DECOUPLING ALGORITHM FROM HARDWARE CUSTOMIZATIONS FOR SOFTWARE-DEFINED RECONFIGURABLE COMPUTING

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yi-Hsiang Lai

May 2022

DECOUPLING ALGORITHM FROM HARDWARE CUSTOMIZATIONS FOR SOFTWARE-DEFINED RECONFIGURABLE COMPUTING

Yi-Hsiang Lai

Cornell University, May 2022

With the pursuit of improving compute performance under strict power constraints, there is an increasing need for deploying applications to heterogeneous hardware architectures with spatial accelerators such as FPGAs. However, although these heterogeneous computing platforms are becoming widely available, they are very difficult to program especially with FPGAs. As a result, the use of such platforms has been limited to a small subset of programmers with specialized hardware knowledge. In this dissertation, we first provide a taxonomy of the essential techniques for building a high-performance FPGA accelerator, which requires customizations of the compute engines, memory hierarchy, and data representations. We also summarize a rich spectrum of work on programming abstractions and optimizing compilers that provide different trade-offs between performance and productivity.

Next we present SuSy, a programming framework composed of a domain-specific language (DSL) and a compilation flow that enables programmers to productively build high-performance systolic arrays on FPGAs. With SuSy, programmers express the design functionality in the form of uniform recurrence equations (UREs). The URE description in SuSy is followed by a set of decoupled spatial mapping primitives that specify how to map the equations to a spatial architecture. More concretely, programmers can apply space-time transformations and several other memory and I/O optimizations to build a highly

efficient systolic architecture productively.

After that, we present HeteroCL, an open-source programming infrastructure composed of a Python-based domain-specific language and an FPGA-targeted compilation flow. Similar to SuSy, HeteroCL cleanly decouples algorithm specifications from three important types of hardware customization in compute, data types, and memory architectures. In addition, HeteroCL produces highly efficient hardware implementations for a variety of popular workloads by targeting spatial architecture templates such as systolic arrays and stencil with dataflow architectures.

Finally, we introduce DrTrace, a trace-based online profiling technique that enables automated validation and recommendation for application-specific data reuse. Unlike existing work that leverages static analysis, our proposed technique can infer stencil operations from programs with data-dependent memory accesses. Moreover, by integrating the proposed profiling technique with HeteroCL, the stencil operations can be mapped to efficient hardware such as dataflow pipelines with line buffers.

BIOGRAPHICAL SKETCH

Yi-Hsiang Lai received his bachelor's and master's degree in Electrical Engineering from National Taiwan University, Taipei, Taiwan in 2013 and 2015, respectively. He then joined the School of Electrical and Computer Engineering (ECE) at Cornell University as a Ph.D. student in 2016. Since then, Yi-Hsiang studied under the supervision of Prof. Zhiru Zhang at the Computer Systems Laboratory, where he passed his Ph.D. candidacy exam and received a Master of Science degree in ECE in April 2020. During his graduate study, Yi-Hsiang has worked on a variety of research areas, including high-level synthesis, deep learning, compiler, and programming language. He interned at Intel and Amazon in summer 2019 and summer 2020, respectively.

For everyone who has faith in me.

ACKNOWLEDGEMENTS

I sincerely appreciate the support from my family, my friends, my mentors, and all other people who have helped and encouraged me during my graduate study. This dissertation would not have been possible without their support.

First and foremost, I would like to thank my advisor, Prof. Zhiru Zhang, for his support and supervision during my Ph.D. Without Zhiru, I could not achieve what I have achieved. In the past six years, Zhiru advised me patiently and consistently provided precious suggestions to my research. In addition to his advice on my research, I learned a lot from Zhiru such as work-life balance, how to interact and collaborate with others, and being a leader.

I would also like to thank my dissertation committee members: Prof. Adrian Sampson and Prof. G. Edward Suh. I thank them for detailed suggestions on my dissertation and concrete feedback on my research. Adrian taught me a lot from the aspect of programming language and compiler. Ed gave me inspirational ideas on how I may proceed with my research.

I am grateful to all members of the Zhang research group and the CSL community for building such a lively and friendly environment for my research and daily life. Specifically, I really appreciate the help from the Zhang group alumni: Dr. Steve Haihang Dai, Dr. Gai Liu, Dr. Ritchie Zhao, Dr. Nitish Srivastava, Dr. Yuan Zhou, Dr. Zhenghong Jiang, and Prof. Cunxi Yu. They provided my valuable suggestions on my research and were amazing mentors for both academics and life. I acknowledge all the hard work from my collaborators within and outside of Cornell. Especially, working with Dr. Hongbo Rong, Dr. Cody Hao Yu, Dr. Jie Wang, Dr. Yuze Chi, Shaojie Xiang, Yuwei Hu, Niansong Zhang, Hongzheng Cheng, and Dr. Debjit Pal was a very pleasant experience. Without them, I could not proceed my research so smoothly. I would also like to thank all

my friends no matter where they are for bringing me unforgettable experiences.

Finally, I would like to express my deepest gratitude to my family members, Shu-Fang Wang, Chi-Chung Lai, and Hsin-Yu Lai, for making me become who I am today and always being supportive throughout my graduate study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

FPGA-based accelerator design primarily concerns performance and energy efficiency. An FPGA programmer has the full freedom to (1) create deep custom pipelines that are composed of simple (often low-bitwidth) compute units instead of full-blown ALUs, (2) construct highly parallel and distributed control logic and on-chip storage, and (3) schedule dataflows in an explicit way to minimize off-chip memory accesses without using caches. This is in stark contrast with programming microprocessors (i.e., CPUs) and general-purpose graphic processing units (i.e., GPUs), where the underlying hardware architecture (instruction pipeline, memory hierarchy, etc.) is fixed, and the control flow of a software program drives the instruction-based execution on the hardware. In other words, FPGAs can be reconfigured/customized for a specific application or a domain of applications to exploit its massive fine-grained parallelism and on-chip bandwidth. This often leads to a higher compute throughput, lower energy consumption, and a more predictable latency, when compared to CPUs and GPUs.

Such great potential and flexibility do come at a substantial cost — the very low productivity of programming FPGAs to achieve high performance in the real world. Even for seemingly simple kernels (e.g., matrix matrix multiply, convolution, sparse matrix vector multiply), it is not uncommon for expert FPGA programmers to spend several months, or even more than one year, to build an accelerator that is both functional and performant on an actual device [175].

In a sense, the extreme flexibility of fine-grained programmable logic on an FPGA is both its greatest asset and its biggest liability. The end result is that

FPGA-based acceleration is one of the most promising approaches to solving challenging problems across many domains in principle, but is within reach for only a few in practice, i.e., large enterprises that can afford teams of hardware and systems experts for high-value applications [22, 72].

1.1 Challenges in Programming FPGAs

First, it requires a paradigm shift of thinking — most programmers are used to von Neumann machines and they tend to think sequentially, with parallelization added later as optimizations (at the level of threads, loops, etc.). However, an FPGA is a spatial architecture that features a massive amount of compute units and memory elements such as look-up tables (LUTs), DSP slices, registers, block RAMs, and more recent ultra RAMs and network-on-chip (NoC). These hardware resources are distributed over the fabric (usually 2-dimensional) and run concurrently. Therefore, programmers have to think spatially and parallel in the first place for FPGA-based acceleration. Unlike CPUs, FPGAs typically do not have a predefined hardware cache hierarchy. In order to keep feeding data at a sufficient rate to the many parallel compute engines, programmers need to build user-managed buffers to maximally utilize both off- and on-chip memory bandwidth. This further adds to the programming complexity.

Second, conventional FPGA design tools mainly target hardware design experts instead of software programmers. It takes significant effort to manually create and optimize accelerator architectures using the traditional register-transfer-level (RTL) methodology. One must wrestle with low-level hardware description language (HDL) descriptions and computer-aided design (CAD) tools to implement a rich set of hardware customizations such as fixed-point

arithmetic, pipelining, banked memories, and double buffering. Even worse, synthesizing an RTL design to a bitstream usually takes hours, or even days. This lengthy compile cycle makes design space exploration (DSE) on FPGAs prohibitively expensive.

Third, FPGA designs have poor debuggability. For instance, when a synthesized design runs into a deadlock on FPGAs, there is no easy way to stop the execution with a breakpoint and retrieve a snapshot of the states of the design. Usually, only CPU-based hardware emulation and cycle-accurate RTL simulation may help. The former does not model some important details of a real FPGA accelerator and may behave inconsistently with the actual on-device execution. The latter is too slow for a complex design since the simulation is running at a very low level of design abstraction. Moreover, it is difficult to map the signals in the final netlist back to variables in the original design, as the variables are often mangled during RTL synthesis and technology mapping.

In short, it is an enormous challenge to achieve both high performance and high productivity for FPGA programming. While similar productivity-performance tension also exists for CPUs and GPUs, the problem is remarkably worse on FPGAs. As a result, *there is a dire need for new compilation/synthesis techniques and programming frameworks to enable productive design, exploration, generation, and debugging of FPGA-based accelerators based on high-level languages.* In the past 10-15 years, numerous research efforts have attempted to address this grand challenge of software-defined FPGA acceleration, and many exciting progresses have been made in both academia and industry.

1.2 Decoupling Algorithm from Hardware Customizations

Recent years have seen promising development on high-level synthesis (HLS) for FPGAs [47]. This is evidenced by the wide availability of commercial C/OpenCL-based HLS compilers such as Xilinx Vivado/Vitis HLS, Intel SDK for OpenCL, and Microchip LegUp HLS. However, programming high-performance FPGA applications with HLS tools requires a deep understanding of hardware details and is entirely different from traditional software programming. In particular, current programming models for HLS entangle algorithm specifications with hardware customization techniques. This approach has several drawbacks: (1) In order to achieve good quality-of-results (QoRs), programmers are required to use various vendor-specific data types and pragmas/directives [222], rendering FPGA-targeted applications even less flexible and portable; (2) Existing HLS programming models cannot cleanly capture the interdependence among different hardware optimization techniques, thus weakening the support of user-guided or automatic design space exploration. For example, there is no easy way to inform the HLS tool that the shape of an on-chip buffer (e.g., depth and number of banks) directly depends on the degree of parallelization; (3) HLS users need to extensively restructure the source program to guide the tool to realize specialized architectures such as data reuse buffers and systolic arrays, which are nontrivial to describe with imperative code in C/C++.

There exists an active body of work attempting to further democratize accelerator programming by using domain-specific languages (DSLs) to simplify the development and optimization of applications in certain fields. For example, Halide [173] and Spark [215] are widely used in image processing and

big data analytics, respectively. Another relevant example is TVM, which is a Python-based DSL for high-performance deep learning applications [25]. Similar to Halide, TVM separates the algorithm from temporal schedule optimization (e.g., loop tiling and reordering), which significantly improves code portability across different CPU and GPU architectures.

1.3 Trade-offs Between Automation and Manual Effort

Programmers and tools have to "collaborate", to carry out a set of optimizations to customize the target hardware for a given application. Performance-critical customizations must be implemented, no matter by the programmers or the tools. How much automation should be expected from the tools, and how much control should be given to the programmer, largely determine the design of the programming abstraction and directly impact the productivity of the programming process. For instance, many popular workloads from image/video processing and machine learning domains can be realized using spatial architectures such as systolic arrays. It is not uncommon for HLS experts in the industry to spend several months on building a high-performance systolic array architecture, even for a seemingly simple computation [175]. Some of the recent HLS research has proposed end-to-end compilation flow to generate application-specific systolic arrays from C/C++ programs in a push-button manner [178, 84, 16, 40]. This approach allows programmers to focus on the algorithms, while the compiler automatically explores the design space and generates systolic arrays. The downside of such methods is that they either lack support for key optimizations (e.g., vectorization and I/O isolation) or fail to support a general class of systolic algorithms.

In most cases, the real problem for FPGA programmers is that it is non-trivial for non-experts to figure out what are the right customizations. In other words, the tools must encompass a certain amount of automation for achieving better performance. Even better, instead of directly implementing the customizations, the tools should provide programmers feedback or hints on what to optimize and thus the programmers also gain some control. Commercial HLS tools such as Xilinx SDAccel and Intel OpenCL Compiler already allow programmers to retrieve profiling data such as the HLS report and system estimate report. However, the insights provided by the profiling data are still limited. Thus, programmers need to analyze the data manually and infer the right customizations.

1.4 Dissertation Overview

In this dissertation, we introduce 1) a programming framework, HeteroCL, that consists of a programming model and a compiler targeting FPGAs, 2) a programming model, SuSy, for generating high-performance systolic arrays, and 3) a profiling technique for validating and recommending data reuse via trace-based analysis.

A more detailed overview of the remaining dissertation is as follows:

- Chapter 2 gives an introduction to essential customization techniques for FPGA-based accelerations. We guide the introduction with the roofline model [203]. We particularly focus on the techniques that are unique to custom accelerator designs, instead of the well-known code optimizations that are established for CPU or GPU targets. In addition, we survey various representative programming abstractions and compilers that facili-

tate the automatic generation of customized accelerator architectures from software programs.

- In Chapter 3, we introduce SuSy, a programming framework composed of a domain-specific language (DSL) and a compilation flow that enables programmers to productively build high-performance systolic arrays on FPGAs. With SuSy, programmers express the design functionality in the form of uniform recurrence equations (UREs), which can describe algorithms from a wide spectrum of applications as long as the underlying computation has a uniform dependence structure. The URE description in SuSy is followed by a set of decoupled spatial mapping primitives that specify how to map the equations to a spatial architecture. More concretely, programmers can apply space-time transformations and several other memory and I/O optimizations to build a highly efficient systolic architecture productively. Experimental results show that SuSy can describe various algorithms with UREs and generate high-performance systolic arrays by spatial optimizations. For instance, the SGEMM benchmark written in SuSy can approach the performance of the manual design optimized by experts, while using 30× fewer lines of code.
- In Chapter 4, we introduce HeteroCL, a programming infrastructure composed of a Python-based domain-specific language (DSL) and an FPGA-targeted compilation flow. The HeteroCL DSL provides a clean programming abstraction that decouples algorithm specification from three important types of hardware customization in compute, data types, and memory architectures. HeteroCL further captures the interdependence among these different customization techniques, allowing programmers to explore various performance/area/accuracy trade-offs in a systematic and

productive manner. In addition, our framework produces highly efficient hardware implementations for a variety of popular workloads by targeting spatial architecture templates such as systolic arrays and stencil with dataflow architectures. Experimental results show that HeteroCL allows programmers to explore the design space efficiently in both performance and accuracy by combining different types of hardware customization and targeting spatial architectures, while keeping the algorithm code intact.

- In Chapter 5, we introduce, DrTrace, a trace-based profiling method for data-dependent data reuse. DrTrace not only validates the data reuse scheme specified by programmers, but recommends data reuse candidates to the programmers. Moreover, by integrating the profiling flow with HeteroCL custom memory primitives, the users can easily improve and/or fix the programs according to the profiling results. Finally, with the HeteroCL compiler, we generate high-performance hardware architecture for stencil applications.
- Chapter 6 summarizes the contributions of this dissertation and discusses future research directions.

1.5 Collaborations, Funding, and Previous Publications

This dissertation would not be possible without the contributions of my colleagues within the Zhang Research Group in the Computer Systems Laboratory at Cornell University, as well as collaborators from University of California Los Angeles (UCLA) and Intel Labs. My advisor and committee chair, Prof. Zhiru Zhang, provided valuable suggestions and assistance to all the projects men-

tioned in this dissertation. The HeteroCL project is in collaboration with Yuwei Hu and Shaojie Xiang at Cornell and Dr. Cody Hao Yu, Dr. Jie Wang, Dr. Yuze Chi, and Prof. Jason Cong at UCLA. In addition, many undergraduate students and MEng students were involved in developing this project. We also got useful feedback from Intel. The SuSy project is in collaboration with Dr. Hongbo Rong, who was my mentor during my internship at Intel. We also got help from other undergraduate students from Tsinghua and Beijing University in China.

This dissertation was supported in part by a DARPA Young Faculty Award, NSF/Intel CAPA Awards #1723715 and #1723773, NSF Award #1909661, #1453378, #1436827, and #1707408, the Semiconductor Research Corporation (SRC), and research gifts from Intel and AMD Xilinx. We thank Amazon for providing AWS EC2 credits. A complete list of my publications during my Ph.D. in reverse chronological order is as follows:

1. Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, Zhiru Zhang: **HeteroFlow: An Accelerator Programming Model with Decoupled Data Placement for Software-Defined FPGAs.** International Symposium on Field-Programmable Gate Arrays (FPGA), 2022 [205]
2. Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, Zhiru Zhang: **Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects.** ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2021 [129]
3. Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, Yida Wang: **Bring Your Own Codegen to Deep Learning Compiler.** arXiv 2021 [27]

4. Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, Pradeep Dubey: **SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs.** International Conference On Computer Aided Design (ICCAD), 2020 [128]

5. Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, Zhiru Zhang: **HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing.** International Symposium on Field-Programmable Gate Arrays (FPGA), 2019 [127]

6. Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Kumar Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, Wenping Wang, Zhiru Zhang: **Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs.** International Symposium on Field-Programmable Gate Arrays (FPGA), 2018 [222]

CHAPTER 2

BACKGROUND

In this chapter, we focus on discussing various hardware customization techniques to address the performance challenge of FPGA programming. Our goal here is to provide the audience insights into *how to systematically design high-performance accelerators given the massive amount of customizable resources on FPGAs*. These hardware customizations can be done manually, automatically, or semi-automatically. To classify the hardware customizations in a systematic way, we leverage the widely used roofline model [203] that visualizes the performance bottlenecks and optimization directions. After introducing the customization techniques, we survey a prominent set of representative programming abstractions and compilers by discussing how they apply the customization techniques.

2.1 Essential Customization Techniques for FPGA-Based Acceleration

As shown in Figure 4.1d, in a roofline model, the x-axis represents the operational intensity (i.e., number of operations per byte of off-chip data access) while the y-axis represents the throughput (i.e., number of operations per second). We can plot different design points (in red) on the diagram according to the operational intensity and throughput. The highest achievable throughput of each design point is limited by the compute roof and memory bandwidth roof (blue lines). In the left part, the designs are memory-bound where the attainable throughput is limited by the off-chip memory bandwidth. In the right

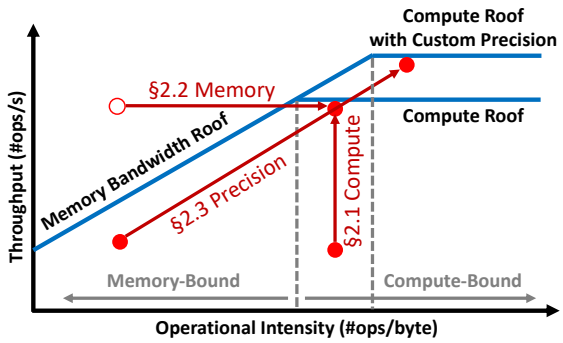


Figure 2.1: **Impact of different hardware customization techniques depicted in the roofline model –** x-axis represents the operational density and y-axis represents the throughput; custom compute engines improve the throughput by concurrently executing more operations per second; custom memory hierarchy can move memory-bound design points towards a more compute-bound region by increasing data reuse and memory bandwidth utilization; custom data representations can benefit both custom compute engines and memory hierarchy, thus further lifting the compute roof.

part, the designs are compute-bound, where the maximum throughput is determined by the amount of physically-available computing resources (e.g., the maximum number of DSPs and LUTs on an FPGA). Similar to the terminology defined in [203], here we use "operations" as a generic term to indicate the essential primitives that characterize the compute intensity of a given workload. Evidently, the meaning of this term is application specific. For example, multiply-accumulate (MAC) operations are commonly used for DSP and deep learning applications. For FPGA-based acceleration, these operations may exploit customized data types, which we discuss in a later section.

We classify the hardware customization techniques into the following major categories (also shown in Figure 2.1).

1. **Custom Compute Engines**. For compute-bound designs, custom compute engines can be developed to move the accelerator throughput towards the compute roof. Inside such a compute engine, designers typi-

cally explore the following pipeline and parallelization techniques to execute more operations concurrently to improve its throughput [71, 42, 41]: (1) accelerator-unique fine-grained custom pipeline, which is often deeply pipelined and tightly-coupled with fine-grained operator-level parallelization, different from CPUs and GPUs, (2) coarse-grained parallelization that further parallelizes multiple fine-grained pipelines, which can be both homogeneous (i.e., data parallelism) and heterogeneous (i.e., task parallelism) parallelization, similar to multicore processors, and/or (3) accelerator-unique coarse-grained pipeline that is composed of multiple fine-grained pipelines.

