz, T. Okuda, and C. Kozyrakis, "Vector lane threading," in *2006 International Conference on Parallel Processing (ICPP'06)*, Aug 2006, pp. 55–64.

[220] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagunas, P. R. Mattson, and J. D. Owens, "A bandwidth-efficient architecture for media processing," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 31. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 3–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=290940.290946

[221] A. Roelke and M. R. Stan, "RISC5: Implementing the RISC-V ISA in gem5," 2017.

[222] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new IR for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 58–68. [Online]. Available: https://doi.org/10.1145/3211346.3211348

[223] I. Rosen, D. Nuzman, and A. Zaks, "Loop-aware SLP in GCC," pp. 131–142, 01 2007.

[224] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, S. Nadathur, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph lowering compiler techniques for neural networks," *CoRR*, vol. abs/1805.00907, 2018. [Online]. Available: https://arxiv.org/abs/1805.00907

[225] N. Samardzic, W. Qiao, V. Aggarwal, M.-C. F. Chang, and J. Cong, "Bonsai: High-performance adaptive merge tree sorting," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 282–294.

[226] K. Sankaralingam, T. Nowatzki, V. Gangadhar, P. Shah, M. Davies, W. Galliher, Z. Guo, J. Khare, D. Vijay, P. Palamuttam, M. Punde, A. Tan, V. Thiruvengadam, R. Wang, and S. Xu, "The Mozart reuse exposed dataflow processor for AI and beyond: Industrial product," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22, 2022, p. 978–992.

[227] G. Schelle, J. Collins, E. Schuchman, P. Wang, X. Zou, G. Chinya, R. Plate, T. Mattner, F. Olbrich, P. Hammarlund, R. Singhal, J. Brayton, S. Steibl, and H. Wang, "Intel Nehalem processor core made FPGA synthesizable," in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 3–12.

[228] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 271–286.

[229] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "Stream semantic registers: A lightweight risc-v isa extension achieving full compute utilization in single-issue cores," *IEEE Trans. Comput.*, 2021.

[230] A. Severance and G. G. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013, pp. 1–10.

[231] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 14–26.

[232] A. Sharifian, R. Hojabr, N. Rahimi, S. Liu, A. Guha, T. Nowatzki, and A. Shriraman, "μir -an intermediate representation for transforming and optimizing the microarchitecture of application accelerators," in *52nd MICRO*, 2019.

[233] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A coarse grained paradigm for FPGAs," in *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[234] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Trans. Comput.*, vol. 49, no. 5, pp. 465–481, May 2000.

[235] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley, "Dataflow predication," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 89–102.

[236] J. E. Smith, "Decoupled access/execute computer architectures," in *9th ISCA*, 1982.

[237] A. Sohrabizadeh, Y. Bai, Y. Sun, and J. Cong, "Automated accelerator optimization aided by graph neural networks," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 55–60. [Online]. Available: https://doi.org/10.1145/3489517.3530409

[238] A. Sohrabizadeh, J. Wang, and J. Cong, "End-to-end optimization of deep learning applications," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020, pp. 133–139.

[239] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "AutoDSE: Enabling software programmers to design efficient fpga accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 27, no. 4, feb 2022.

[240] V. G. Somashekaracharya G. Bhaskaracharya, Julien Demouth, "Automatic kernel generation for Volta tensor cores," *arXiv preprint arXiv:2006.12645*, 2020. [Online]. Available: https://arxiv.org/abs/2006.12645

[241] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 531–543.

[242] D. Suh, K. Kwon, S. Kim, S. Ryu, and J. Kim, "Design space exploration and implementation of a high performance and low area coarse grained reconfigurable processor," in *CFP*, Dec 2012.

[243] T. Swamy, A. Zulfiqar, L. Nardi, M. Shahbaz, and K. Olukotun, "Homunculus: Auto-generating efficient data-plane ml pipelines for datacenter networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 329–342. [Online]. Available: https://doi.org/10.1145/3582016.3582022

[244] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *36th MICRO*, ser. MICRO 36, 2003.

[245] C. Tan, T. Geng, C. Xie, N. B. Agostini, J. Li, A. Li, K. Barker, and A. Tumeo, "DynPaC: coarse-grained, dynamic, and partially reconfigurable array for streaming applications," in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, 2021.

[246] C. Tan, T. Tambe, J. J. Zhang, B. Fang, T. Geng, G.-Y. Wei, D. Brooks, A. Tumeo, G. Gopalakrishnan, and A. Li, "ASAP: automatic synthesis of area-efficient and precision-aware CGRAs," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ser. ICS '22, 2022.

[247] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *DATE*, 2021.