2. **Custom Memory Hierarchy**. For memory-bound designs, the accelerator memory hierarchy can be customized to move the design close to the (off-chip) memory bandwidth roof and move it right towards a compute-bound design. Different from CPUs/GPUs where their memory hierarchy is pre-designed and (almost) fixed, one unique opportunity (and challenge) for FPGAs is that their memory hierarchy is flexible and can be fully customized. Common optimizations include: (1) custom on-chip buffering and/or caching to improve data reuse and exploit much higher on-chip memory bandwidth, and (2) streaming optimization and/or custom network-on-chip to enable direct on-chip communication between multiple computing elements and/or bypass off-chip memory access.

3. **Custom Data Representations**. Finally, for both compute- and memory-bound applications, custom data representations with a reduced (or widened) bitwidth can play a vital and unique role in further improving the accelerator throughput. On one hand, it reduces the bytes of off-chip data access required by computing operations and benefits the cus-

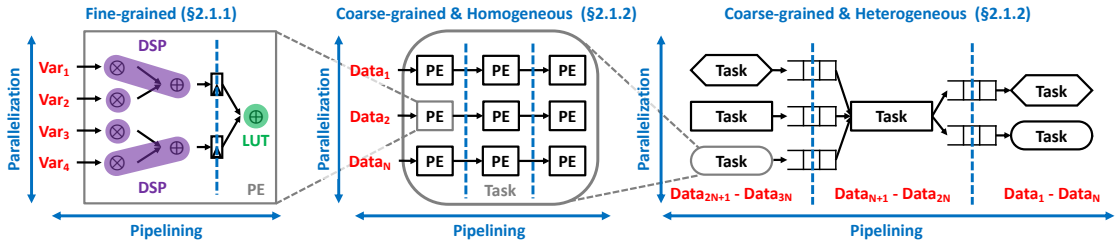


Figure 2.2: **Major forms of custom compute engines –** At a fine granularity, each PE runs a custom pipeline with a low II. In addition, optimizations such as operation chaining (i.e., multiple operations scheduled into one cycle as combinational logic) can reduce the area and sometimes the II. We can then connect multiple PEs to build a coarse-grained heterogeneous or homogeneous pipeline (e.g., systolic arrays). Meanwhile, with optimizations such as PE duplication, we can achieve coarse-grained homogeneous parallelization (e.g., data-level parallelism). We can also build a coarse-grained heterogeneous pipeline using heterogeneous tasks composed of homogeneous PEs (e.g., task-level parallelism).

tom memory hierarchy. On the other hand, with reduced bit-width, a single fine-grained custom pipeline consumes much fewer resources and one can accommodate more such pipelines in the custom compute engine to achieve more coarse-grained parallelism and a higher throughput. Hence with the custom precision, the compute roof is further moved up as the same FPGA can now run more operations.

In the following, we discuss more details of these hardware customization techniques and their interplay.

2.1.1 Compute Customization

The primary objective of customizing the compute engines is to improve the throughput of the accelerator, i.e., moving upward in the roofline diagram (see Figure 4.1d). To achieve this goal, an FPGA accelerator needs to make the best

use of the on-chip resources to maximize hardware parallelism and compute efficiency by instantiating many processing elements (PEs) in parallel. In this work we use "PE" as a generic term and loosely define it as a replicable hardware building block that executes a fine-grained loop (or function) pipeline with tens to hundreds operations. Similar to the notion of operations used in the roofline model, the exact function of a PE is highly application specific. To be more precise, we can break down *throughput* into three major factors, as shown in the following equation:

$$\begin{aligned} \textbf{\textit{Throughput}}\ (\text{OPs/sec}) = \textbf{\textit{Hardware Parallelism}}\ &(\text{max OPs/cycle}) \\ \times\ \textbf{\textit{Compute Utilization}}\ &(\text{busy OPs/max OPs}) \\ \times\ \textbf{\textit{Clock Frequency}}\ &(\text{cycles/sec}) \end{aligned} \tag{2.1}$$

$$\textit{Hardware Parallelism} = \#\textit{PEs} \times \textit{PE-Level Parallelism} \propto \#\textit{PEs} \times \frac{\#\textit{OPs per PE}}{\textit{II}}$$

Here *hardware parallelism* represents the maximum number of operations that can be concurrently executed per cycle with the given accelerator architecture, which typically exploits both parallelization and pipelining. This term can further be decomposed into a product of coarse-grained parallel factor (i.e., the number of PEs) and fine-grained PE-level pipeline parallelism that is typically reflected by the pipeline initiation interval (II) — the smaller the II, the higher the pipeline throughput. *Compute utilization* is defined to be a ratio that captures the utilization of the physical compute resources, namely, the functional units that execute the operations defined in the software algorithm. These functional units should be kept as busy as possible (ideally near 100% utilization). *Clock frequency* determines the actual operating clock rate of the hardware circuits that run on the FPGA.

Unlike CPUs and GPUs where the clock rate is fixed and often at the order of GHz for a given device, the operating frequency of an FPGA accelerator is usually one order-of-magnitude lower and highly depends on the degree of pipelining and parallelization, as well as the resource usage of the underlying architecture. Hence FPGA programmers must significantly improve the other two factors (i.e., hardware parallelism and compute utilization) and explore intricate design trade-offs amongst the three factors.

In the following, we introduce a set of optimization techniques for customizing the compute engines of an FPGA accelerator, which are classified by four dimensions. First, in terms of the parallelism form, we have *parallelization* and *pipelining*. Second, in terms of granularity, we have *fine-grained* and *coarse-grained* optimizations. We refer to the intra-PE parallelization/pipelining as fine-grained optimizations. In the HLS terminology, the scope of such optimizations is limited to a loop or function body where the inner loops, if any, are unrolled. On the other hand, we call the inter-PE optimizations coarse-grained. Third, the composition of the parallel or pipelined PEs can be either *homogeneous* or *heterogeneous*. Finally, these PEs can be scheduled *statically* at compile time and/or *dynamically* at run time.

Figure 2.2 gives an overview of custom compute engines and examples of optimizations from the first three dimensions (i.e., parallelism, granularity, and composition). Starting with fine-grained optimizations (the left figure), the primary goal is to reduce the II of pipelines inside each PE. To achieve that, one can apply techniques such as modulo scheduling, operation chaining and multi-pumping, loop transformation, and dynamic scheduling. Moving forward, at a coarse granularity, we focus on inter-PE optimizations. Depending on the com-

position of PEs, different techniques are proposed. With homogeneous composition (the middle figure), we focus on data-level parallelism. For instance, one can perform PE duplication for parallelization and build systolic architectures for pipelining. With heterogeneous composition (the right figure), we focus on task-level parallelism, where we have dataflow pipelining and multithreading for parallelization.

Fine-Grained Pipelining (and Parallelization)

In contrast to general-purpose processors, FPGAs allow a programmer to build a deep application-specific pipeline that obviates the need for instruction decoding, branch prediction, and out-of-order logic, which incur nontrivial overhead in both performance and energy. In this section, we discuss the common techniques to improve the PE-level throughput through fine-grained pipelining, parallelization, as well as some of the associated trade-offs between resource efficiency and frequency.

Loop Transformations that Improve Pipelining – Modulo scheduling typically does not alter the underlying control data flow graph of the loop subject to pipelining. However, in many cases, the inter-iteration dependences can be removed (or alleviated) through code transformations in multi-dimensional iteration space. A helpful tutorial is given in [63], which summarizes a number of loop transformations that resolve the II-limiting recurrences through reordering and interleaving nested loops.[1] Polyhedral compilation is also increasingly used to improve the efficiency of pipelining [223, 154, 170, 140, 136, 15].

[1]The same paper [63] also includes a comprehensive table that summarizes the commonalities and differences between traditional CPU-oriented code transformations and their counterparts in HLS. Hence we do not repeat those discussions here.

To further increase the hardware parallelism, HLS designs commonly use loop unrolling in combination with pipelining to increase the number of parallel operations per pipeline. Unlike unrolling on CPUs, the unfolded copies of the loop body require additional hardware on FPGAs. Nevertheless, an unrolled loop body is usually smaller than the original version, as unrolling often enables additional code optimizations such as constant propagation. Some loop transformations are also useful for increasing the utilization of the pipeline by minimizing "bubbles" due to the filling/draining of the pipeline. For example, loop flattening (also known as loop coalescing) can be applied to coalesce a nested loop into a single-level flattened loop so that the pipeline continuously executes the innermost loop without much frequent switching to an outer loop.

Dynamic Scheduling – Certain data dependences or resource contentions may occur in an input-dependent manner, which are known as data or control hazards in the terminology of a processor pipeline. These hazards may result in poor performance with the mainstream HLS tools, as they cannot be resolved statically with compile-time analysis and transformations. There are ongoing efforts to improve statically scheduled HLS to better handle these hazards [139, 4, 55, 66]. For example, Dai et al. [57] propose to insert application-specific hazard detection units into the pipeline generated by a modulo scheduler. Variable-latency operations can also be handled in a cost-effective way through pipeline flushing [56]. Another active line of research proposes to generate dataflow circuits to enable dynamically scheduled pipeline that yields better-than-worst-case performance in the presence of dynamic data dependencies or variable-latency operations [108, 92, 107, 69, 109]. However, extensive fine-grained handshaking is required to implement dynamic scheduling, which often results in nontrivial overhead in both resource usage and clock frequency.

To address this challenge, a hybrid scheduling method is proposed, which partitions the input program into statically and dynamically scheduled portions to improve efficiency [28].

Coarse-Grained Pipelining and Parallelization

The coarse-grained optimizations involve inter-PE parallelization and pipelining [49], where the composition of the PEs can be either homogeneous or heterogeneous. Compared to CPUs/GPUs, FPGAs have additional flexibility in customizing the shape and topology of the parallel PE array and the synchronization mechanism between PEs.

Inter-PE Parallelization – For computations that can be executed independently, one can allocate multiple PEs to execute them in a data-parallel fashion. This method is also known as PE duplication or compute unit replication. There are many ways to perform PE duplication. One example is unrolling the outer loop [41]. After loop unrolling, each unrolled inner-loop body becomes a PE that can be executed in parallel. A more automated approach is implemented in the Fleet framework [190], which automatically generates massively parallel PEs along with a high-performance memory controller. Many efforts realize homogeneous parallelization by leveraging useful features in the input language [164, 31]. For instance, FCUDA [164] is a compilation flow that maps CUDA code to FPGAs. This allows programmers to use the high-level CUDA APIs to describe coarse-grain data-level parallelism, where each thread-block inside a CUDA kernel becomes a hardware PE that can be executed in parallel with other PEs.

Software multithreading constructs can be used to explicitly specify coarse-grained parallelism for multiple (often heterogeneous) PEs. To this end, Hsiao et al. [90] propose a technique called thread weaving, which statically schedules computation in each thread with the awareness of other threads, and guarantees that the shared resources are accessed in a time-multiplexed and non-interfering manner. There are also approaches that resolve synchronization dynamically. FlexArch [23] is one example, which schedules the computations dynamically via work stealing [12] and assigns them to customized PEs. With work stealing, the computations can be more efficiently distributed and balanced across PEs.

Inter-PE Dataflow Pipelining – FPGAs programmers can also compose PEs (or tasks) into a coarse-grained dataflow pipeline that is typically dynamically scheduled. By overlapping the executions of different PEs and passing data between them with a streaming interface or Ping-Pong buffers, one can overlap the PE execution and decouple the compute from memory accesses to hide the off-chip memory access latency (more details in Section 2.1.2). Such dataflow pipelining is well-suited for many application domains, such as image processing [29, 196, 171], machine learning [202, 183], graph processing [53, 117, 220, 216, 160], and network switching [165]. Current HLS tools typically provide (vendor-specific) pragmas to realize dataflow pipelines of different forms. In general, the dataflow architecture is more scalable if the PEs are connected by asynchronous channels, but at the cost of additional handshaking logic between PEs; the pipeline is more efficient but perhaps less scalable (in terms of timing closure) if its stages are connected by wires or registers pulsed by the same clock.

Similar to coarse-grained parallelization, a dataflow pipeline can be com-

posed either heterogeneously or homogeneously. To construct heterogeneous pipelines, an example is the TAPA framework [30], which defines a programming interface to describe the dataflow parallelism within an application. The ElasticFlow architecture [189] is another example that implements the loop body as a multi-stage pipeline. There also exist some application-specific dataflow architectures [89, 176, 29, 87]. For instance, SODA [29] is a framework that generates a dataflow architecture for stencil computation, where data elements are updated with some fixed patterns. With the SODA framework, computations from different pipeline stages can be executed simultaneously with designated forwarding units, which also serve as reuse buffers.

Systolic arrays represent a well-known class of homogeneous pipelines, where connected PEs work rhythmically (shown in Figure 2.2). For each time step, each PE reads from its neighbors, processes the data, and forwards the results to its neighbors. Systolic arrays are also considered as a generalization of 1-dimensional pipelining. There is a long line of research that focuses on generating high-performance systolic arrays [178, 48, 85, 16, 202, 186, 128, 48, 193].

2.1.2 Memory Customization

As shown by the roofline model in Figure 4.1d, the key for memory hierarchy customization is to reduce the number of off-chip data accesses required by computing operations, i.e., increase the operational intensity. First, one can explore on-chip buffering and caching techniques to improve the on-chip data reuse. This is also often a key enabler of low-latency and high-throughput custom compute engines. Second, one can explore streaming optimization and

custom on-chip networks to bypass off-chip memory access and enable fast communication between multiple compute engines. Finally, for the essential off-chip memory accesses that cannot be avoided, one can explore data access reorganization techniques to fully utilize the off-chip memory bandwidth and prefetching techniques to hide the off-chip memory access latency.

On-Chip Buffer Optimization

Unlike general-purpose processors that have a fixed and pre-designed multi-level cache system, FPGAs provide a fully customizable on-chip memory hierarchy that can harvest the bandwidth of a massive amount of distributed registers, LUT RAMs, block RAMs (BRAMs), and ultra RAMs (URAMs). One of the most common and effective optimizations is to customize the on-chip buffers based on the application-specific memory access behavior. The FPGA programmers may use different types of reuse buffers such as (1) shift registers, line buffer, and window buffer that are predefined by HLS vendors [208], and (2) user-customized buffer structures. For both vendor-provided and user-customized reuse buffers, programmers often need to perform various loop transformations.

To minimize the required on-chip memory usage, an important technique is to apply loop tiling and carefully select the right tile size to balance the computation and memory access with the minimum buffer size.

Loop fusion can also be leveraged to reuse one buffer for multiple loops. Besides these commonly used loop transformations, several accelerator-specific optimizations can also be applied. For example, one can use shift register, line

buffer, and/or window buffer to avoid duplicate buffers required by multiple compute operations. The line buffer and/or window buffer is a more generalized version of a shift register, where it buffers just enough elements on-chip for data reuse as required by the computing engine and it shifts at every new iteration/cycle. As a result, only a minimum amount of new data has to be read from the off-chip memory into the buffer every clock cycle. One can also use small streaming FIFOs (first in, first out) to dataflow between multiple computing engines and avoid large on-chip buffers.

Another unique optimization for FPGA accelerators is to fully utilize the heterogeneous memory resources to increase parallel on-chip data accesses so that the compute units are not idling due to lack of data, namely, higher compute utilization in Equation (2.1).

Such on-chip data are typically large and require distributed BRAMs and URAMs, which are composed of hundreds to thousands of dual-port SRAM banks. Each physical memory port allows up to one read or one write—with no more than 36 bits for BRAM and no more than 72 bits for URAM—in one cycle. To enable parallel on-chip data access, the key is to apply the *memory banking* optimization in HLS to partition a large on-chip buffer into multiple small buffers that can be mapped onto multiple BRAM or URAM banks. When each data item has fewer than 36 bits for BRAM or 72 bits for URAM, *on-chip memory coalescing* can be further applied to stack partitioned smaller arrays to increase the port access bitwidth (or word width) of the BRAM and URAM banks.

Several FPGA HLS tools often provide directives for users to specify such array partitioning and reshaping to realize these optimizations [208, 101]. For example, Xilinx Vivado HLS [208] provides pragmas for users to partition an

array at multiple dimensions, in a block, cyclic, or complete fashion. Recent years have also seen an active body of research on automatic array partitioning.

For array accesses with affine indices based on the loop indices, several studies [45, 198, 197, 37] find that polyhedral compiler analyses can statically find array partitioning solutions to avoid on-chip memory bank conflicts and automate the partitioning process. In general, the polyhedral compilation is shown to be effective in co-optimizing the loop pipelining, parallelization, data reuse, and array partitioning together given a polyhedral program [170, 223]. For stencil applications, prior efforts have demonstrated success with polyhedral analysis [46, 29], where they can find the optimal reuse buffer setting and partition the buffer in a way to minimize both off-chip memory accesses and the on-chip buffer size. For non-affine programs, a few studies have taken a profiling-driven approach to automatically partition the arrays with trace-based address mining [221, 26].

On-Chip Cache Optimization

Most of the special-purpose accelerators target applications with regular memory accesses using the aforementioned on-chip buffering optimizations. On-chip caching provides another attractive alternative to accelerate applications with memory access patterns that are hard to predict statically; it also eliminates the tedious programming effort to explicitly manage the on-chip buffering.

As one of the early studies, the work in [32] evaluates the multi-port L1 cache design for parallel accelerators and finds the performance highly depends on the cache architecture and interface. LEAP-scratchpad [2] is another early

effort to provide multi-level cache support for FPGA accelerators. It abstracts away the memory system from accelerator designers. Another effort similar to (but different from) the cache abstraction is the connected RAM (CoRAM) approach [36]. CoRAM virtualizes a shared, scalable, and portable buffer-based memory architecture for the computing engines, which naturally comes at the cost of performance and resource overhead.

On several FPGA SoC devices from Intel and Xilinx, a shared coherent cache is available between the hardened ARM CPU and the FPGA accelerators [209, 102]. For datacenter FPGAs, efforts such as Intel Xeon-FPGA multi-chip package and IBM CAPI also provide coherent shared cache memory support for Xeon/PowerPC CPUs and FPGAs. A quantitative evaluation of modern CPU-FPGA platforms with and without coherency support can be found in [34, 35]. However, such cache designs are shared by the CPU and FPGA; there are no practically available, dedicated on-chip cache for the FPGA accelerators themselves yet. It is a potential area for more research.

Off-Chip Memory Optimization

We further discuss how to optimize the essential off-chip memory accesses that cannot be avoided easily. Here we mainly focus on the direct memory access (DMA) from a DDR or high-bandwidth memory (HBM). For the communication optimization between the host program and the FPGA accelerators, we refer the interested readers to [34, 35] for more details.

There are two major types of off-chip memory optimizations. The first type attempts to hide the off-chip memory access latency from the compute engines.

One of the common techniques is to use the double (or ping-pong) buffer technique [41, 42], which decouples the memory access (read or write) from execution via coarse-grained pipelining. Several HLS tools have automated the generation of the double buffers. Another optimization is to prefetch the data to the on-chip buffer/cache [24, 83].

The second type of optimization aims to fully utilize the off-chip memory bandwidth (BW), which is nontrivial for FPGAs. A recent study [143] characterizes the off-chip DRAM and HBM bandwidth for HLS programs under a comprehensive set of factors, including (1) the number of concurrent memory access ports, (2) the data width of each port, (3) the maximum burst access length for each port, and (4) the size of consecutive data accesses. To fully utilize the off-chip memory bandwidth, programmers have to carefully tune these parameters based on the insights summarized in [143]. Some common optimizations include *memory bursting* to increase the size of consecutive data access and the *off-chip memory coalescing* to increase the data access bitwidth. The Falcon Merlin compiler [43, 50] has partially automated some of these optimizations. The recent HBM Connect work [33] further proposes a fully customized HBM crossbar to better utilize HBM bandwidth.

2.1.3 Data Type Customization

Exploiting custom data representation is a vital optimization to achieve efficient hardware acceleration on FPGAs. In contrast to CPUs/GPUs, which employ arithmetic units with a fixed bitwidth, FPGAs allow programmers to use customized numeric types with the precision tailored for the given application.

When properly leveraged, such flexibility can lead to substantially improved efficiency in both custom compute engines and the custom memory hierarchy.

In this section, we discuss two major aspects of custom data representations: (1) parameterized numeric types and (2) automatic bitwidth analysis and optimization techniques.

Parameterized Numeric Types – Most FPGA HLS tools support arbitrary-precision integer and fixed-point data types. For example, Xilinx Vivado HLS provides `ap_(u)int` and `ap_(u)fixed` classes in C++ [208], while Intel HLS compiler uses `ac_(u)int` and `ac_(u)fixed` [101], which were originally developed by Mentor.

Using arbitrary-precision integer types, multiple low-bitwidth data elements can be packed together into a wide bit vector without increasing the footprint on the main memory. The packed data can then be read/written in a single memory transaction, which greatly improves the bandwidth utilization and the overall operational intensity of the accelerator. In addition, operations with reduced bitwidth require fewer resources, and thus more PEs can potentially be allocated on the same FPGA device to increase hardware parallelism. As a result, the compute roof is further lifted up, as illustrated in Figure 4.1d. For some applications, one can pack hundreds or thousands of bits into a single integer and perform bitwise operations very efficiently using the distributed LUTs and registers on the FPGA fabric. One example is calculating the hamming distance between two wide binary vectors via bitwise XORs. This kernel is used in many domains such as telecommunication, cryptography, and machine learning (e.g., binary neural networks [75, 82, 137, 212, 219], hyper-dimensional computing [93, 94, 177]).

Parameterized fixed-point types are also extensively used in FPGA design [144, 184, 81]. Fixed-point values are essentially integers with a predetermined position for the binary point. Their range is determined by the number of bits to the left of the binary point, while the precision depends on those to the right of it. Unlike floating-point units, fixed-point arithmetic units do not require expensive logic to manipulate the mantissa and exponent through rounding and normalization. Hence on an FPGA, fixed-point operations typically have a shorter latency and consume much fewer resources than their float-point counterparts.

In some cases, fixed-point types may cause nontrivial accuracy degradation if the represented data have a wide dynamic range. Hence efficient floating-point computation is desired. To this end, some recent FPGA devices (e.g., Intel Arria 10) offer hardened floating-point units (FPU), which obviate the need to perform the aggressive fixed-point conversion for an accuracy-sensitive application. Besides relying on FPUs that are strictly compliant with the IEEE 754 standard, the FPGA programmers can also leverage several existing libraries and tools that generate custom FPUs with reduced bitwidth [195, 9, 105]. For instance, FloPoCo is an open-source C++ framework that can generate customized FPUs in synthesizable VHDL [60].

There is an active line of research that explores new floating-point formats to accelerate machine learning workloads. Brain floating-point (bfloat) (originally proposed by Google) [194] is a truncated version of the IEEE single-precision floating-point format, which is now supported by the Intel Agilex FPGAs [99]. In addition, multiple recent efforts have implemented Posit arithmetic operators on FPGAs [21, 185]. Most recently, Microsoft has proposed MSFP [59], which is

a form of the block floating-point format, where the data in an entire tensor block share a single exponent. Hardened MSFP units have been added to the latest Intel Stratix 10 NX FPGAs [151].

Automatic Bitwidth Optimization – For an FPGA accelerator, the bitwidth settings of the numeric types can be a major determinant of its resource usage, the achievable frequency, and hence the throughput. It often requires a nontrivial amount of manual effort to quantize floating-point values into fixed-point types with the appropriate bitwidth. Hence there has been a large body of research that attempts to automate the float-to-fixed conversion process. With the existing approaches, *range analysis* is first performed (typically by a compiler analysis pass or a profiler) to obtain the minimum and maximum values of each variable in a given application. Note that such range analysis is also useful for reducing the bitwidth of the integer values. Afterward, *bitwidth allocation* is carried out to quantize or downsize the variables to a certain bitwidth. Finally, the resulting accuracy and other performance/area metrics need to be evaluated through estimation or simulation.

There are two popular methods to perform range analysis. The first method is to statically analyze a program that exploits the information on compile-time constants (e.g., loop bounds) and additional user-provided hints (often through pragmas or attributes) [118, 161, 191, 13, 14, 112]. The second one is to determine the input-dependent value ranges at run time [76, 120]. Static analysis often relies on interval arithmetic [88] and affine arithmetic [61] to infer the bound of each variable. In contrast, dynamic analysis can achieve additional reductions in bitwidth, although it may also introduce errors when the input samples do not cover some of the outliers. There are also hybrid approaches that attempt to

combine compile- and run-time methods. For instance, Klimovic and Anderson propose to first perform static analysis according to the common-case inputs suggested by the users while leveraging a run-time mechanism to fall back to software execution when outliers occur [120].

For bitwidth allocation, prior arts mostly adopt methods such as simulated annealing and satisfiability modulo theory [118, 134, 161, 191, 112]. It is worth noting that both range analysis and bitwidth allocation are computationally hard problems and can be very time consuming to solve. To address this challenge, Kapre and Ye propose a GPU-accelerated tool flow to automate and speed up the bitwidth optimization process for the FPGA-targeted HLS designs [112].

2.2 Representative Programming Abstractions and Compilers

There is a wide spectrum of programming models and compilers for FPGAs, such as HDL, HLS, polyhedral compilers, domain-specific languages, and new accelerator design languages. Due to the large body of existing work, we can only survey a (small) subset of the representative and more recent efforts. When describing a language/compiler, we focus on its productive features (e.g., metaprogramming, parametric types), performance features (e.g., supported hardware customizations), generality, and portability across FPGAs and other hardware targets.

2.2.1 Modern Hardware Description Languages (HDLs)

HDLs, such as Verilog, VHDL, and SystemVerilog, are the most widely-used languages for programming FPGAs at the register transfer level. They can be considered as the *assembly* languages of FPGAs. HDLs can be extended through libraries to appear somewhat higher level or can be integrated into more productive frameworks [18, 204]. Some tools like MathWorks HDL Coder [146] and LabVIEW [95] provide a graphic interface for users to create hardware blocks and compose a larger design visually. A recent trend in HDLs is to use a high-level language with modern features for describing hardware at RTL with better productivity, and build intermediate representations (IRs) that ease the analyses and optimizations. Examples include Bluespec [158], MyHDL [64], Cλash [5], PyMTL [142, 181], Chisel/FIRRTL [6, 104], and PyRTL [38].

Bluespec SystemVerilog (BSV) [158] is a non-traditional HDL in that it is based on rules that describe finite-state machines and are fired conditionally and atomically. A whole system behaves as a sequential composition of rule firings, which enables static verification. BSV incorporates polymorphism and overloading into its type system. The Bluespec compiler automatically detects conflicts between firings and generates a parallel execution schedule. BSV has been extended with scheduling decisions that determine performance [17].

PyMTL [142, 181] is embedded inside Python as a unified framework for building accelerators progressively from function-level (without timing) to cycle-level (with ticks or events as time), and finally to RTL (i.e., cycle-accurate, resource-accurate, and bit-accurate representation of hardware), so as to effectively assess performance, area, and energy of various accelerator architectures.

Chisel [6] is another modern HDL, which is gaining increasing visibility as a key part of the RISC-V ecosystem [199]. It is embedded in Scala and leverages many modern language features, such as object orientation, functional programming, parametric types, and type inference. With these features, a hardware description can be made more succinct and more reusable. The object orientation enables Chisel to implement highly parameterized circuit generators, such as memory cache generators. The Chisel compiler builds FIRRTL [104], to transform target-independent RTL to target-specific RTL. FIRRTL admits a variety of analyses (e.g., node counting, early area estimation, and module hierarchy depictions), optimizations (e.g., constant propagation, common subexpression elimination, and dead code elimination), instrumentation (e.g., hardware counters, hardware assertions, and source line coverage), and specializations for targets (e.g., memory is mapped to stylized Verilog, or hard macros, for FPGAs). Chisel and FIRRTL have been increasingly used in both industry and academy.

2.2.2 High-Level Synthesis (HLS)

Compared with HDL, HLS is a jump in productivity of programming FPGAs, at some reasonable cost of performance. HLS tools provide programmers high-level abstractions and derive efficient RTL from them, saving programmers from extensive hand-coding and tuning using low-level HDLs [47]. Consequently, HLS tools have been increasingly deployed for FPGA-based accelerations.

There is a large body of HLS tools. Below we briefly review them from the aspects of input languages, programming models, and compiler transformations.

For more comprehensive details on HLS, one may read surveys specifically on this topic [47, 156, 159, 58]. From the perspective of input languages, there are HLS tools based on C/C++ and other languages. From the perspective of programming models, HLS tools can be classified as control- or data-flow. From the perspective of compiler transformations, HLS tools can also be classified as purely pragma-driven or automatic polyhedral compilation.

C-Based and Non-C-Based HLS – State-of-the-art HLS compilers from major FPGA vendors allow a computation for acceleration to be described in C/C++, which is then synthesized into an FPGA accelerator either in the form of RTL or directly in bitstream by invoking downstream CAD tools. Representative commercial tools include Intel OpenCL SDK [52], Intel HLS [101], Intel OneAPI [103], Xilinx triSYCL [116], Microchip LegUp [20, 150], Mentor Catapult HLS [149], Cadence Stratus HLS [19], Bambu [168], GAUT [51], and Xilinx Vivado/Vitis HLS [47, 208]. The C-based HLS languages and compilers are easily accessible to programmers who are familiar with CPU programming. The same design can execute on both CPUs and FPGAs, and the HLS tools further provide basic performance portability across different target FPGA devices (from the same vendor). However, achieving high-performance with HLS requires nontrivial hardware knowledge and is very different from the conventional performance tuning process of CPU software programming. Programmers typically need to apply many vendor-specific pragmas that direct an HLS tool to generate the desired accelerator architecture. Also, programmers often have to significantly restructure their code to manually implement a number of customizations described in Section 2.1.

As FPGAs become increasingly common, FPGAs have also been added as a

target of other popular languages, including Python and Java. These languages generate either C-based HLS code or HDL code. For example, Hot & Spicy [182] translates Python code, which is written in a subset of Python syntax and annotated with types and pragmas, into synthesizable Xilinx Vivado HLS code; Synthesijer [188] maps Java into VHDL and Verilog, while Juniper [111, 77] maps Java to Xilinx Vivado HLS code.

Control- and Data-flow HLS – Mainstream HLS tools are based on control-flow (i.e., imperative) languages. In an untimed sequential program, the portion to be accelerated can be synthesized into fully timed HDL modules realizing the same functionality on FPGAs. Often, programmers can add pragmas/annotations in the accelerated portion so as to expose parallelism. HLS tools fall into this category include Xilinx Vivado/Vitis HLS [47, 208], Intel OpenCL SDK [52], Microchip LegUp [150], Mentor Catapult HLS [149], Intel oneAPI [103], Xilinx triSYCL [116], Bambu [168], Cadence Stratus HLS [19], and GAUT [51].

FPGA is naturally fit for dataflow execution due to its massive distributed hardware resources. A dataflow HLS program expresses a dataflow graph. For example, Maxeler [147], CAPH [179], and OpenDF [11] use meta-languages (e.g., CAL, MaxJ) to define a graph structure, and synthesize the nodes into individual RTL modules that are connected with communication channels. Other dataflow HLS tools (e.g., FAST-LARA [78] and OXiGen [167]) provide a high-level abstraction (e.g., C/C++ structs) to represent the dataflow structure.

Pragma-Driven and Automatic Polyhedral Compilation – Existing HLS tools commonly allow programmers to manually insert pragmas that direct the compiler to transform loops for performance. Recent years have also seen an active body of research on FPGA HLS that builds on polyhedral compiler frame-

works to perform many useful loop transformations in a fully automated fashion [223, 154, 170, 140, 49, 48, 7, 193].

Polyhedral compilation is a powerful approach to analyzing and optimizing loop nests that are static control parts (SCoP) [39, 15, 79, 10, 140, 141]. SCoP is a subclass of general loop nests with constant strides, affine bounds, and statically predictable conditionals that are affine inequalities of the associated loop iterators. Such a restricted form of loop nests is commonly seen in a broad range of numerical computations such as dense linear or tensor algebra, image processing, and deep learning algorithms. A polyhedral compiler uses parametric polyhedra as an intermediate representation (IR). The polyhedra represents (either perfectly or imperfectly) nested loops and their data dependences. Such an IR enables many useful dataflow analyses and effective compositions of a sequence of loop transformations. Polyhedral analyses and code transformations have been extensively used in optimizing compilers targeting CPUs and GPUs, especially for high-performance computing.

2.2.3 Domain-Specific Languages (DSLs)

Using DSLs can simplify the work of both programmers and compilers to identify and exploit opportunities for advanced customizations that are commonly used in a specific application domain. For this reason, it is not surprising to see many domain- or application-specific languages emerging in the past several years either for FPGAs (and maybe also ASICs) [157, 187, 152, 113, 162, 67] or re-purposed from CPUs or GPUs to FPGAs [163, 113, 162, 67]. These languages provide a limited number of, but optimized, functions/operators specific to

"hot" or important domains and applications (e.g., image processing, machine learning, network packet processing, software-defined radios, and even controllers of FPGA systems).

The narrower focus of the application domains enables DSLs to provide high-level programming interfaces for high productivity, and at the same time, very specialized implementations for high performance. DSLs could be much more productive than HLS to express FPGA and domain-specific customizations. Below we briefly review a few domains with some representative DSLs.

Image Processing – RIPL [187] defines a set of operations at the levels of pixels, window, and image. These operations are compiled into dataflow actors, which are finite state machines. The dataflow actors are lowered into CAL dataflow language [70] and then into Verilog. Darkroom [86] is a functional language restricted to stencil operations on images and automatically generates a schedule that is a pipeline of operations with minimum-sized line-buffers between them. Rigel [87] extends Darkroom for generating multi-rate pipelines, where the pipeline stages can fire with different rates. Hipacc [174] generates optimized OpenCL/CUDA code for image processing kernels running on FPGAs/GPUs.

Many computing patterns in image and video processing can be concisely described as nested loops in a declarative programming paradigm, as can be seen from several DSLs [86, 148, 119, 201]. Halide is the first to propose decoupling an *algorithm* from a *schedule* for a compute [173]. Here the algorithm is a declarative specification of the compute. The schedule then specifies how to optimize the algorithm for performance. Programmers only specify what optimizations to do, while the actual implementation of the optimizations is

left to a compiler. In this way, both productivity and performance can be achieved. Halide was originally only for CPUs and GPUs. Halide-HLS [171] extends Halide to target FPGAs. It generates a high-throughput image processing pipeline with line buffers, FIFOs, and also the glue code between the host and the FPGA accelerator.

Machine Learning – Many promising tools have emerged in this hot domain, such as TVM [25, 153], TABLA [145], DNNWeaver [180], DNNBuilder [218], Caffeine [217], and HLS4ML [67]. TVM [25] builds a deep learning compiler stack on top of Halide IR, supporting both CPUs and GPUs. TVM programs can target FPGAs as a back end by using VTA, a programmable accelerator that uses a RISC-like programming abstraction to describe tensor operations [153]. TABLA [145] implements an accelerator template, uniform across a set of machine learning algorithms that are based on stochastic gradient descent optimization. DNNWeaver [180] maps a Caffee [106] specification to a dataflow graph of macroinstructions, schedules the instructions, and uses hand-written, customizable templates to generate an accelerator. HLS4ML [67] translates neural network models learned by popular machine learning frameworks like Keras [115], PyTorch [166] and TensorFlow [1] into Xilinx Vivado HLS code, which generates bitstreams to run on Xilinx FPGAs.

Networking, Data Analytics, and Other Domains – P4FPGA framework [192] translates a network packet processing algorithm written in the P4 language into BlueSpec System Verilog for simulation and synthesis for FPGAs, and provides runtime support such as the communication between the host and the synthesized accelerator. Xilinx SDNet [207] is another FPGA-targeting framework for packet processing that supports the P4 language. Spark to FPGA

Accelerator (S2FA) [214] generates HLS C code from Apache Spark program written in Scala for big data applications. FSMLanguage [3] leverages Haskell to simplify the specification of finite state machines, which are common in hardware design.

2.2.4 Emerging Trends and Accelerator Design Languages

Recent years have seen several trends in accelerator design languages. First, *DSLs are becoming increasingly generalized, mixing imperative and declarative paradigms and/or embedded in general host languages*. While DSLs offer many advantages in productivity and compilation for individual application domains, more general-purpose language constructs are needed to (1) bridge the gaps between popular domains, (2) provide programmers with greater control on important customizations, (3) and serve as a compilation target for multiple high-level DSLs. Along this direction, we see new languages like DHDL, Spatial, and HeteroCL. DHDL [122] describes hardware with a set of parameterizable architectural templates that capture locality, memory access, and parallel compute patterns at multiple levels of nesting. DHDL is embedded in Scala and thus leverages Scala's language features and type system. Spatial [121] is a successor of DHDL, with more hardware-centric abstractions. HeteroCL, as is discussed in Chapter 4, is embedded in Python and blends declarative symbolic expressions with imperative code.

Second, *new programming models are providing more explicit controls to programmers for more predictable behaviors of the generated accelerators.* It has become a common practice for languages to be designed with high-level types and parame-

terized constructs modeling reconfigurable hardware components. These hardware components are typically implemented manually with high performance and maybe parameterized for flexibility. For example, Spatial [121] builds on parallel patterns, which map to efficient hardware templates; reduction can be done in registers or memory; programmers can build a custom memory hierarchy with at least three levels (DRAM, SRAM, and register); data can be stored on-chip in specific resources like look-up tables and specialized structures like queues, stacks, and line buffers. HeteroCL allows programmers to customize memory (via creating a custom memory hierarchy through banking, reuse buffers, and data streaming), compute (various loop transforms), and data types (bit-accurate types and quantization); systolic arrays and stencils can be efficiently built/handled by leveraging PolySA/AutoSA [48, 193] and SODA [29]. Dahlia [157] resembles traditional C-based HLS, but enhances the type system to prevent unsafe, simultaneous uses of the same hardware resource; with the type system, programmers can explicitly enforce some of the unwritten rules only known by the HLS experts (e.g., the unrolling factor should divide the banking factor of the arrays accessed inside the loop to achieve the best performance-area trade-off). Aetherling [68] abstracts sequential or parallel hardware modules into a set of data-parallel operators and features types for space sequence and time sequence that encode the parallelism and throughput of the hardware modules. Well-typed operators can thus be composed into statically scheduled streaming circuits.

Third, *separation of concerns becomes popular in language designs.* Inspired by Halide [173], HeteroCL and T2S [175, 186] separate algorithm definition from hardware customization, which leads to higher productivity. In the algorithm definition, HeteroCL allows both imperative programming and declara-

tive programming, making the language more general compared to other DSLs such as Halide and TVM. For hardware customization, HeteroCL supports all three essential techniques described in Section 2.1. For custom data representations, HeteroCL supports arbitrary-precision integers and fixed-point types. For custom compute engines, HeteroCL supports pipelining, unrolling, and several other loop transformations. For custom memory hierarchy, HeteroCL supports several on-chip buffering optimizations such as memory banking and data reuse.

T2S is designed based on an observation that "no matter how complicated an implementation is, every spatial piece of it must be realizing a part of the functionality of the original workload, and they are communicating based on production-consumption relationship" [175]. Therefore, a programmer could specify a temporal definition and a spatial mapping. The temporal definition defines the original workload functionally, while the spatial mapping defines how to decompose the functionality and map the decomposed pieces onto a spatial architecture. The specification precisely controls a compiler to actually implement the loop and data transformations specified in the mapping. So far, T2S focuses on expressing high-performance systolic arrays, which are often the most efficient way for accelerating a workload on an FPGA. While previous studies focus on how a systolic array computes, the input/output data paths are barely discussed. However, I/O is often the most complicated part in a real-world implementation. T2S allows a full systolic system to be built, including the core systolic array for a compute, other helper arrays for I/O data paths, host pre- and post-processing, and host and FPGA communication. T2S has two implementations, T2S-Tensor [186] and SuSy (discussed in Chapter 3) for building asynchronous and synchronous arrays, respectively. Specifically, SuSy

expresses the temporal definition as uniform recurrence equations (UREs), creates PEs with a space-time transform, and connects the PEs via shift registers. Note that UREs and space-time transformation are the theoretical foundation of most systolic arrays.

CHAPTER 3

SUSY: A PROGRAMMING MODEL FOR PRODUCTIVE CONSTRUCTION OF HIGH-PERFORMANCE SYSTOLIC ARRAYS ON FPGAS

Systolic algorithms have been extensively studied and employed in many important application domains such as bioinformatics, image processing, linear algebra, machine learning, and relational database [155, 126, 202, 97, 110, 62, 74, 124]. In a systolic algorithm, the dependence structure is uniform, where every data dependence has a constant distance. Mapping such dependence structures to spatial architectures lead to near-neighbor connections. The connected processing elements (PEs) jointly compose a *systolic array* that works rhythmically — at every time step, each PE reads inputs from some neighbors, performs computation, and forwards the inputs and results to other neighbors [123].