[248] C. Tan, C. Xie, A. Li, K. J. Barker, and A. Tumeo, "Aurora: Automated refinement of coarse-grained reconfigurable accelerators," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE).* IEEE, 2021, pp. 1388–1393.

[249] S. Tavarageri, A. Heinecke, S. Avancha, G. Goyal, R. Upadrasta, and B. Kaul, "PolyDL: Polyhedral optimizations for creation of high performance DL primitives," *arXiv preprint arXiv:2006.02230*, 2020. [Online]. Available: https://arxiv.org/abs/2006.02230

[250] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *International Conference on Compiler Construction.* Springer, 2002, pp. 179–196.

[251] F. Tombs, A. Mellat, and N. Kapre, "Mocarabe: High-performance time-multiplexed overlays for FPGAs," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 115–123.

[252] Y. Turakhia, G. Bejerano, and W. J. Dally, "Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2018, pp. 199–213.

[253] Y. Turakhia, S. D. Goenka, G. Bejerano, and W. J. Dally, "Darwin-wga: A co-processor provides increased sensitivity in whole genome alignments with high speedup," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2019, pp. 359–372.

[254] J. Turley, "Tensilica CPU Bends to Designers's Will," *Microprocessor Report*, vol. 13, no. 4, March 1999.

[255] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August, "Microarchitectural exploration with Liberty," in *35th MICRO*, 2002.

[256] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions,” *arXiv preprint arXiv:1802.04730*, 2018.

[257] L. Vega, J. McMahan, A. Sampson, D. Grossman, and L. Ceze, “Reticle: A virtual machine for programming modern FPGAs,” in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 756–771.

[258] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, “Scaledeep: A scalable compute architecture for learning and evaluating deep networks,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017, pp. 13–26. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080244

[259] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, “Polyhedral parallel code generation for cuda,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400713

[260] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, “Schedule trees,” in *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*, 2014.

[261] M. Vilim, A. Rucker, Y. Zhang, S. Liu, and K. Olukotun, “Gorgon: Accelerating machine learning from relational data,” in *ISCA*, 2020.

[262] D. Voitsechov and Y. Etsion, “Control flow coalescing on a hybrid dataflow/von neumann gpgpu,” in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 216–227.

[263] D. Voitsechov and Y. Etsion, “Single-graph multiple flows: Energy efficient design alternative for gpgpus,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 205–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665703

[264] D. Voitsechov and Y. Etsion, “Inter-thread communication in multithreaded, reconfigurable coarse-grain arrays,” *arXiv preprint arXiv:1801.05178*, 2018.

[265] Wai Hong Ho and T. M. Pinkston, “A methodology for designing efficient on-chip interconnects on well-behaved communication patterns,” in *9th HPCA*, Feb 2003.

[266] M. J. Walker and J. H. Anderson, "Generic connectivity-based CGRA mapping via integer linear programming," in *27th FCCM*, 2019.

[267] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 97–110.

[268] J. Wang, L. Guo, and J. Cong, "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 93–104.

[269] S. Wang, M. Zhang, K. Yang, K. Chen, S. Ma, J. Jiang, and Y. Wu, "Noswalker: A decoupled architecture for out-of-core random walk processing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 466–482. [Online]. Available: https://doi.org/10.1145/3582016.3582025

[270] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 736–749.

[271] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019, pp. 736–749. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322229

[272] G. Weisz and J. C. Hoe, "Coram++: Supporting data-structure-specific memory interfaces for fpga computing," in *25th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2015, pp. 1–8.

[273] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

[274] J. Weng, A. Jain, J. Wang, L. Wang, Y. Wang, and T. Nowatzki, "UNIT: unifying tensorized instruction compilation," *CoRR*, vol. abs/2101.08458, 2021. [Online]. Available: https://arxiv.org/abs/2101.08458

[275] J. Weng, S. Liu, D. Kupsh, and T. Nowatzki, "Unifying spatial accelerator compilation with idiomatic and modular transformations," *IEEE Micro*, pp. 1–12, 2022.

[276] J. Weng, S. Liu, Z. Wang, V. Dadu, and T. Nowatzki, "A hybrid systolic-dataflow architecture for inductive matrix algorithms," in *HPCA*, 2019.

[277] M. Willsey, V. T. Lee, A. Cheung, R. Bodík, and L. Ceze, "Iterative search for reconfigurable accelerator blocks with a compiler in the loop," *IEEE TCAD*, vol. 38, no. 3, pp. 407–418, 2018.

[278] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," in *19th ASPLOS*, 2014.