The characteristics of near-neighbor connections make systolic arrays a great match for FPGAs, where it is particularly important to minimize long interconnects to meet the target clock frequency. Indeed recent years have seen a growing number of application-specific systolic arrays implemented on modern FPGAs for efficient compute acceleration [155, 97, 202, 40, 73]. While systolic arrays typically have a very regular structure, it is far from trivial to achieve high performance unless the following optimizations are carried out: 1) finding an efficient mapping between a systolic algorithm and the physical array, 2) building an input/output (I/O) network to transfer data within the bandwidth limit, 3) constructing customized on-chip storage for data reuse, 4) vectorizing data accesses to better utilize the off-chip memory bandwidth, and 5) pipelining control signals to further increase throughput.

Obviously, any of the above optimizations would require substantial effort using the traditional RTL-based design methodology. The introduction of high-level synthesis (HLS) helps raise the level of design abstraction and hence increase productivity [47]. However, it remains challenging to strike the right balance between design quality and productivity using the existing HLS tools. To achieve high quality of results (QoRs), HLS users often have to perform "micro-coding", where some of the low-level micro-architectural details must be explicitly described and mixed into the behavioral specification that is supposed to be algorithmic and target-independent. In fact, it is not uncommon for HLS experts in the industry to spend several months on building a high-performance systolic array architecture, even for a seemingly simple computation [175]. Some of the recent HLS research has proposed end-to-end compilation flow to generate application-specific systolic arrays from C/C++ programs in a push-button manner [178, 84, 16, 40]. This approach allows programmers to focus on the algorithms, while the compiler automatically explores the design space and generates systolic arrays. Unfortunately, the existing methods either lack support for key optimizations (e.g., vectorization and I/O isolation) or fail to support a general class of systolic algorithms.

There exists another line of work that further raises the abstraction level of FPGA programming by using domain-specific languages (DSLs) [121, 127, 186, 171, 7, 153]. One recent example is HeteroCL (Chapter 4), a Python-based embedded DSL that provides a back end for mapping designs to systolic arrays. It is worth noting that systolic array support in HeteroCL also employs a push-button compilation flow and shares the same problems as other systolic compilers mentioned earlier. Another example is T2S-Tensor [186], which is a DSL built on Halide [173] that generates high-performance systolic arrays. With the

T2S-Tensor DSL, programmers can productively explore different optimizations with decoupled temporal definition and spatial mappings. However, this DSL is restricted to dense tensor computations.

Along this line, we propose SuSy, a programming framework built upon Halide [173] for productively building high-performance systolic arrays on FPGAs. SuSy decouples the algorithm specification from spatial optimizations, where the former can concisely express any systolic algorithm while the latter can describe essential optimizations for systolic arrays. Figure 3.1 provides a high-level overview of the proposed framework. The input program is specified in the SuSy DSL, which is composed of (1) an algorithm (or temporal definition) expressed in uniform recurrence equations (UREs) and (2) decoupled spatial optimization. The SuSy compiler lowers the input to an intermediate representation (IR) extended from Halide, where we perform user-specified optimizations and several target-specific transformations. The compiler then produces the HLS code (in OpenCL) as output, which is eventually compiled to bitstream for FPGA execution. Our main technical contributions are summarized as follows:

- This work is the first to demonstrate that high-performance customized systolic arrays can be built with many optimizations succinctly expressed in a DSL that is not tied to a specific application domain. The proposed SuSy DSL provides a clean programming model that decouples temporal algorithmic definitions from spatial mappings. Notably, the URE-based temporal specification can model a rich set of systolic algorithms used in many different applications. Examples include the Smith-Waterman algorithm in bioinformatics, convolution in deep learning, matrix multiplication in linear algebra,

and sorting.

- We introduce in SuSy an explicit and concise representation of space-time transformation, which allows the programmers to explore the trade-offs between performance and area with various temporal scheduling on different shapes of systolic arrays. In addition, SuSy further supports several essential spatial optimizations for building highly efficient systolic arrays, including vectorization, customized reuse buffer, data gathering/scattering for the I/O network.
- We have developed a comprehensive compilation flow targeting Intel FPGAs for SuSy. Experimental results show that SuSy can close the expert-designer performance gap on widely used compute kernels such as SGEMM, convolution, and Smith-Waterman. For dense tensor computations, we achieve more than 96% DSP efficiency. While for Smith-Waterman, we achieve 6.3× higher performance over a state-of-the-art framework.

3.1 Background

This section introduces the concepts of UREs and space-time transformations, and provides two illustrating examples.

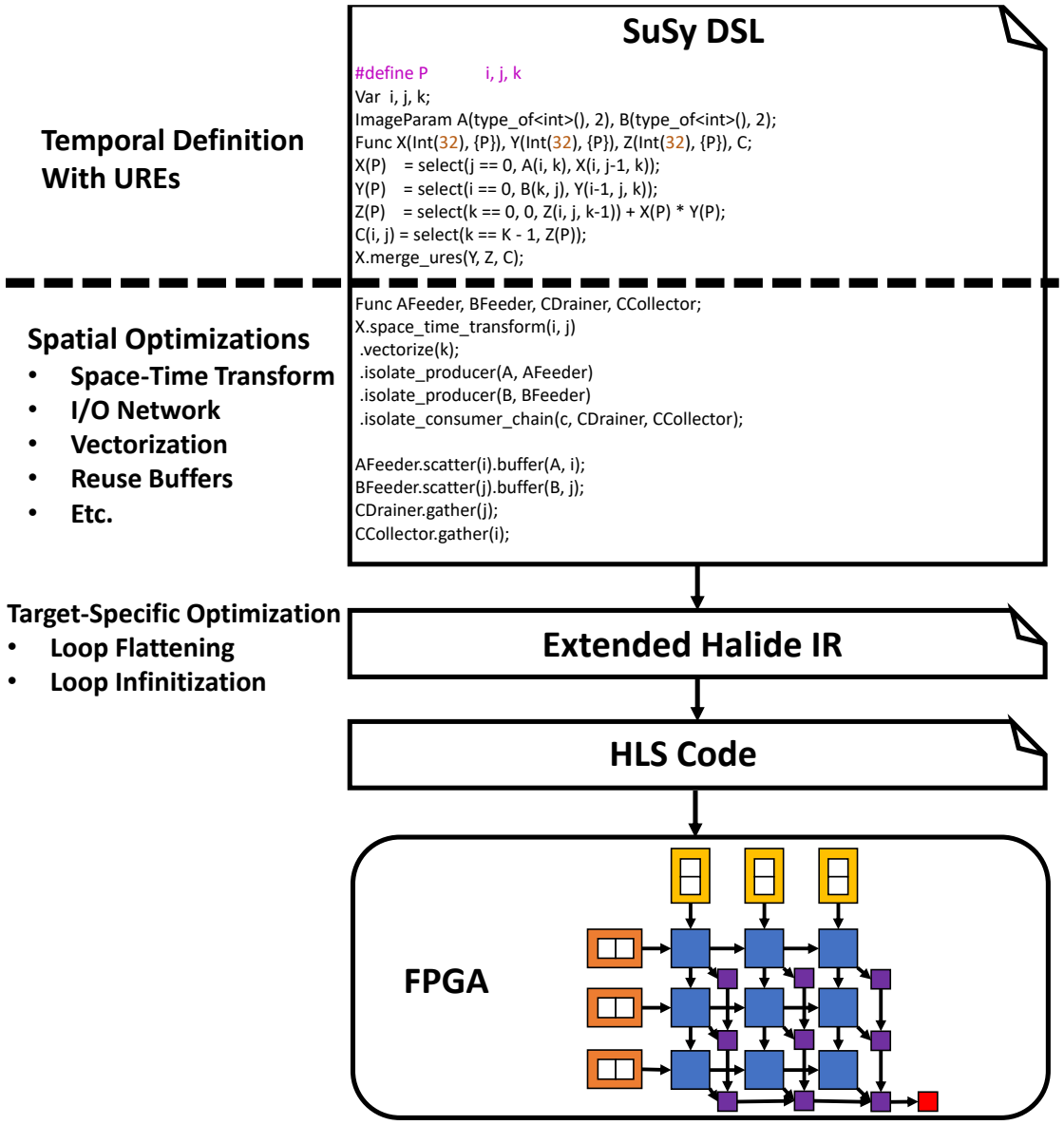


Figure 3.1: Overview of the SuSy programming framework.

3.1.1 Uniform Recurrence Equations (UREs)

Given an n-dimensional iteration space D, a system of UREs consists of a set of recurrence equations expressed in the following form [114]:

$$V_i(z) = f(V_1(z - d_1), V_2(z - d_2), ..., V_p(z - d_p)), \text{ for } z \in D$$

where $V_1, V_2, ..., V_p$ are variables, f is an arbitrary function, z is an n-dimensional vector representing a computation point (i.e., an iteration) in D, and d_i is an n-dimensional **constant** vector representing the distance from z. Basically, the UREs collectively represent an n-dimensional perfectly nested loop with constant dependence distances.

UREs have been extensively used in many programming frameworks for generating systolic arrays. The main reasons are twofold: 1) they are general and expressive enough to describe probably most systolic algorithms [132, 211, 172], and 2) they can specify both computation and data flow, which exposes more optimization opportunities to the compiler and programmers [131].

```
1 // select(cond, then_case, else_case) = then_case if cond is true
2 // otherwise it returns else_case (if provided)
3 for (k = 0; k < K; k++)
4  for (j = 0; j < J; j++)
5   for (i = 0; i < I; i++)
6    // URE for computing the multiplication and accumulation
7    Z(i, j, k) = select(k == 0, 0, Z(i, j, k-1)) + A(i, k)*B(k, j);
8    // Assign results to the output C
9    C(i, j)    = select(k == K - 1, Z(i, j, k));
```

(a) GEMM

```
1 // MAX = maximum possible value
2 for (i = 0; i < N; i++)
3  for (j = 0; j < N; j++)
4   X(i, j) = select(j == 0,
5                    max(A(i), select(i == 0, MAX, Y(i-1, j))),
6                    select(i >= j, max(X(i, j-1), Y(i-1, j))));
7   Y(i, j) = select(i == j,
8                    select(j == 0, A(i), X(i, j-1)),
9                    select(i > j, min(select(j == 0, A(i), X(i, j-1)),
10                                     select(i == 0, MAX, Y(i-1, j)))));
11  B(j) = select(i == N-1, Y(i, j));
```

(b) Insertion sort

Figure 3.2: Examples of using UREs.

Here we show two examples of UREs in Figure 3.2 along with the loop nests representing the iteration space. In Figure 3.2a, a general matrix multiplication (GEMM) kernel is described with UREs. In this example, we calculate `C` = `A`×`B`,

where `A` is an `I×K` matrix, `B` is a `K×J` matrix, and `C` is an `I×J` matrix. We use a single URE (L7) to describe the multiplication and accumulation, where we have one variable `Z` in a 3-dimensional domain `(i, j, k)` for storing the partial sum. After the calculation completes, we assign the results to output `C` in L9. Note that if the `select` expression does not have a false case, nothing is performed should the condition fail. Another example is shown in Figure 3.2b, where we perform insertion sort on an input vector `A` and store the final output in `B`. Here we have two UREs (L4-L10) with variable `Y` storing the sorted results after step `j` and `X` being an auxiliary variable. From these two examples, we can see that as long as an algorithm has constant dependence distances, we can describe it using UREs.

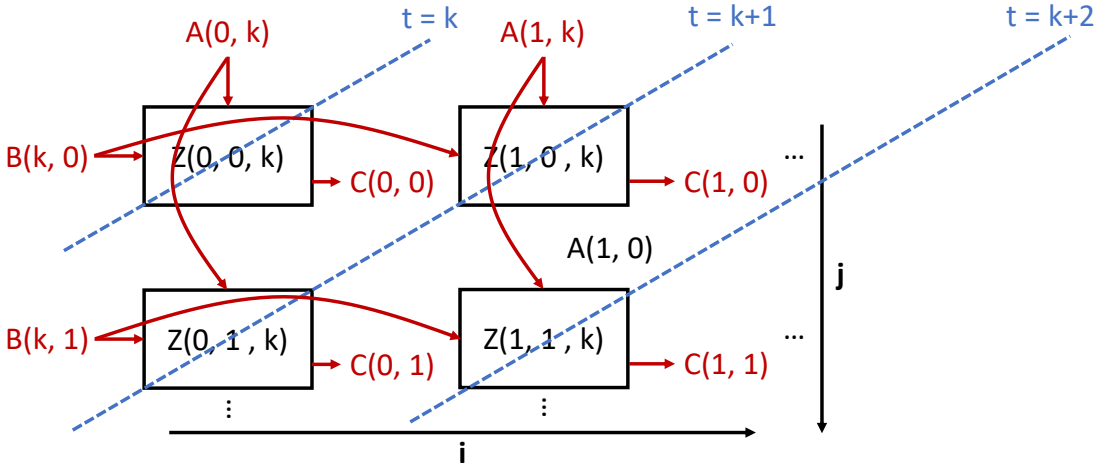
3.1.2 Space-Time Transformation

UREs alone only describe the function of the systolic algorithm without providing any spatial information. To build a systolic array from UREs, we need to determine the mapping between the domain of the UREs and the physical array dimensions. Space-time transformation [125, 131] is in essence a loop transformation that specifies the mapping. To be more specific, the transformation maps an n-deep loop nest to a time loop and $n-1$ space loops. The space loops are mapped to different PEs, and the time loop is used to schedule the original iterations to run on the PEs. The transformation can be described by a transformation matrix T:

$$T = \begin{pmatrix} \Pi \\ \tau \end{pmatrix},$$

```
1 // t = k + j + i
2 time for (t = 0; t < I+J+K-2; t++)
3  space for (j = 0; j < J; j++)
4   space for (i = 0; i < I; i++)
5    // recover the original loop variable before transformation
6    k = t - j - i;
7    // only compute if it is in the original domain
8    if (0 <= k < K)
9     // we use the last dimension to represent the time distance
10    Z(i, j, 0) = select(k == 0, 0, Z(i, j, 1)) + A(i, k) * B(k, j);
11    C(i, j)    = select(k == K - 1, Z(i, j, 0));
```

(a) Loop structure after space-time transformation.



(b) Mapped systolic arrays.

Figure 3.3: Example of applying space-time transformation to GEMM UREs.

where τ is a *scheduling vector* that generates the time loop and Π is an $(n-1) \times n$ *projection matrix* that generates space loops. A transformation matrix is valid only if it preserves the data dependence, and if no two iterations are scheduled to run on the same PE at the same time. In this work, we always set the projection matrix Π to be an identity matrix. This is a common practice when experts manually build systolic arrays [155, 97]. The support for non-identity projection matrices is left as future work.

Figure 3.3 shows an example of applying space-time transformation to the UREs in Figure 3.2a, where

$$T = \begin{pmatrix} 100 \\ 010 \\ 111 \end{pmatrix}, \tau = \begin{pmatrix} 111 \end{pmatrix}, \Pi = \begin{pmatrix} 100 \\ 010 \end{pmatrix}.$$

If we take a look at the loop structure after the transformation (Figure 3.3a), loops `i` and `j` become space loops and loop `k` is replaced with a time loop `t`. In other words, after transformation, we have a total of $I \times J$ PEs. Then, we need to check if the data dependence is still preserved by calculating new distances, which can be done by multiplying T with the distance vector. For example, the new distance of `Z(i, j, k-1)` can be calculated by $\left(\begin{smallmatrix} 100 \\ 010 \\ 111 \end{smallmatrix}\right)\left(\begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 0 \\ 0 \\ 1 \end{smallmatrix}\right)$. This is a positive dependence vector, and thus the original data dependence is preserved [131]. [1]

We also need to make sure the computation after transformation lies in the original domain by recovering the original loop indices and adding a check (L5-8). We can now easily map the computation into a 2-D systolic array, as shown in Figure 3.3b, where the red lines represent the communication with I/O and the dotted blue lines represent the time schedule.

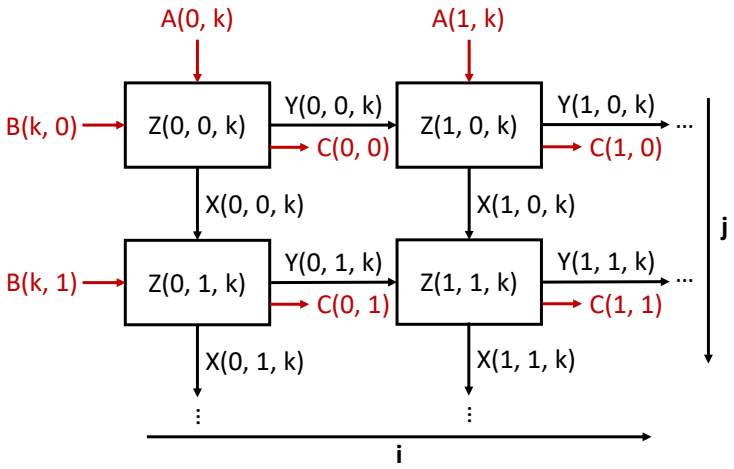
3.1.3 Design Space Exploration

In this section, we demonstrate how we can explore different design choices by combining UREs and space-time transformation. As can be seen in Figure 3.3b,

[1] Traditionally, all data dependence should be strongly satisfied for time loops (i.e., the dependence distance should be greater than zero). However, such a transformation usually introduces loop skewing that leads to complicated hardware. In SuSy, we allow the dependence to be weakly satisfied (i.e., the dependence distance could be zero) and let the hardware compiler take over the scheduling of PEs.

```
1  for (k = 0; k < K; k++)
2   for (j = 0; j < J; j++)
3    for (i = 0; i < I; i++)
4     // UREs for reusing the inputs
5     X(i, j, k) = select(j == 0, A(i, k), X(i, j-1, k));
6     Y(i, j, k) = select(i == 0, B(k, j), Y(i-1, j, k));
7     // URE for computing the product and accumulation
8     Z(i, j, k) = select(k == 0, 0,
9                         Z(i, j, k-1)) + X(i, j, k)*Y(i, j, k);
10    // Assign results to the output C
11    C(i, j)    = select(k == K - 1, Z(i, j, k));
```

(a) New UREs with input data reuse.



(b) Mapped systolic arrays.

Figure 3.4: Example of modifying UREs with data reuse.

inputs `A` and `B` are broadcast to all PEs, which is not scalable. To solve this, we can reuse the inputs by sending them through neighbor PEs. We can describe such data flow between PEs by modifying the UREs (Figure 3.4).

Figure 3.4a shows the new set of UREs with data reuse by introducing two new equations in L5-6. Specifically, variables `X` and `Y` store the values of inputs `A` and `B`, respectively. After applying the same space-time transformation, the mapped systolic array is shown in Figure 3.4b, where the black lines represent the communication between PEs. This simple example demonstrates how UREs provide programmers more flexibility when exploring the design space. Similarly, by choosing different transformation matrices, programmers can explore

the trade-offs between area and performance.

3.2 The Programming Model

The SuSy programming model is built upon Halide [173], and the main reasons are as follows: 1) The Halide DSL cleanly decouples the algorithm specification and temporal schedule. In SuSy, we inherit the same concept by decoupling the temporal definition from spatial optimizations, allowing programmers to efficiently explore different design choices without modifying the algorithm definition. 2) Halide abstracts algorithms composed of multi-level loop nests with declarative programming, which is a good fit for UREs because of the underlying multi-dimensional iteration space. The bounds can either be inferred from the input shapes or explicitly specified by the users. 3) Halide provides a concise yet expressive IR, which can be easily extended for describing optimizations required to generate high-performance systolic arrays.

In this section, we explain the SuSy programming model in detail. We first explain how we use UREs to describe temporal definitions in Section 3.2.1. Then we demonstrate how we apply a set of spatial optimizations in Section 3.2.2. For better illustration, we continue to use the GEMM example.

3.2.1 Temporal Definition

In SuSy, we extend Halide to express UREs, since the original Halide syntax does not support recurrent functions.

```
1 // Define the inputs with integer type and two dimensions
2 ImageParam A(type_of<int>(), 2);
3 ImageParam B(type_of<int>(), 2);
4 // Extend Halide's syntax for describing data type and placement
5 // We use C macros to simplify the code
6 #define ftype Int(32), {i, j, k}, Place::Device
7 Var i, j, k;
8 Func X(ftype), Y(ftype), Z(ftype), C;
9 X(i, j, k) = select(j == 0, A(i, k), X(i, j-1, k));
10 Y(i, j, k) = select(i == 0, B(k, j), Y(i-1, j, k));
11 Z(i, j, k) = select(k == 0, 0,
12                             Z(i, j, k-1)) + X(i, j, k)*Y(i, j, k);
13 C(i, j)    = select(k == K - 1, Z(i, j, k));
```

Figure 3.5: Describing UREs for GEMM in SuSy.

We show an example in Figure 3.5, where we describe the UREs for GEMM. We first declare the input matrices `A` and `B` with `ImageParam`, where we specify the data type and the number of dimensions (L2-3). Currently, SuSy supports the same set of data types as Halide (i.e., 64/32-bit float and 64/32/16/8-bit integer types). Then we define the iteration space and variables with `Var` and `Func`, respectively (L6-8). Unlike Halide, programmers can specify the data placement with either `Place::Device` (i.e., FPGA) or `Place::Host` (i.e., CPU). Since we offload the entire application to the FPGA, we choose `Place::Device` for all variables. We write down the UREs in L9-13 by referencing Figure 3.4a. With the declarative programming style, programmers do not need to explicitly write down the loop nests.

3.2.2 Spatial Optimization

After describing the temporal definition with UREs, we need to specify how we map them to systolic arrays as well as other spatial optimizations. With the decoupled programming style, users can efficiently apply different spatial mappings by using the SuSy *primitives* (or *scheduling functions* in terms of Halide). In

Table 3.1: Primitives for spatial optimizations in SuSy.

Primitive	Description
`F.merge_ures(U1, U2, ..., Un)`	Define the set of UREs `F`, `U1`, `U2`, ..., `Un` to optimize.
`F.space_time_transform(space, tau)`	Specify the space-time transformation that will be applied to `F`, where `space` is the set of space loops, and `tau` is the scheduling vector.
`F.vectorize(var)`	Vectorize the specified loop variable `var` of `F`.
`F.reorder(var1, var2, ..., varn)`	Reorder the loop nest for `F` according to the specified order, starting from the innermost level.
`F.tile(var, varo, vari, factor)`	Tile the loop variable `var` of `F` into two new variables `varo` and `vari` with a factor of `factor`.
`F.isolate_producer({E1, E2, ...}, P)`	Isolate a list of expressions {`E1`, `E2`, ...} (usually inputs) in `F` to a separate producer kernel `P`.
`F.isolate_consumer(E, C)`	Isolate an expression `E` (usually an output) in `F` to a separate consumer kernel `C`.
`F.remove(var)`	Remove loop `var` of `F`.
`F.buffer(E, v, mode)`	Insert a reuse buffer at loop `v` for expression `E` with `mode` (either `Buffer::Single` or `Buffer::Double`).
`F.scatter(E, var)`	Reduce data communication overhead (i.e., data broadcast) by scattering the expression `E` to the consumer along loop `var`.
`F.gather(E, var)`	Reduce data communication overhead (i.e., data broadcast) by gathering the expression `E` from the producer along loop `var`.

this section, we describe the syntax and semantics of selected primitives in detail. Table 3.1 shows the set of primitives we currently support.

Space-Time Transformation – As we have described in Section 3.1, to map UREs to a physical systolic array, we need to perform space-time transformation. An example is shown in Figure 3.6.

To specify the transformation in SuSy, we first need to define the tar-

```
1 X.merge_ures(Y, Z, C);
2 X.space_time_transform({i, j},     // space loops
3                        {0, 0, 1}); // scheduling vector
```

Figure 3.6: Primitive for specifying the transformation.

get set of UREs, which can be achieved by using the primitive `merge_ures` (L1). Then we establish the transformation by employing the primitive `space_time_transform` (L2-3), where the first argument specifies the space loops, and the second argument defines the scheduling vector. To better illustrate the optimizations without losing the generality, here we use a simpler time schedule than the one in Figure 3.3b.

We omit the space matrix here since it is an identity matrix as mentioned in Section 3.1.2. There are several constraints to the arguments. First, only the inner-most loops can be space loops. Otherwise, programmers need to perform loop reordering with `reorder` before applying space-time transformation. Second, the transformation matrix must be valid in terms of preserving the dependence.

Tiling and Vectorization – In most cases, the problem size may be too large to fit the given hardware resources. To solve that, we can tile the design and compute only the partial results of each tile on-chip. In addition, with tiling, programmers can explore another dimension of parallelism by applying vectorization, where we compute a fixed-length of data at a single time. With vectorization, we can perform vector loads/stores from/to the off-chip memory to better utilize the off-chip memory bandwidth. An example is shown in Figure 3.7a.

In this example, we first tile the `k` loop into `ko` and `ki` with a factor of `KI` via

```
1 // Tile loop k with factor KI
2 X.tile(k, ko, ki, KI)
3 // Vectorize the inner loop
4 X.vectorize(ki);
```

(a) Vectorization

```
1 // Define the loaders and unloaders
2 Func A_Loader, B_Loader, C_Unloader;
3 // Isolate the inputs to loaders
4 X.isolate_producer(A, A_Loader)
5  .isolate_producer(B, B_Loader)
6 // Isolate the output to unloaders
7  .isolate_consumer(C, C_Unloader);
```

(b) Isolation

Figure 3.7: Applying vectorization and isolation in SuSy.

the primitive `tile` (L2). Then, we vectorize the `ki` loop in L4. After vectorization, we are computing a total of $\texttt{I} \times \texttt{J} \times \texttt{KI}$ computations in parallel.

Input/Output Isolation – To further improve the performance, we can overlap the execution of the off-chip memory accesses with on-chip computations so that the communication latency does not throttle the overall throughput of the systolic array. We name such an optimization as *isolation*, which is conceptualized in Figure 3.7b.

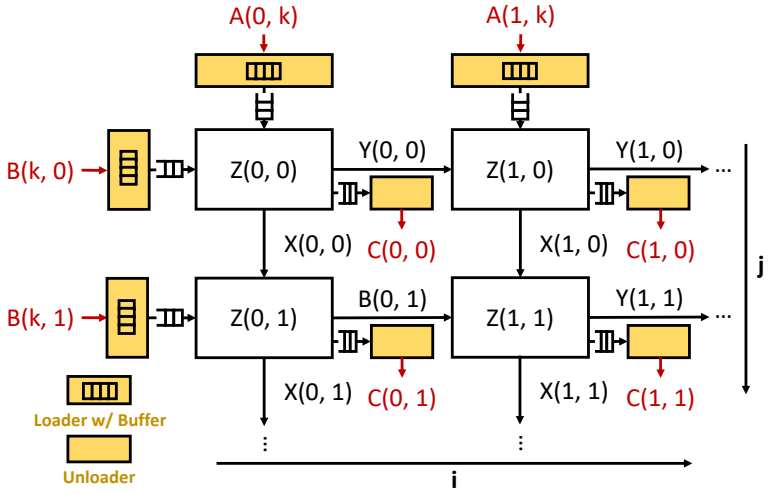
In the GEMM example, we have three off-chip memory accesses, which are *loading* input values from `A` and `B` and *unloading* output values to `C`. To isolate the access, we introduce new computation stages – two *loaders* for reading the input values and one *unloader* for writing the output values (L2). To describe the behavior, we use the primitives `isolate_producer` to isolate inputs (L4-5) and `isolate_consumer` to isolate outputs (L7). After isolation, the main computation kernel reads/writes data from/to loaders/unloaders instead of the off-chip memory.

Reuse Buffer Insertion – In many cases, we are loading repeated data from inputs due to the underlying iteration space. For instance, in GEMM, input `A` only depends on loop `i` and `k`. However, under the three-dimensional iteration space, we need to load the same data for `J` times. To reduce the memory ac-

cesses, we can load the data once from the off-chip memory and store it into an on-chip reuse buffer. In other words, all succeeding data accesses will load from the reuse buffer instead of the host memory. Figure 3.8a L2-3 show how we remove the loop with repeated access via `remove` and insert a reuse buffer via `buffer`. Users can further specify the loop level for inserting the buffer, which allows users to explore the trade-off between area (buffer size) and throughput.

```
1 // Remove the reuse loop and insert double buffers
2 A_Loader.remove(j).buffer(A, j, Buffer::Double);
3 B_Loader.remove(i).buffer(B, i, Buffer::Double);
```

(a) Primitives for inserting reuse buffer.



(b) Optimized systolic arrays.

Figure 3.8: Inserting reuse buffer to SuSy.

Finally, Figure 3.8b shows the systolic array after applying all spatial optimizations mentioned above. After isolation and buffer insertion, the main computation kernel reads input data from the double buffers inside the loaders. Meanwhile, loaders read input data from the off-chip memory.

Other Optimizations – SuSy provides several additional spatial optimizations, including gathering/scattering and data serialization/de-serialization. With gathering and scattering, we reduce the number of connections between

the systolic array and off-chip memory, which makes our design more scalable. Meanwhile, data serialization improves the utilization of the off-chip memory bandwidth by serializing data on the host before sending them to the systolic array. Similarly, we can perform de-serialization after we collect the results from the systolic array.

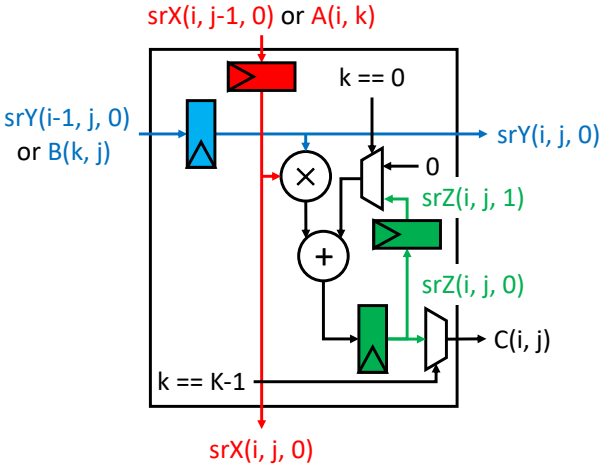
3.3 Compilation

In this section, we first explain the PE architecture generated by SuSy. We then describe a few representative back-end specific optimizations that are automatically applied. We also briefly discuss how we generate HLS code and deploy it to FPGAs.

PE Architecture – There are two ways for each PE to communicate with each other. First, they can communicate *asynchronously* through channels. However, channels may introduce unnecessary control overhead in hardware (e.g., handshaking). Therefore, SuSy generates *synchronous* architecture using shift registers. Specifically, each PE is associated with several shift registers that store the values of each variable (Figure 3.9b). For instance, variable `X` is associated with a shift register `srX`. The equivalent loop structure with shift registers is shown in Figure 3.9a, where we have three shift register for the variables `X`, `Y`, and `Z` (L1). The shift register size equals to the maximal time distance plus one. For example, the maximal time distance for variable `Z` is one (L15) as described in Section 3.1.2. Thus, the size of shift register `srZ[i][j]` for PE `(i,j)` is two (L1). The registers are shifted at the beginning of each time step (L3-8), right before we perform the computations (L10-16). In addition, after space-time transfor-

```
1  int srX[I][J][1], srY[I][J][1], srZ[I][J][2];
2  for (t = 0; t < K; t++)
3   // shift register logics
4   unrolled for (j = 0; j < J; j++)
5    unrolled for (i = 0; i < I; i++)
6     unrolled for (s = 0; s < 1; s++)
7      srZ(i, j, 1-s) = srZ(i, j, 0-s);
8      // no need to shift srA and srB
9   // computations
10  unrolled for (j = 0; j < J; j++)
11   unrolled for (i = 0; i < I; i++)
12    k = t;
13    srX(i, j, 0) = select(j==0, A(i, k), srX(i, j-1, 0));
14    srY(i, j, 0) = select(i==0, B(k, j), srY(i-1, j, 0));
15    srZ(i, j, 0) = select(k==0, 0, srZ(i, j, 1)) +
16                               srX(i, j, 0) * srY(i, j, 0);
17    C(i, j)      = select(k==K-1, srZ(i, j, 0));
```

(a) Equivalent HLS code.



(b) Hardware architecture of PE(i, j).

Figure 3.9: Equivalent HLS code and corresponding PE architecture after performing space-time transformation in Figure 3.6 — In the hardware architecture, we can see that there are three shift registers, which are `srX`, `srY`, and `srZ` respectively. For `srX` and `srY`, they take values from either inputs or neighbor PEs and send the values to the neighbor PEs. On the other hand, `srZ` is updated with its previous value within the same PE and sends out the results only when the accumulation is complete.

mation, we mark the space loops as unrolled while the time loops are pipelined automatically by the HLS compiler.

Target-Specific Optimizations – The SuSy compiler also applies a set of optimizations automatically to further improve the performance. These are de-

signed for the back end we currently target, namely, the Intel HLS tool. The specific optimizations include 1) loop flattening, which flattens a loop nest by combining neighbor loops into a single loop to reduce the control overhead, and 2) loop inifinitization, which replaces a flattened loop with a `while(1)` loop to further reduce pipeline stalls.

Code Generation – We extend the Halide OpenCL code generation to generate Intel HLS code. Since data serialization, de-serialization, and some low-level optimizations are still under development at this stage, we manually implemented them by slightly changing the generated HLS code. Then we push the code through the Intel HLS compiler and downstream CAD tool flow to produce the final bitstream that runs on the hardware.

3.4 Evaluation

In this section, we evaluate the systolic arrays generated by SuSy. All experiments are conducted on Intel vLab Academic Cluster [100], equipped with Intel Xeon Platinum 8280 CPU (2.70 GHz) and Intel Arria 10 GX FPGA. We first demonstrate the flexibility and productivity of SuSy by showing results on four benchmarks from different application domains, including single-precision general matrix multiplication (SGEMM), tensor-tensor multiplication (TTM), convolution (Conv), and Smith-Waterman (SW). We further provide in-depth analysis on SGEMM, Conv, and SW, where we perform quantitative comparison against existing frameworks such as Spatial [121], HeteroCL [127], T2S-Tensor [186], and PolySA [40].

Table 3.2 lists the key characteristics of Intel Arria 10 GX and Xilinx Ultra-

Table 3.2: Specifications of two FPGAs used in evaluation.

	Intel Arria 10 GX	**Xilinx VU9P**
Targeted By	[186][202] [SuSy]	[40][121][127]
Technology Node	Intel 20nm	TSMC 14nm/16nm
Soft Logic	427K ALMs	1,182K LUTs
DSPs	1,518 FP DSP	6,840 DSP48E2
BRAMs	2,713	2,160
Max Device Frequency	500 MHz	800 MHz

Scale+ VU9P; these two FPGA devices are used by the related work that we are comparing against in the remaining section. Note that each single-precision floating-point (FP) multiplication and accumulation (MAC) operation maps to one hardened FP DSP on Intel Arria 10, whereas the same MAC operation consumes five 27x18 DSP48E2 units on Xilinx Ultrascale+ VU9P. There are 6840 DSP48E2 units in total on VU9P, which roughly translates to 6840/5=1368 Intel FP DSPs (vs. 1518 on Arria 10). Hence we argue that these two FPGAs have a similar computation power in terms of the peak throughput on MAC, although the Xilinx VU9P is listed with a higher maximum device frequency.

Table 3.3: Evaluation results for benchmarks in SuSy.

Benchmark	**Problem Size**	**LOC**	**#ALMs**	**#DSPs**	**#BRAMs**	**Freq. (MHz)**
SGEMM [40][121][127][186]	(1024, 1024, 1408)	25	40%	93%	32%	202
TTM [186]	(256, 64, 256, 352)	25	33%	93%	31%	209
Conv [40][202]	3rd Layer of VGG-16 (64, 128, 112, 112, 3, 3)	28	35%	84%	30%	220
SW [121][127]	(1M, 128)	44	33%	0%	20%	225

General Evaluation – First, we evaluate the flexibility and productivity of SuSy using four benchmarks, including SGEMM and TTM in linear/tensor al-

gebra, convolution in deep learning, and SW from bioinformatics. Table 3.3 shows that we can describe a rich set of systolic algorithms in SuSy, each with just tens of lines of code. If we compare with related work in terms of the expressiveness, only Spatial [121] and HeteroCL [127] can describe benchmarks that are not dense tensor computations. However, these two frameworks cannot achieve the same level of performance as SuSy. Another existing framework PolySA [40], which is based on a polyhedral compiler, can only handle algorithms without dynamic control flows such as SGEMM and Conv. The work proposed by Wei et al. [202] can generate highly efficient systolic arrays, but only for convolutional neural networks.

In the following, we provide more detailed case studies on SGEMM, Conv, and SW to compare SuSy and other frameworks.

Table 3.4: Performance impact of different spatial optimizations on a reduced SGEMM – We select a smaller input size ($512 \times 512 \times 512$) and also a smaller systolic array (8×8 with a vector length of 8 if applicable).

	+ Space-Time Transform	+ Vectorize	+ Isolate	+ Buffer & Others
#LUTs/ALMs	28%	21%	33%	24%
#DSPs	4.2%	34%	34%	34%
#BRAMs	16%	20%	16%	19%
Frequency (MHz)	250	203	225	259
Throughput (GFLOPs)	2.29	18.8	52.8	255
DSP Efficiency	7.2%	9.0%	23%	96%

Case Study: SGEMM – We first demonstrate how each spatial optimization affects the performance by using a smaller problem size ($512 \times 512 \times 512$) since, for large inputs, some of the design variants can be time-consuming for bitstream generation or do not even fit the device. In Table 3.4, we show not

only the performance numbers, but also the resource usage, frequency, and DSP efficiency. To calculate the DSP efficiency, we divide the throughput by theoretical throughput, which is defined as $\#DSP \times 2 \times Frequency/K$, where K is a target-dependent constant. We set $K = 5$ for VU9P and $K = 1$ for Arria 10.

From the table, we observe a trend of increasing throughput and DSP efficiency after each optimization step. We select the design with space-time transformation as our baseline, where we unroll the space loops and map them to PEs. After applying vectorization, a degradation of frequency occurs because the number of PEs increases. However, the increased computation power covers frequency degradation, and the throughput is consequently better. After I/O isolation, both frequency and DSP efficiency are improved, and the bottleneck now becomes the off-chip memory bandwidth. Introducing reuse buffers and other optimizations such as data scattering and gathering solve the issue (i.e., the DSP efficiency is now close to 100%). With all optimizations combined, the throughput is over 100× better than that of the baseline.

Table 3.5: Performance comparison for SGEMM.

	Spatial [121]	**HeteroCL [127]**	**PolySA [40]**	**T2S-Tensor [186]**	**Ninja [186]**	**SuSy**
LOCs	44	16	7	20	750	25
Systolic Aarray	No	Yes	Yes	Yes	Yes	Yes
Target FPGA	VU9P	VU9P	VU9P	Arria10	Arria10	Arria10
#LUTs/ALMs	36%	52%	49%	50%	54%	40%
#DSPs	12%	58%	89%	84%	84%	93%
#BRAMs	23%	45%	89%	51%	39%	32%
Frequency (MHz)	200	198	229	215	245	202
Throughput (GFLOPs)	2.4	246	555	549	626	547
DSP Efficiency	3.5%	79%	98%	99%	99%	96%

To further analyze the quality of results, we compare with other programming frameworks, including Spatial, HeteroCL, PolySA, and T2S-Tensor. We also compare with the Ninja implementation [186], which is written in HLS OpenCL by experts. We show the results in Table 3.5.

To begin with, there exists a stark difference in performance between the designs implemented without and with systolic arrays, namely, Spatial versus other frameworks. Naturally, there also exists a gap between general-purpose frameworks (i.e., HeteroCL) and those designed for generating systolic arrays (i.e., PolySA, T2S-Tensor, and SuSy). Finally, SuSy achieves similar throughput and DSP efficiency compared with other systolic array compilers specialized for certain application domains. Notably, SuSy achieves 87% of the throughput of the hand-written Ninja implementation, while only using 30× fewer lines of code (LOC). Moreover, if we compare on the same FPGA device (i.e., Arria 10), SuSy requires much less resource usage in ALMs and BRAMs mainly because we generate synchronous architectures with shift registers while T2S-Tensor and the Ninja manual design adopt asynchronous architectures with channels. As for PolySA, although it uses fewer LOCs and achieves similar performance, it is not as general as SuSy, as mentioned earlier.

Case Study: Convolutional Layer – We further compare the quality of results among frameworks that generate high-performance systolic arrays (Table 3.6). The design generated by Wei et al. [202] can achieve higher throughput at a higher frequency since their framework is designed explicitly for convolutional neural networks by mapping to manually optimized systolic array templates. However, this means they are not as general as SuSy. Moreover, under the same problem size and similar systolic array size, there also exists a reduc-

Table 3.6: Performance comparison for convolutional layer – The array shape is interpreted as width × height × vector length.