[279] Y. Xiao, E. Micallef, A. Butt, M. Hofmann, M. Alston, M. Goldsmith, A. Merczynski-Hait, and A. DeHon, "PLD: fast FPGA compilation to make reconfigurable acceleration compatible with modern incremental refinement software development," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 933–945.

[280] C. Xu, C. Kjellqvist, and L. W. Wills, "Sns's not a synthesizer: A deep-learning-based synthesis predictor," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 847–859. [Online]. Available: https://doi.org/10.1145/3470496.3527444

[281] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 634–643.

[282] S. Yang, M. Zhang, W. Dong, and D. Li, "Betty: Enabling large-scale gnn training with batch-level graph partitioning," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 103–117. [Online]. Available: https://doi.org/10.1145/3575693.3575725

[283] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture.* IEEE Press, 2018, pp. 650–661.

[284] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of FPGA-based soft processors," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 202–212.

[285] J. Yin, P. Zhou, S. S. Sapatnekar, and A. Zhai, "Energy-efficient time-division multiplexed hybrid-switched noc for heterogeneous multicore systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 293–303.

[286] S. Yin, X. Lin, L. Liu, and S. Wei, “Exploiting parallelism of imperfect nested loops on coarse-grained reconfigurable architectures,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3199–3213, Nov 2016.

[287] X. Yin, Z. Zhao, and R. Gupta, “Glign: Taming misaligned graph traversals in concurrent graph processing,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2022, p. 78–92. [Online]. Available: https://doi.org/10.1145/3567955.3567963

[288] Y. Zhai, Y. Zhang, S. Liu, X. Chu, J. Peng, J. Ji, and Y. Zhang, “Tlp: A deep learning-based cost model for tensor program tuning,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 833–845. [Online]. Available: https://doi.org/10.1145/3575693.3575737

[289] D. Zhang, S. Huda, E. Songhori, K. Prabhu, Q. Le, A. Goldie, and A. Mirhoseini, “A full-stack search technique for domain optimized deep learning accelerators,” in *ASPLOS*, 2022.

[290] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, “Graphp: Reducing communication for pim-based graph processing with efficient data partition,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 544–557.

[291] Y. Zhang, N. Zhang, T. Zhao, M. Vilim, M. Shahbaz, and K. Olukotun, “Sara: Scaling a reconfigurable dataflow accelerator,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1041–1054.

[292] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, “Pipezk: Accelerating zero-knowledge proof with a pipelined architecture,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 416–428.

[293] Y. Zhang, P.-A. Tsai, and H.-W. Tseng, “Simd2: A generalized matrix instruction set for accelerating tensor computation beyond gemm,” in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 552–566. [Online]. Available: https://doi.org/10.1145/3470496.3527411

[294] Z. Zhang, Y. Ou, Y. Liu, C. Wang, Y. Zhou, X. Wang, Y. Zhang, Y. Ouyang, J. Shan, Y. Wang, J. Xue, H. Cui, and X. Feng, “Occamy: Elastically sharing a simd co-processor across multiple cpu cores,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming*

Languages and Operating Systems, Volume 3, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 483–497. [Online]. Available: https://doi.org/10.1145/3582016.3582046

[295] R. Zhao, "Wls design of centro-symmetric 2-d fir filters using matrix iterative algorithm," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, July 2015, pp. 34–38.

[296] T. Zhao, A. Rucker, and K. Olukotun, "Sigma: Compiling einstein summations to locality-aware dataflow," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 718–732. [Online]. Available: https://doi.org/10.1145/3575693.3575694

[297] H. Zheng, K. Wang, and A. Louri, "Adapt-NoC: A flexible network-on-chip design for heterogeneous manycore architectures," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 723–735.

[298] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor: Generating High-Performance tensor programs for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 863–879. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zheng

[299] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 559–578. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[300] Y. Zhou, J. Leng, Y. Song, S. Lu, M. Wang, C. Li, M. Guo, W. Shen, Y. Li, W. Lin, X. Liu, and H. Wu, "Ugrapher: High-performance graph operator computation via unified abstraction for graph neural networks," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 878–891. [Online]. Available: https://doi.org/10.1145/3575693.3575723

[301] J. Zhuang, J. Lau, H. Ye, Z. Yang, Y. Du, J. Lo, K. Denolf, S. Neuendorffer, A. Jones, J. Hu, D. Chen, J. Cong, and P. Zhou, "Charm: Composing heterogeneous accelerators for matrix multiply on versal acap architecture," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 153–164. [Online]. Available: https://doi.org/10.1145/3543622.3573210

[302] J. Zuckerman, D. Giri, J. Kwon, P. Mantovani, and L. P. Carloni, “Cohmeleon: Learning-based orchestration of accelerator coherence in heterogeneous socs,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 350–365.