	PolySA [40]	Wei et al. [202]	SuSy
Target FPGA	VU9P	Arria 10	Arria 10
Systolic Array Shape	$8 \times 19 \times 8$	$8 \times 19 \times 8$	$8 \times 10 \times 16$
#LUTs/ALMs	- (49%)	- (57%)	150K (35%)
#DSPs	- (89%)	- (81%)	1,280 (84%)
#BRAMs	- (71%)	- (45%)	827 (30%)
Frequency (MHz)	229	253	220
Throughput (GFLOPs)	548	600	551
DSP Efficiency	98%	97%	98%

tion in resource usage similar to SGEMM. For PolySA, we can reach the same conclusion as the previous case study.

Table 3.7: Performance comparison for Smith-Waterman.

	Spatial [121]	HeteroCL [127]	SuSy
Target FPGA	VU9P	VU9P	Arria 10
#LUTs/ALMs	330K (28%)	111K (9.4%)	139K (33%)
#DSPs	0 (0%)	0 (%)	0 (0%)
#BRAMs	1,409 (65%)	470 (22%)	539 (20%)
Frequency (MHz)	200	152	250
Throughput (GCUPs)	0.11	1.25	7.89

Case Study: Smith-Waterman Algorithm – In this final case study, we compare the results with the two general-purpose frameworks (i.e., Spatial and HeteroCL). For Smith-Waterman, the typical performance metric is cell updates per second (CUPs), which can be derived by dividing the number of cells (i.e., the product of the lengths of the two input sequences) by the run time. Table 3.7 shows that SuSy achieves more than 5× performance improvement compared with HeteroCL and more than 70× improvement compared with Spatial. In addition, we are running at a much higher frequency because, with SuSy, we can

explicitly skew the iteration space by using space-time transformation to better pipeline the design.

3.5 Related Work

There exists a large body of literature on systolic array synthesis that enables programmers to generate systolic arrays at a high abstraction level [80, 40, 186, 127, 138, 178, 84, 202, 16].

Systolic array compilers with a push-button compilation flow – Compilers such as [84, 178, 16, 40] provide an end-to-end flow to generate systolic arrays without much user intervention. These compilers select the space-time transformation and other necessary optimizations based on built-in heuristics or automatic design space exploration. While delivering high productivity, these compilers usually fail to achieve high performance due to two major reasons: incomplete optimization directives and design space. Many optimizations are missing in the previous work. For example, vectorization and I/O isolation are missing in [178, 16]. Loop infinitization is missing in PolySA [40]. The missing of such optimizations could lead to sub-optimal designs. Apart from the compilers that target general systolic algorithms, there are also efforts attempting to generate domain-specific systolic arrays [73, 202]. For instance, Gemmini [73] and the framework proposed by Wei et al. [202] propose to generate efficient systolic arrays for deep neural networks (DNNs). Although both of them adopt configurable templates that generate high-performance systolic arrays, they are limited to DNNs.

Systolic array compilers with user-guided optimizations – In comparison,

works such as MMAlpha [80] and T2S-Tensor [186] take in user-specified optimizations. MMAlpha [80] is built upon UREs and lets programmers specify the space-time transformation. It supports both manual and automatic scheduling selection similar to the works in the previous category. However, it lacks the support for optimizations such as vectorization and reuse buffer insertion. A more recent work T2S-Tensor [186] incorporates richer optimizations compared with MMAlpha. It is the first work that inherits the principle to decouple the computation from the scheduling in designing systolic arrays. Nonetheless, T2S-Tensor can only generate systolic arrays for dense tensor kernels. In addition to those kernels, SuSy can generate systolic arrays for a much wider range of applications with UREs. Moreover, users can explore a larger design space with space-time transformation. Finally, by generating synchronous hardware, we can largely reduce the resource usage.

General HLS compilers – Beyond generating systolic arrays, there is also a plethora of work targeting implementing general applications on FPGAs [121, 127, 49, 44, 206]. However, experimental results show that there still exists a performance gap between such frameworks and dedicated systolic array compilers like SuSy.

CHAPTER 4

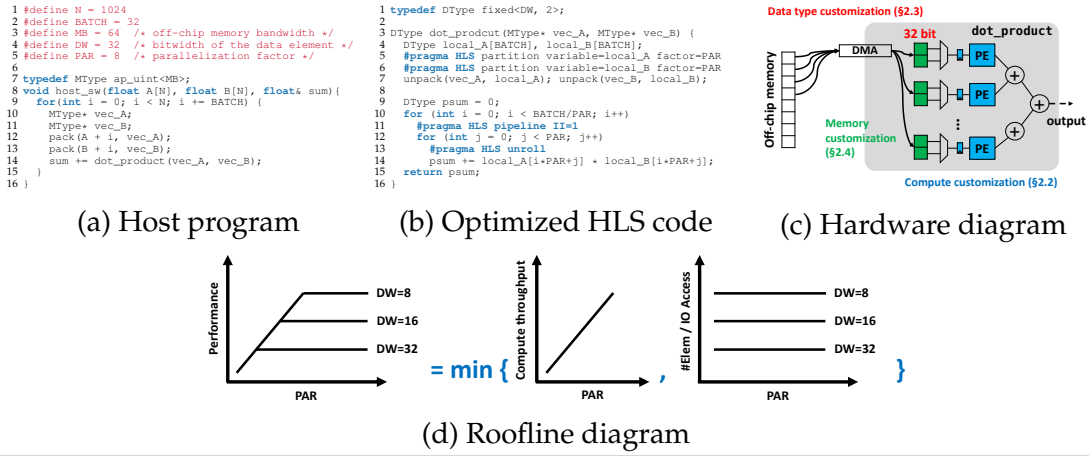
HETEROCL: A MULTI-PARADIGM PROGRAMMING INFRASTRUCTURE FOR SOFTWARE-DEFINED RECONFIGURABLE COMPUTING

HeteroCL is a Python-based DSL extended from TVM [25]. We choose TVM for the following reasons: (1) Python-based DSL provides programmers with a rich set of productive language features such as introspection and dynamic type system. (2) TVM is a tensor-oriented declarative DSL. Its declarative style is similar to TensorFlow [1], which compiles and executes the computation graph constructed by a programmer. This approach is beneficial for uncovering more high-level optimization opportunities in extracting parallelism and maximizing data reuse. (3) TVM inherits the idea of decoupling the algorithm specification from the temporal schedule, which is first proposed by Halide [173].

In addition to the features offered by TVM, HeteroCL further exposes heterogeneity in two dimensions: in hardware optimization techniques and programming paradigms. In the rest of the chapter, we first use a motivating example to show how HeteroCL abstracts different types of hardware customization and captures their interdependence. We then describe each customization in more detail. Finally, we present the imperative DSL in HeteroCL.

4.1 The Programming Model

In this section, we first use a motivating example to show how HeteroCL abstracts different types of hardware customization and captures their interdependence. We then describe each customization in more detail. Finally, we present



(a) Host program (b) Optimized HLS code (c) Hardware diagram

(d) Roofline diagram

Figure 4.1: Motivating example: dot product — This example demonstrates the interdependence between the parallelization factor `PAR` and the data bitwidth `DW`. By tuning them with different values, the performance of the whole design can be bounded by either the compute throughput (if `PAR` is too small) or the number of elements per I/O access (if `DW` is too large).

the imperative DSL in HeteroCL.

4.1.1 A Motivating Example

We use dot product operation as a motivating example that utilizes all three types of hardware customization. Figure 4.1a shows the host program, where we compute the dot product between vectors `A` and `B`. We offload function `dot_product` (L14) to FPGA for acceleration. Note that we need to batch the inputs due to FPGA on-chip resource limitation (L9). Before sending the batched inputs to the accelerator via DMA, we pack them to fully utilize the off-chip memory bandwidth (L12-13).

Figure 4.1b shows the optimized `dot_product` program implemented in HLS C++ code, where we apply all three types of hardware customization. First, we utilize data type customization by quantizing the data type of local

buffers `local_A` and `local_B` from floating to fixed-point type `DType` (L4). By reducing the bitwidth `DW`, we increase the number of elements per memory I/O access, which shortens the data transfer latency but introduces the trade-off between throughput and accuracy. Second, we apply memory customization by using `partition` pragmas to bank the buffers (L5-6). Finally, we apply compute customization to improve the performance by tiling the loop (L10, 12) and using parallelization pragmas (L11, 13). This results in `PAR` processing elements (PEs) computing the multiplication and accumulation in parallel. With larger `PAR`, we achieve higher compute throughput with the trade-off of more on-chip resource. Moreover, there exists an interdependence between compute and memory customization, where we need to match the number of PEs with the memory banking factor. In this specific example, we set both parameters to `PAR`. Finally, we show the hardware diagram in Figure 4.1c, where we illustrate each type of hardware customization.

In addition, it is important to balance the computation time and the data communication time to maximize the hardware efficiency. Specifically, we need to carefully balance the two components by tuning the data bitwidth `DW` and the number of PEs `PAR`. We increase the compute throughput by increasing `PAR`. We also increase the number of elements per I/O access by lowering `DW`. However, the final performance is bounded by the minimum of the two. We use the Roofline [203] diagram in Figure 4.1d to show the relation between `DW` and `PAR`.

Figure 4.2 shows how we apply compute customization in HeteroCL. First, we define the algorithm in Figure 4.2a, where we first import the HeteroCL module (L1), define the range to be sum up (L3), and use a vector/tensor-oriented *compute operation* `hcl.compute` to describe the multiplication and ac-

```
1 import heterocl as hcl
2 # algorithm
3 i = hcl.reduce_axis(0, BATCH)
4 psum = hcl.compute((1,), lambda x:
5   hcl.sum(A[i]*B[i], axis=i))
6 # customization primitives
7 s = hcl.create_schedule()
8 i, j = s[psum].split(i, PAR)
9 s[psum].pipeline(i)
10 s[psum].unroll(j)
```

(a) HeteroCL program

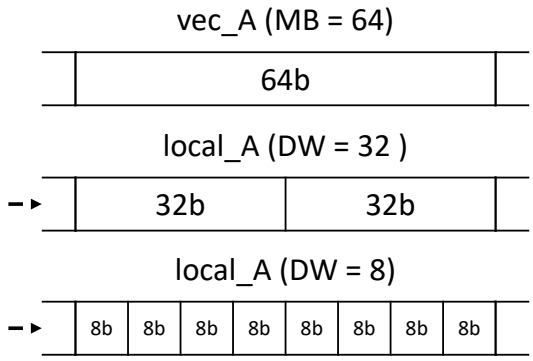
```
1 // algorithm only
2 for(int i = 0; i < BATCH; i++)
3   psum += A[i] * B[i];
4
5 // primitives applied
6 for(int i = 0; i < BATCH/PAR; i++)
7   #pragma HLS pipeline II=1
8   for(int j = 0; j < PAR; j++)
9     #pragma HLS unroll
10    psum += A[i*PAR+j] * B[i*PAR+j];
```

(b) Equivalent HLS code

Figure 4.2: Example of compute customization in HeteroCL.

```
1 # algorithm
2 vec_A = hcl.placeholder((128,), UInt(64))
3 local_A = hcl.unpack(vec_A)
4
5 # quantization scheme 1
6 s1 = hcl.create_scheme()
7 s1.quantize(local_A, Fixed(32, 30))
8
9 # quantization scheme 2
10 s2 = hcl.create_scheme()
11 s2.quantize(local_A, Fixed(8, 6))
```

(a) HeteroCL program



(b) Results after unpack

Figure 4.3: Example of data type customization in HeteroCL — Here we unpack the data sent from DMA `vec_A` to a local buffer `local_A`. The shape of the local buffer varies according to the quantization schemes. If we quantize `local_A` to a 32-bit/8-bit fixed-point buffer, each element of `vec_A` will be unpacked to two/eight elements in `local_A`.

cumulation operation that sums across `i` and returns a scalar (L4-5). The equivalent HLS code is shown in Figure 4.2b (L1-3). After that, we apply *compute customization primitives*, which are called scheduling functions in Halide/TVM, to a customization scheme created in separation of the algorithm (L7). The first primitive is a loop transformation primitive which splits loop `i` into a two-level nested loop `i` and `j` by a factor `PAR` (L8). We further apply two parallelization primitives that pipeline the outer loop `i` (L9) and unroll the inner loop `j` (L10). The equivalent code after applying customization primitives is in Figure 4.2b (L5-10). We can see that after applying primitives, we need to restructure the HLS code, while in HeteroCL, the algorithm specification stays unchanged.

```
1 # memory customization primitives
2 s = hcl.create_schedule()
3 s[local_A].partition(dim=1, factor=PAR)
```

(a) HeteroCL program

```
1 DType local_A[BATCH];
2 #pragma HLS partition variable=local_A factor=PAR
```

(b) Equivalent HLS code

Figure 4.4: Example of memory customization in HeteroCL.

Unlike existing DSLs, we further decouple the algorithm from data type customization. Figure 4.3 shows the results of applying decoupled quantization schemes in HeteroCL. In the algorithm specification, we unpack data transmitted from the 64-bit DMA `vec_A` to a local buffer `local_A` without specifying the implementation (Figure 4.3a L3). Then, we create a quantization scheme (L6) and quantize `local_A` to a 32-bit fixed-point buffer using a *quantization primitive* (L7). The result of unpacking is illustrated in Figure 4.3b. We can get a buffer with a different shape by quantizing to another bitwidth with a separate scheme (L10-11), while the algorithm stays the same.

Similar to decoupled compute and data type customization, we further decouple the algorithm from memory customization. In Figure 4.4a, we first create a customization scheme (L2). We then specify the *memory customization primitive* (L3). Equivalent HLS code is shown in Figure 4.4b.

Finally, Figure 4.5 shows the complete dot product kernel in HeteroCL, where we cleanly separate the algorithm specification (L1-8) from the hardware optimization specification (L14-32). We first apply data type customization to quantize the local buffers for better utilization of the off-chip memory bandwidth (L22-24). We then specify compute customization to tile and parallel the main computation for higher compute throughput (L26-29). Finally, we apply

```
1  # algorithm specification
2  def dot_product(vec_A, vec_B):
3    local_A = hcl.unpack(vec_A, name="local_A")
4    local_B = hcl.unpack(vec_B, name="local_B")
5    i = hcl.reduce_axis(0, BATCH, "i")
6    return hcl.compute((1,),
7      lambda x: hcl.sum(local_A[i] * local_B[i], axis=i),
8      name="psum")
9
10 # exploring a range of DW and PAR
11 for DW in [4, 8, 16, 32]:
12   for PAR in [4, 8, 16, 32]:
13     # key parameters that depend on data bitwidth (DW)
14     DType = hcl.Fixed(DW, DW-2)
15     MType = hcl.UInt(MB)
16     NPACK = BATCH*DW/MB
17
18     vec_A = hcl.placeholder((NPACK,), dtype=MType)
19     vec_B = hcl.placeholder((NPACK,), dtype=MType)
20     psum = hcl.placeholder((1,), dtype=DType)
21     # data type customization
22     sm = hcl.create_scheme([vec_A, vec_B, psum], dot_product)
23     sm.quantize([dot_product.vec_A,
24                  dot_product.vec_B], DType)
25     # compute customization
26     sl = hcl.create_schedule_from_scheme(sm)
27     i, j = sl[dot_product.psum].split(dot_product.i, PAR)
28     sl[dot_product.psum].pipeline(i)
29     sl[dot_product.psum].unroll(j)
30     # memory customization
31     sl[dot_product.local_A].partition(dim=1, factor=PAR)
32     sl[dot_product.local_B].partition(dim=1, factor=PAR)
33
34     f = hcl.build(sl)
35     # evaluate f and pick the best customization scheme
36     if QoR(f) > best_QoR:
37       best_QoR = QoR(f)
38       best_scheme = sl
```

Figure 4.5: Complete dot product example in HeteroCL — This example demonstrates how HeteroCL explores the interdependence between the data bitwidth `DW` and parallelization factor `PAR`.

memory customization that banks the buffers to match the on-chip memory bandwidth with compute throughput (L31-32). Moreover, we use a two-level loop to explore the interdependence between `DW` and `PAR` (L11-12). We then evaluate the built kernel `f` generated by our back end for each pair of `DW` and `PAR` (L34) and pick the best scheme for final FPGA synthesis (L36-38).

In the following sections, we describe the syntax and semantics of HeteroCL

Table 4.1: Compute customization primitives currently supported by HeteroCL.

Primitive	Description
	Loop transformation
`C.split(i, v)`	Split loop `i` of operation `C` into a two-level nest loop with `v` as the factor of the inner loop.
`C.fuse(i, j)`	Fuse two sub-loops `i` and `j` of operation `C` in the same nest loop into one.
`C.reorder(i, j)`	Switch the order of sub-loops `i` and `j` of operation `C` in the same nest loop.
`P.compute_at(C, i)`	Merge loop `i` of the operation `P` to the corresponding loop level in operation `C`.
	Parallelization
`C.unroll(i, v)`	Unroll loop `i` of operation `C` by factor `v`.
`C.parallel(i)`	Schedule loop `i` of operation `C` in parallel.
`C.pipeline(i, v)`	Schedule loop `i` of operation `C` in pipeline manner with a target initiation interval `v`.

for each type of customization in more detail.

4.1.2 Compute Customization

Compute customization improves the performance of a design by performing loop transformations and executing the computation in parallel. Similar to TVM [25], we decouple the algorithm specification from compute customization schemes. Table 4.1 lists compute customization primitives currently supported by HeteroCL. The primitives prevent programmers from using vendor-specific pragmas, which makes HeteroCL programs portable to different back ends.

Table 4.2: Data types currently supported by HeteroCL.

Data Type	Description
`Int(bw)`	Bit-accurate signed integer with `bw` bits.
`UInt(bw)`	Bit-accurate unsigned integer with `bw` bits.
`Fixed(bw, fr)`	Signed fixed-point type with `bw` bits, where there are `fr` fractional bits.
`UFixed(bw, fr)`	Unsigned fixed-point type with `bw` bits, where there are `fr` fractional bits.
`Float(bw)`	Floating-point type with `bw` bits, where `bw` could be 64 or 32.

Table 4.3: Quantization primitives currently supported by HeteroCL.

Primitive	Description
`quantize(t, d)`	Quantize a list of tensors `t` from floating to fixed point type `d` in the format defined in Table 4.2.
`downsize(t, d)`	Downsize a list of tensors `t` from integers with larger bitwidth to integers `d` with smaller bitwidth in the format defined in Table 4.2.

4.1.3 Data Type Customization

Quantized computation using low-bitwidth integers and/or fixed-point types is an essential technique to achieve efficient execution on FPGAs. To represent bit-accurate data types, traditional C-based HLS tools use templates such as `ap_int<>` and `ap_fixed<>`. Although this approach allows programmers to parameterize the bitwidths, they need to run a separate script to iterate through different quantization schemes. HeteroCL addresses this challenge by utilizing Python classes to represent the data types, which allows users to try out different quantization schemes within the same program. Table 4.2 lists the data types currently supported by HeteroCL.

Even with the bit-accurate data type support, it remains a challenge for most application developers to determine the right data types with the right bitwidth to achieve the best trade-off between accuracy and efficiency. To solve this, HeteroCL further decouples the algorithm specification from quantization

Table 4.4: Memory customization primitives currently supported by HeteroCL.

Primitive	Description
`C.partition(i, v)`	Partition dimension `i` of tensor `C` with a factor `v`.
`C.reshape(i, v)`	Pack dimension `i` of tensor `C` into words with a factor `v`.
`memmap(t, m)`	Map a list of tensors `t` with mode `m` to new tensors. The mode `m` can be either vertical or horizontal.
`P.reuse_at(C, i)`	Create a reuse buffer storing the values of tensor `P`, where the values are reused at dimension `i` of operation `C`.

schemes. HeteroCL provides two quantization primitives in Table 4.3, where `quantize(t, d)` quantizes a list of floating-point variables `t` to a fixed-point type `d` whose format is defined in Table 4.2. In addition, `downsize(t, d)` reduces the precision of a list of integer variables `t` to an arbitrary-bit integer type `d`. With `quantize` and `downsize`, programmers can explore the trade-off between performance/area and accuracy by tuning the bitwidths of variables in the algorithm. Note that this decoupled approach is well-suited for automated bitwidth-tuning frameworks based on autotuning or rule-based heuristics. Users can further provide domain-specific knowledge such as the numerical range or the distribution of a variable to quantization primitives to guide the bitwidth searching process.

4.1.4 Memory Customization

Accelerating applications on FPGAs usually requires a high on-chip memory bandwidth to match the throughput of massively parallel compute units. Without customized memory architectures such as reuse buffers, the memory bandwidth could become the main hindrance preventing designs from achieving better performance. We decouple the algorithm from the memory customization

Table 4.5: Spatial architecture macros currently supported by HeteroCL.

Primitive	Description
`C.stencil()`	Specify operation `C` to be implemented with stencil with dataflow architectures using the SODA framework.
`C.systolic()`	Specify operation `C` to be implemented with systolic arrays using the PolySA framework.

and provide a set of primitives (Table 4.4). Moreover, programmers can apply several customization primitives in a user-defined sequence, which is not possible using pragmas supported by existing HLS tools.

4.1.5 Mapping to Spatial Architecture Templates

Many popular workloads from image/video processing and machine learning domains can be realized in a highly efficient manner on FPGAs using spatial architectures such as systolic arrays [175, 202]. However, with the traditional C-based HLS methodology, it typically requires extensive code restructuring and the insertion of a right combination of pragmas to guide the tool to generate a high-performance spatial architecture. This tedious and error-prone process is one of the major barriers for the mainstream adoption of HLS for FPGA designs. HeteroCL addresses this deficiency by introducing a set of optimization macros that synthesize the code into highly efficient spatial architecture templates. Each of these macros consists of a combination of compute and memory customization primitives. As indicated in Table 4.5, we currently support stencil with dataflow architectures and systolic arrays, each of which is described in more detail as follows.

Stencil with Dataflow Architecture – Stencil computation is commonly seen

in many areas including image processing and numerical computing, where data elements are updated over a multidimensional grid according to some fixed, local patterns. HeteroCL incorporates the SODA framework [29], which synthesizes stencil patterns to a highly efficient dataflow architecture composed of reuse buffers and data streams.

Systolic Array – HeteroCL further provides efficient support for mapping to systolic arrays, which are widely used spatial architectures that consist a group of processing elements locally connected to each other [123]. Featuring local interconnects and modular designs, the systolic array architecture is highly scalable and can take advantage of the enormous amount of computation resources on modern FPGAs. It is particularly suitable for applications having perfectly nested loops with uniform dependency, such as matrix-matrix multiplication. However, it is a complex task to manually create systolic array designs on FPGAs. Recent research from Intel reports that it takes several to tens of months of human effort to implement a high-performance systolic array design, even with an HLS design entry like OpenCL [175]. Similar to stencil optimization, we introduce a `systolic` macro in the HeteroCL DSL to allow convenient mapping from tensor code to systolic array architectures.

4.1.6 Mixed Declarative and Imperative Programming

HeteroCL blends imperative programming with an embedded declarative, symbolic style for expressing tensor-based code. The idea is to combine the advantages of both styles — Imperative programming is general and flexible, while symbolic tensorized code exposes higher-level optimization opportuni-

```
BinOp   := + | - | * | / | % | & | ^ | >> | <<
BinEqOp := += | -= | *= | /=
CompOp  := > | >= | == | <= | = | < | !=
Expr    := Var | Tensor[Expr] | Number
           | not Expr | Expr BinOp Expr
           | Expr[Expr] # get bit
           | Expr[Expr:Expr] # get slice of bits
Cond    := Expr Comp Epxr | hcl.and_(*Cond) | hcl.or_(*Cond)
CondStmt:= hcl.if_(Cond) | hcl.elif_(Cond) | hcl.else_()
Stmt    := Tensor[Expr] = Expr | Tensor[Expr] BinEqOp Expr
           | Expr[Expr] = Expr # set bit
           | Expr[Expr:Expr] = Expr # set slice of bits
           | with CondStmt:
               Stmt
           | with hcl.for_(Expr, Expr, Expr) as Var:
               Stmt
           | # HeteroCL compute operations (e.g., compute)
           | # And more
```

Figure 4.6: Imperative DSL in HeteroCL — We provide equivalent semantics for commonly used expressions and statements in normal Python. We also support bit-level operations for bit-accurate data types. The imperative DSL highly resembles normal Python in that they use same indentations, same rules for variable scope, and similar keywords. This relieves new users from learning a whole new set of syntax and semantics.

ties when the code allows. The flexibility offered by imperative programming enables designers to implement algorithms with less-regular parallelism that cannot be efficiently represented with declarative programming (e.g., sorting algorithms). It also gives full control to programmers for specifying the algorithmic details.

Some of the existing Python-based DSLs use normal Python to support imperative programming, such as TVM [25] and Hot&Spicy [182]. This approach, however, has some drawbacks: (1) The normal Python semantic is too flexible to be FPGA synthesizable. (2) A designated parser/compiler must be built, which could be error-prone. Instead of using normal Python to support imperative programming, HeteroCL provides an imperative DSL listed in Figure 4.6. HeteroCL further extends existing compute operations (e.g., `compute`) and de-

Table 4.6: Compute operations currently supported by HeteroCL.

Operation	Description
`compute(s, f)`	Compute a new tensor of shape `s`. The value of each element in the new tensor is calculated according to lambda function `f`.
`update(t, f)`	Update each element of tensor `t` according to lambda function `f`.
`mutate(s, f)`	Write a for loop of shape `s` in vector code, where `f` is a lambda function describing the for loop body.

Table 4.7: Correspondence between HeteroCL primitives and Merlin C pragmas.

`unroll(i, v)` → `#pragma ACCEL parallel flatten factor=v` Partial unroll the target loop by factor `i` and fully unroll all its sub-loops.
`parallel(i)` → `#pragma ACCEL parallel` Wrap the body of loop `i` to a function and form a PE array.
`pipeline(i, v)` → `#pragma ACCEL pipeline` Wrap the body of loop `i` to a function and form a load-compute-store coarse-grained pipeline.

velops new operations for mixed-paradigm programming. We show examples of supported compute operations in Table 4.6.

4.2 Back-end Code Generation and Optimization

The HeteroCL framework has multiple back end supports including CPU and HLS flows targeting FPGA. Specifically, we extend the Halide IR used by TVM [173, 25] for our multi-paradigm programming model and customization primitives. The extended Halide IR serves as a unified representation for all back-end flows. In this section, we briefly summarize our FPGA back-end code generation flow.

General Back End – The HeteroCL compiler can generate a corresponding accelerator kernel in many languages, including HLS C/C++, OpenCL, and

Merlin C. Merlin C is an OpenMP-like programming model used by the Merlin compiler [44] from Falcon Computing Solutions. We choose the Merlin compiler as one of our back-end tools for two reasons. First, it leverages a small set of OpenMP-like pragmas to apply certain architecture structures by source-to-source C code transformation. Since Merlin pragmas share lots of similarity with HeteroCL customization primitives, it is relatively straightforward to integrate with HeteroCL. Second, the Merlin compiler generates both HLS C kernels and OpenCL kernels for FPGAs from the unified Merlin C source code.

Table 4.7 shows the correspondence between HeteroCL primitives and Merlin C pragmas. Since HeteroCL primitives and Merlin C pragmas mainly specify loop scheduling or memory organization, the implied architecture can be represented as a composable, parallel, and pipeline (CPP) architecture [49]. The authors in [49] have demonstrated that the CPP architecture can be applied to broad classes of applications with a good performance.

Stencil Back End – We incorporate the SODA framework proposed in [29] to implement stencil patterns with optimized dataflow architecture that minimizes the on-chip reuse buffer size. SODA takes in a lightweight DSL that describes the stencil compute patterns and design parameters. After the HeteroCL compiler identifies stencil patterns according to user-specified macros, it generates the proper DSL code to the SODA framework for hardware generation. In addition, hardware customization primitives such as loop unrolling and data quantization are also reflected in the SODA DSL as design parameters, which in turn guide the SODA framework for further optimization.

Systolic Array Back End – Similar to the stencil back end, our compiler analyzes the user-specified `systolic` macros and generates annotated HLS C++

Table 4.8: Evaluation results of benchmarks in HeteroCL — The speedup is over a single-core single-thread CPU execution.

Benchmark	Data Sizes & Type	Speedup	Back End
KNN Digit Recognition [222] Image classification	K=3 #images=1800 `uint49`	12.5	General
K-Means Clustering	K=16 #elem=320 × 32 `int32`	16.0	General
Smith-Waterman [200] Genomic sequencing	string len=128 `uint2`	20.9	General
Seidel [169] Image processing / linear algebra	2160 pixel × 3840 pixel `fixed16`	5.9	Stencil
Gaussian [169] Image processing	2160 pixel × 3840 pixel `fixed16`	13.2	Stencil
Jacobi [169] Linear algebra	2160 pixel × 3840 pixel `fixed16`	5.0	Stencil
GEMM Matrix-matrix multiplication	1024 × 1024 × 1024 `fixed16`	8.9	Systolic Array
LeNet Inference [133] Convolutional neural network	MNIST [65] `fixed16`	10.6	Systolic Array

code as an input to the PolySA framework [40], which further performs automated design space exploration that optimizes the systolic array architecture including the shape of it and the interconnection between PEs.

4.3 Evaluation

In this section, we evaluate the accelerators generated by HeteroCL. The platform we target is the AWS EC2 `f1.2xlarge` instance, which has 8 vCPU cores, 122GiB main memory, and a Xilinx Virtex UltraScale+™ VU9P FPGA. The default target frequency for this platform is 250 MHz.

We select several common FPGA benchmarks from a broad range of applications that are applied with either general, stencil, or systolic array back ends.

For the general back end, we have (1) KNN-based digit recognition, which is simplified from that of Rosetta [222], (2) K-means algorithm, and (3) Smith-Waterman [200]. For the stencil back end, we have (1) Gaussian, (2) Jacobi, and (3) Seidel. All of them are from Polybench [169]. For the systolic back end, we use (1) general matrix multiplication (GEMM) and (2) deep learning inference with LeNet model [133]. Among these benchmarks, KNN-based digit recognition, K-means, and Smith-Waterman need to be implemented with the HeteroCL imperative DSL.

Table 4.8 shows the benchmarks and the overall evaluation results. We run the baseline designs on one CPU core with a single thread. For the two systolic benchmarks, we are comparing our FPGA implementations with the GEMM function provided in Intel MKL [96] and a LeNet model optimized with TVM [25], respectively. We include memory transfer time (i.e., between DDR4 and FPGA) as part of the total run time. After applying proper customization primitives, we achieve up to 20.9× speedup for the benchmarks with the general back end. Moreover, we can achieve up to 13.2× and 10.6× speedup for benchmarks applied with stencil and systolic array back end, respectively.

4.4 Related Work

There exists a large body of work on HLS and domain-specific programming. In this section, we survey a small subset of representative efforts on C-based HLS, DSLs for hardware accelerator designs, and those that support decoupled algorithm and optimizations.

C-based HLS – HLS tools such as LegUp [20], Intel FPGA SDK [98], and

Xilinx Vivado HLS [210] allow developers to write FPGA designs in C/C++ and OpenCL, delivering higher productivity than traditional register-transfer-level (RTL) designs. The recently introduced Merlin compiler greatly simplifies the HLS design by applying source-to-source transformation to automatically generate optimized HLS-C or OpenCL programs [44]. However, to achieve good QoRs, developers are required to use various vendor-specific data types and pragmas/directives, rendering FPGA design with HLS less flexible and portable.

HeteroCL lifts the abstraction level of FPGA programming and provides developers with a systematic way to efficiently explore various trade-offs, making FPGA design more portable and productive.

DSLs for Hardware Accelerator Design – There is a growing interest in compiling programs written in high-level languages (e.g., Python, Scala) into reconfigurable hardware accelerators. Hot & Spicy compiles annotated Python code into HLS C/C++, where the annotations are translated into pragmas [182]. DHDL introduces a representation of hardware using parameterized templates in Scala that captures locality and parallelism information and compiles the representation into FPGAs and CGRAs [122]. Spatial extends DHDL by adding a set of low-level abstractions for control and memory [121]. However, in these DSLs the algorithm specification is tightly entangled with hardware optimizations, making design space exploration less productive.

HeteroCL decouples algorithm specification from hardware customization, and abstracts three important types of hardware customization into a set of customization primitives, enabling productive and systematic design space exploration. HeteroCL further offers additional macros in `stencil` and `systolic`

for efficient mapping to highly optimized spatial architecture templates.

DSLs with Decoupled Algorithm and Optimization – Most computing patterns in image processing and deep learning can be concisely described as nested loops in a declarative programming paradigm, as illustrated in a lot of DSLs [86, 148, 119, 201]. Halide first proposes to decouple the algorithm specification from the temporal schedule [173]. Tiramisu extends Halide by adding explicit communication, synchronization, and mapping buffers to different memory hierarchies [8, 175]. Jing Pu, et al. also extend Halide to support custom reuse buffers and support FPGAs and CGRAs as back end [171]. T2S extends Halide by decoupling the spatial schedule from the algorithm specification, which allows programmers to define systolic-array-like architectures [175]. TVM builds a deep learning compiler stack on top of Halide IR, supporting both CPUs and GPUs [25]. While the declarative programming paradigm in these DSLs is powerful, it cannot express applications beyond image processing and deep learning.

HeteroCL, as a multi-paradigm programming infrastructure, nicely blends declarative symbolic expressions with imperative code, and provides a unified interface to specify customization schemes for both declarative and imperative programs. This allows HeteroCL to support a broader range of applications.

More specifically, we list the major differences between TVM and HeteroCL as follows: (1) TVM extensively uses declarative programming to target deep learning applications, while HeteroCL supports mixed imperative and declarative programming to target general applications. (2) TVM tries to solve the optimization challenges mainly for CPUs and GPUs, while HeteroCL focuses on hardware customization for FPGA and incorporates advanced spatial architec-

ture templates. (3) TVM programs can target FPGAs as back end by using VTA, a programmable accelerator that uses a RISC-like programming abstraction to describe tensor operations [153]. On the other hand, HeteroCL programs are not limited to tensor operations. In addition, programmers can apply various hardware customization techniques with provided primitives while the hardware generated by VTA is pre-defined. (4) HeteroCL supports bit-accurate data types, which are not available in TVM. Furthermore, HeteroCL proposes to decouple the quantization scheme from algorithm specification.

CHAPTER 5

DRTRACE: ONLINE TRACED-BASED PROFILING TECHNIQUE FOR HETEROCL

As suggested by the roofline model in Figure 4.1d, the key to memory customization is to increase the operational intensity. In other words, we need to reduce the number of off-chip data accesses required by computing operations. One common practice is to explore on-chip buffering and caching techniques to improve the on-chip data reuse. This is also often a key enabler of low-latency and high-throughput custom compute engines. Unlike general-purpose processors that have a fixed and pre-designed multi-level cache system, FPGAs provide a fully customizable on-chip memory hierarchy that can harvest the bandwidth of a massive amount of distributed registers, LUT RAMs, block RAMs (BRAMs), and ultra RAMs (URAMs). One of the most common and effective optimizations is to customize the on-chip buffers based on the application-specific memory access patterns. The FPGA programmers may use different types of reuse buffers such as shift registers, line buffers, and window buffers.

However, since software programmers are more used to implicit memory orchestration such as caches on CPUs, it is non-trivial for them to explicitly design and manage custom memory hierarchy on FPGAs. For instance, with HeteroCL, even if it provides primitives such as `.reuse_at()` which largely simplifies the implementation of reuse buffers, a programmer may still face the following issues. First, without experience and expertise, it is difficult for a programmer to tell how and where to apply such a primitive. Second, the specified primitives might be invalid (e.g., not functionally correct). Therefore, there is an urgent need for an analysis method that resolves the above two challenges by

Table 5.1: Existing work on automated memory customization.

	Application	Agnostic to Input Language	Analysis Method
SODA [29]	Regular Data Reuse	No (DSL)	Compile-Time (Program Analysis)
Halide-HLS [171]	Regular Data Reuse	No (DSL)	Compile-Time (Program Analysis)
Lattice-Based Partitioning [37]	Regular Memory Banking	Yes	Compile-Time (Polyhedral Analysis)
TraceBanking [221]	Regular & Irregular Memory Banking	Yes	Run-Time (Trace Analysis)
DrTrace	Regular & Irregular Data Reuse	Yes (HeteroCL)	Run-Time (Trace Analysis)

providing programmers recommendations and validations.

There already exists a line of work that automatically generates custom memory hierarchy from different aspects. Table summarizes some of the existing works. For SODA [29] and Halide-HLS [171], both of them target regular data reuse patterns and generate custom reuse buffers via compile-time analysis. Moreover, both of them require programmers to use their own DSLs, which can limit input application domains. For example, SODA does not support image processing applications with a stride greater than one. There are other works focusing on different domains, such as lattice-based memory partitioning [37] and TraceBanking [221], which solves the memory banking problem. With TraceBanking, since it uses run-time analysis, it handles both regular and irregular memory banking problems. Moreover, both works are agnostic to the input language, which allows a wider application domain.

To this end, we propose DrTrace, a run-time trace-based profiling technique that provides automated validation and recommendation for application-specific data reuse on FPGAs. With run-time trace-based analysis, we handle

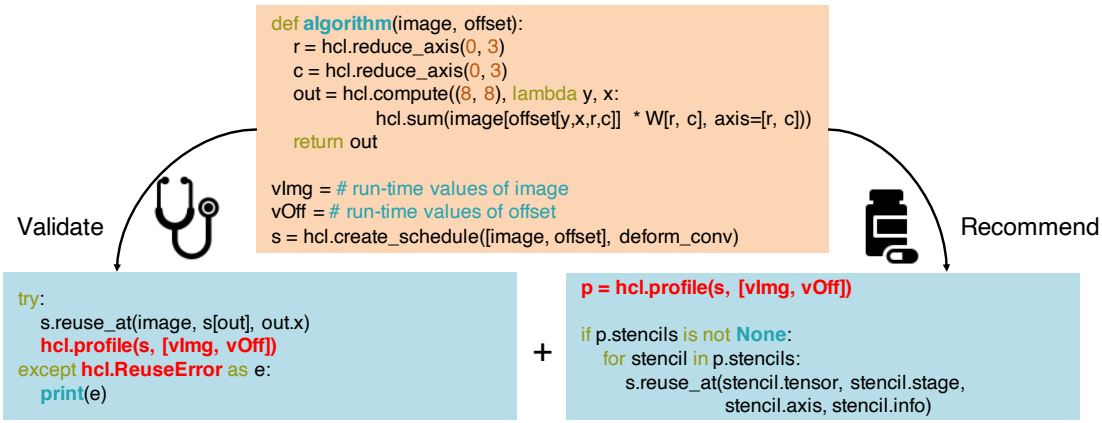


Figure 5.1: Overview

both regular and irregular data reuse patterns. To realize that, we integrate DrTrace with HeteroCL. The reason that we build on HeteroCL is two-fold. First, with decoupled primitives, users can apply the recommended data reuse primitives without modifying the algorithm. Second, with mixed-paradigm programming, HeteroCL provides a better design coverage compared with existing tools such as SODA and Halide-HLS.

Figure 5.1 shows the overview of the proposed analysis flow. Given an input program in HeteroCL, programmers can do two things with DrTrace. First, with a given schedule, DrTrace validates the specified data reuse. If the validation fails, users can analyze the errors using the returned exception. Second, if no schedule is provided, DrTrace recommends users with a set of data reuse primitives. Moreover, all analyses are performed in an online fashion. Our main technical contributions are as follows:

- To our knowledge, this is the first work to use run-time trace-based analysis for data reuse. With run-time analysis, DrTrace supports applications with both regular and irregular access patterns.
- By integrating with HeteroCL, DrTrace is agnostic to the input language,

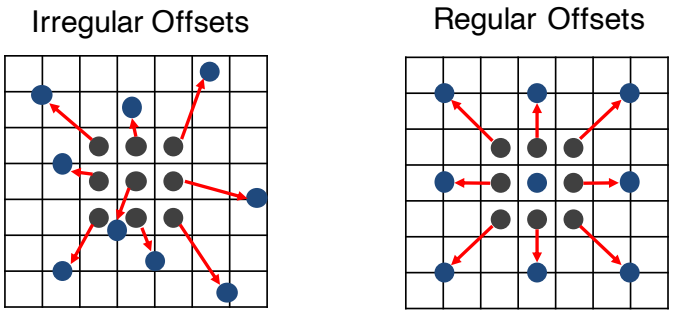


Figure 5.2: Deformable convolution

which opens to a wide range of applications. With mixed-paradigm programming provided by HeteroCL, programmers can describe any access patterns.

- We perform the trace-based analysis online. In other words, DrTrace does not store the entire trace generated from the analysis, which makes us memory efficient. In addition, by combining online analysis with exception handling, users can get the errors early without the need of running through the entire program.

The rest of the chapter is organized as follows, in Section 5.1, we use a motivational example to show the need of a run-time analysis technique. Then, we formally define the problem in Section 5.2. After that, in Sections 5.3 and 5.4, we describe the profiling algorithm and the corresponding hardware architecture. Finally, we evaluate the proposed technique in Section 5.6.

5.1 Motivational Example: Deformable Convolution

In this section, we use deformable convolution [54] as a motivating example. Figure 5.2 shows the concept of a deformable convolution. Unlike regular con-

```
1 int input[10][10];
2 int output[8][8];
3 int weight[3][3];
4 Loop C:
5 for (int y = 0; y < 8; y++)
6   for (int x = 0; x < 8; x++)
7     for (int r = 0; r < 3; r++)
8       for (int c = 0; c < 3; c++)
9         oy = offset_y[y+r][x+c]
10        ox = offset_x[y+r][x+c]
11        output[y][x] += input[oy][ox] * weight[r][c];
```

Figure 5.3: Deformable convolution

volution that samples the pixels from the input feature map with a fixed pattern, deformable convolution samples the input pixels with additional offsets that can be learned during training. Such a modification allows the convolution layer to capture objects with different scales and angles, which lead to better precision. Figure 5.3 shows the algorithm of deformable convolution. As can be seen, the final pixels to be convolved (L11) are determined by a set of pre-trained offsets (L9-10). This information can only be gathered at run time.

However, since there is no constraint on the offsets and the values vary pixel by pixel, it is extremely unfriendly to hardware due to the irregular memory accesses. To solve that, the authors of CoDeNet [91] propose to introduce some restrictions to the offsets by making them regular. An example is shown in Figure 5.2 right, where the pixels after applying the offsets are now on the edges of a square. This not only saves the number of parameters but more importantly, introduces data reuse opportunities. CoDeNet shows that the constrained offsets have limited effect on the accuracy. However, since whether the offsets are regular or not can only be observed at run time, traditional compilers using static analysis cannot exploit the data reuse opportunity brought by regular offsets. In other words, we need a run-time analysis tool to discover the potential data reuse.

5.2 Problem Formulation

In this section, we provide the definitions and formally define the problem of online trace-based profiling for data reuse. To make the problem and analysis simpler without loss of generality, we first preprocess the input program by linearizing the memory addresses and canonicalizing the loops. Then, we define several important concepts before formulating the problem.

Memory Address Linearization. Given an N-dimensional tensor T with shape $(T_0, T_1, ..., T_{N-1})$, a memory address $\vec{x}$ can be linearized to a scalar x, where

$$x = x_0 + x_1 T_1 + ... + x_{N-1} \prod_{d=0}^{N-2} T_d$$

For instance, given a 4-dimensional tensor with shape $(3, 8, 6)$ and a memory address $(1, 3, 5)$, the linearized address is

$$5 + 3 * 6 + 1 * 8 * 6 = 71$$

Loop Canonicalization. Given a loop nest L, where at each level, the loop variable v starts from b, increases s per iteration, and ends at $b+e$. We can canonicalize the loop by deriving a new variable v', where it starts from 0, increases 1 per iteration, and ends in $\lfloor e/s \rfloor$.

Definition 1. ***Access Function:*** *Given a tensor T, a loop L, and an iteration $\vec{i}$, we define the access function $f(T, \vec{i})$ as the set of linearized memory accesses being read from T under iteration $\vec{i}$.*

Example: To better illustrate the definitions, we use the following example across this section. The example is a deformable convolution with square off-

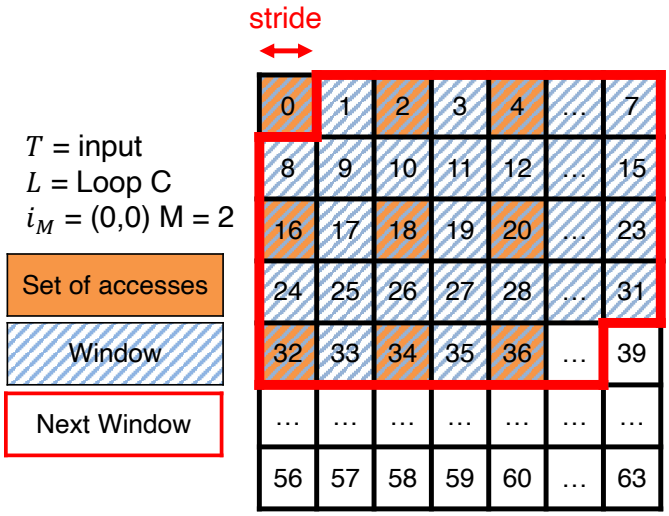


Figure 5.4: Example of windows and strides.

sets. We show the code in Figure 5.3 and the corresponding input accesses in Figure 5.4.

Given tensor `input`, loop `C`, and iteration $\vec{i} = (1, 3, 2, 1)$, the only memory address is $(3, 4)$ or tensor `input`, which after linearization is 34. Thus, $f(\texttt{input}, \vec{i}) = \{34\}$.

Definition 2. ***Partial Iteration:*** *Given a N-level loop nest L and an iteration* $\vec{i} = (i_0, i_1, ..., i_{N-1})$*, a partial iteration* $\vec{i}_M$ *with respect to loop level M is a subset of iteration* $\vec{i}$*, where* $\vec{i}_M = (i_0, i_1, ..., i_{M-1})$*. When* $M = N$*, the partial iteration is the same as a regular iteration.*

Definition 3. ***Partial Access Function:*** *Given a tensor T, a loop L, and a partial iteration* $\vec{i}_M$*, we define the partial access function* $f_p(T, \vec{i}_M)$ *as the set of linearized memory accesses being read from T in iteration* $\vec{i}_M$*, which can be defined recursively using*

$$f_p(T, \vec{i}_M) = \cup_{\forall \vec{i}_{M+1}} f_p(T, \vec{i}_{M+1})$$

When $M = N$*,* $f_p(T, \vec{i}_M) = f(T, \vec{i})$*.*

Example: Let $M = 2$, the partial iteration $\vec{i}_2$ becomes $(1, 3)$. Now we can derive $f_p(\texttt{input}, \vec{i}_2)$ recursively by

$$f_p(\texttt{input}, \vec{i}_2) = f_p(\texttt{input}, (1, 3, 0)) \cup f_p(\texttt{input}, (1, 3, 1)) \cup f_p(\texttt{input}, (1, 3, 2))$$

For each partial access function, we can also derive it recursively by

$$f_p(\texttt{input}, (1, 3, 0)) = f(\texttt{input}, (1, 3, 0, 0)) \cup f(\texttt{input}, (1, 3, 0, 1)) \cup f(\texttt{input}, (1, 3, 0, 2))$$

Note that now we can directly evaluate the access functions.

$$f_p(\texttt{input}, (1, 3, 0)) = \{13, 14, 15\}$$

Similarly,

$$f_p(\texttt{input}, \vec{i}_2) = \{13, 14, 15, 23, 24, 25, 33, 34, 35\}$$

In other words, calculating the partial access function is similar to unrolling the inner loops.

Definition 4. ***Reuse Distance:*** *Given a tensor T, a loop L, and a partial iteration $\vec{i}_M$, the reuse distance is defined as*

$$max(f_p(T, \vec{i}_M)) - min(f_p(T, \vec{i}_M))$$

Example: This one is straightforward, the reuse distance is $35 - 13 = 22$.

Definition 5. ***Stride:*** *Given a tensor T, a loop L, and two consecutive partial iterations $\vec{i}_M$ and $\vec{i}'_M$, the stride is defined as*

$$min(f_p(T, \vec{i}'_M)) - min(f_p(T, \vec{i}_M))$$

Example: First, we find the next iteration, which is $(1, 4)$. Then, we derive $f_p(\texttt{input}, (1, 4)) = \{14, 15, 16, 24, 25, 26, 24, 35, 36\}$. Finally, we calculate the difference between the minimum value of each. In other words, stride is $14 - 13 = 1$.

Definition 6. ***Intra-Loop Stride:*** *Given a tensor T, a loop L, and two consecutive partial iterations $\vec{i}_M$ and $\vec{i'}_M$, if $\forall 0 \leq k \leq M-2$, $i_k = i'_k$, we call the stride an intra-loop stride.*

Definition 7. ***Inter-Loop Stride:*** *Given a tensor T, a loop L, and two consecutive partial iterations $\vec{i}_M$ and $\vec{i'}_M$, if $\forall K \leq k \leq M-1$, $i'_k = 0$, we call the stride an inter-loop stride from level K.*

Example: If $\vec{i}_2 = (1,3)$ and $\vec{i'}_2 = (1,4)$, the stride we derive is an intra-loop stride. However, if $\vec{i}_2 = (1,9)$ and $\vec{i'}_2 = (2,0)$, we are calculating the inter-loop stride. In this case, it becomes 3 instead of 1.

Definition 8. ***Relative Accesses:*** *Given a tensor T, a loop L, and a partial iteration $\vec{i}_M$, we define the relative accesses to be a set*

$$\{A - min(f_p(T, \vec{i}_M)) | \forall A \in f_p(T, \vec{i}_M)\}$$

Example: If $\vec{i}_2 = (1,3)$, the calculation of relative accesses is also straightforward, which is $\{0,1,2,10,11,12,20,21,22\}$.

Definition 9. ***Reuse Tuple:*** *A reuse tuple is composed of three elements: a tensor, a loop, and the loop level. The loop level must not exceed the maximum level of the loop.*

Definition 10. ***Stencil Accesses:*** *Given a reuse tuple (T, L, M), the memory access pattern with respect to this reuse tuple is called* ***stencil*** *if and only if the reuse distance stays the same and the strides are always positive for all partial iterations $\vec{i}_M$. Moreover, if all intra-loop strides, inter-loop strides from all levels, and relative accesses stay the same for all partial iterations, we call it* ***perfect stencil****.*

Definition 11. ***Memory Trace:*** *A memory trace records all linearized memory accesses. For each access, we also record the following information: 1) The tensor the*

access comes from, which must be a read access, and 2) The loop and the iteration that invoke the access.

With the above definitions, we can now formally define the trace-based profiling problem.

Problem: Given a memory trace, find all (tensor, loop, loop level) tuples whose access pattern is a perfect stencil.

5.3 Profiling Algorithm

In this section, we describe the profiling algorithm in detail and also analyze its time and space complexity. There are two major challenges we need to solve. First, we need to co-design the hardware while analyzing the traces. In other words, in addition to verifying and finding the candidates for perfect stencils, the analysis should also collect information for generating the hardware. Second, since trace-based profiling is a run-time analysis, the analysis can only make a conclusion after processing the entire trace. If no optimization is made, the time and space complexity grows linearly to the size of the trace, which is not scalable.

To solve the first challenge, we need to store all necessary information needed for hardware generation during the analysis. For the second challenge, to reduce time complexity, we preprocess the input program via memory address linearization and loop canonicalization, while the space complexity can be reduced via online analysis. To be more specific, we analyze the trace at the same time it is produced. Figure 5.5 shows the algorithm for inferring and

Algorithm 1 InferAndCheckPerfectStencil

```
Input  : trace – memory trace;
         specs – user specified reuse tuple.
Output: recomms – a list of recommendations.
for (T, L, i) in trace do
    /* T: Tensor; L: N-level loop; i:  Iteration          */
    if (T, L, i) not in recomms and (T, L, i) not in specs then
        recomms ∪= (T, L, i)
    end
    for M ← 0 to N do
        accesses ← f_P(T, i_M)
        reuse_dist ← max(accesses) − min(accesses)
        /* In practice, we need to calculate both intra-loop
        strides and inter-loop strides                     */
        if not first iteration then
            stride ← min(accesses) − prev_min
            relative_accesses ← accesses − min(accesses)
            if reuse_dist or stride or relative_accesses changed then
                if (T, L, i) in specs then
                    ERROR! Specified reuse tuple is invalid!!
                else
                    recomms −= (T, L, M)
                end
            end
            prev_min ← min(accesses)
        end
    end
end
return recomms
```

Figure 5.5: The algorithm for perfect stencil inference and validation in trace-based profiling.

checking perfect stencils. For simplicity, we do not show the analysis for different strides (i.e., intra-loop and inter-loop strides). We also do not show the extra hardware information that needs to be stored.

The algorithm takes in two inputs: the memory trace generated by the input program and a specification, which contains a list of reuse tuples specified by

users. The output is a list of recommended reuse tuples. Since the algorithm is performed online, as soon as we get a new line of trace, we perform the analysis. At first, we assume all possible reuse tuples are perfect stencils. In other words, the recommendation list contains all possible reuse tuples. Then, according to the input trace, if a tuple no longer satisfies the conditions of perfect stencil, we remove it from the recommendation list (or we return an error if the tuple is in the user specification).

For time complexity, for each line of trace, we only do a fixed amount of computation per loop level (e.g., comparison for finding the minimum access and subtraction for finding the reuse distance). Thus, the final time complexity is linear to the number of lines of trace (i.e., the total number of accesses) times the maximum possible loop level. Since the maximum loop level is usually a small integer, the time complexity is $O(N)$, where N is the number of total accesses in a program. As for space complexity, we only need to store a constant number of parameters for each reuse tuple. Thus, we can describe the space complexity as $O(T \times L \times M)$, where T is the number of tensors in a program, L is the number of loop nests, and M is the maximum loop level. In practice, each loop might only be associated with a few tensors. Thus, the space complexity can be approximated by just $O(L)$.

5.4 Hardware Architecture

For the hardware, we generate data flow pipelines with line buffers, which is similar to the architecture proposed by SODA [29]. An example is shown in Figure 5.6, where the generated hardware is shown on the top of the figure. The

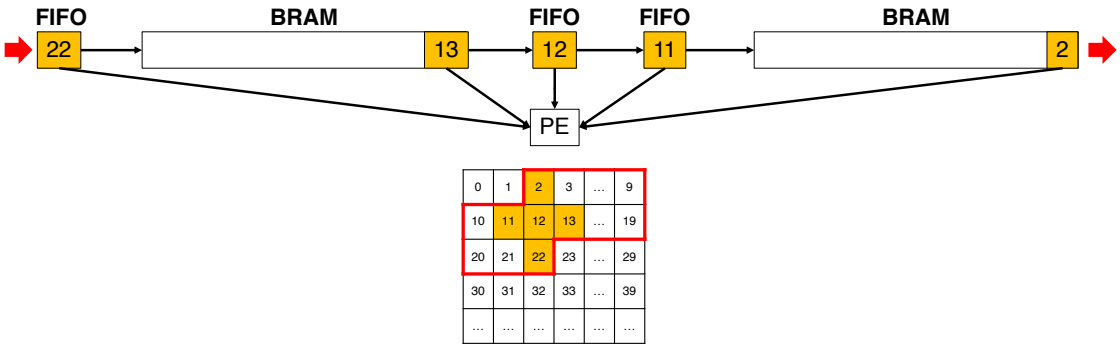


Figure 5.6: Hardware Architecture

hardware is composed of five FIFOs (two of them are implemented as BRAMs), and one PE. The arrows denote the wire connection. For each cycle, a pixel is read from the input and sent to the leftmost FIFO. Meanwhile, the old data in the FIFO is written to the connected FIFO/BRAM and is taken in by the PE at the same time. The same operation is performed by all other FIFOs/BRAMs.

The major difference is how the hardware is executed. More precisely, how each line buffer is updated. With SODA, it updates the buffers by sending one element from the input tensor per cycle. The major problem is that there might be redundant computations (e.g., at the boundary of the input tensor). To resolve that, the host needs to prune out the redundant values. However, with trace-based analysis, we update the buffers according to the inter-loop and inter-loop strides, which avoids the redundant computations.

5.5 Integration with HeteroCL

The trace-based profiling technique can be easily integrated with the existing HeteroCL programming framework. We only need to introduce a new API `profile()`, which takes in two inputs: the run-time values of inputs and the

schedule. It checks whether the given schedule is valid. Moreover, it also returns a list of recommended reuse tuples. An example is shown in Figure 5.7.

```
1  import heterocl as hcl
2
3  def deform_conv(image, offset, weight):
4      r = hcl.reduce_axis(0, 3)
5      c = hcl.reduce_axis(0, 3)
6      out = hcl.compute((8, 8), lambda y, x:
7                 hcl.sum(image[offset[y+x, r+c]]
8                         * weight[r, c], axis = [r, c]))
9      return out
10
11 s = hcl.create_schedule([image, offset, weight], deform_conv)
12
13 vImg = # run-time values of image
14 vOff = # run-time values of offset
15 vWgt = # run-time values of weight
16 p = hcl.profile(s, [vImg, vOff, vWgt])
17
18 if p.stencils is not None:
19     for stencil in p.stencil:
20         s.reuse_at(stencil.tensor,
21                    stencil.loop,
22                    stencil.level,
23                    stencil.info)
```

Figure 5.7: Deformable convolution in HeteroCL augmented with trace-based profiling.

In this example, we first define the deformable convolution algorithm in Lines 3 to 9. After that, we create the schedule in Line 11. Although we do not introduce any user-specified data reuse, users can easily introduce them by using `s.reuse_at()`. After that, we prepare the run-time values for each input tensor in Lines 13 to 16. Then, we feed those values along with the schedule to the profiling API in Line 16. Next, we can optimize the program according to the returned recommendations. In Line 18, we first check if there is any candidate. Then, for each reuse tuple, we apply the modified `reuse_at` primitive. The first three arguments to the primitive are the same as the one introduced in Chapter 4. The only difference is the last argument, where we provide additional information for generating the hardware. After that, we can move on with the existing HeteroCL compilation flow.

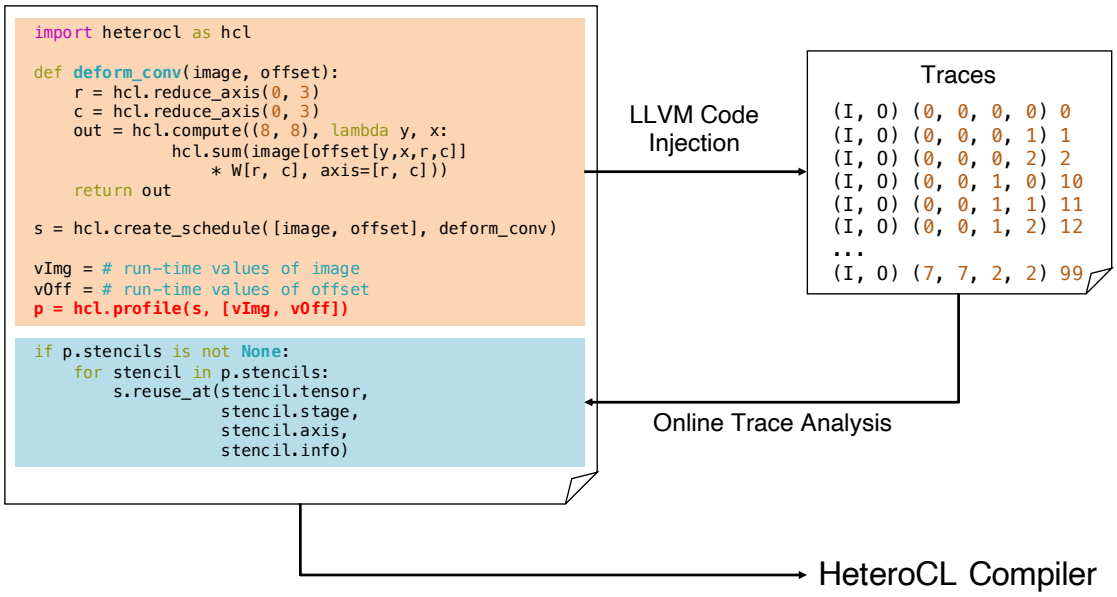


Figure 5.8: Integration of DrTrace and HeteroCL.

Figure 5.8 shows an overview for the implementation details. To generate the trace, we inject new IR nodes to the IR from the original program. The traces are produced when targeting LLVM as the back end. We implement the profiling algorithm as an independent program. To perform online analysis, we create a pipe between the HeteroCL program and the analysis program. In other words, the traces are directly written to the pipe, which is read by the analysis program. Similarly, the results are also written to a separate pipe, which is read by the HeteroCL program. Finally, we reorganize the results before returning them back to the users.

5.6 Evaluation

In this section, we perform two evaluations. First, we evaluate the time complexity of the profiling algorithm. Next, we evaluate the effectiveness of data

Table 5.2: Evaluation on the run time of the profiling algorithm.

#Accesses	Total Run Time (s)	Trace Generation	Trace Parsing	Trace Analysis
16×16×16×7 (-)	0.190 (-)	0.103 (54%)	0.057 (30%)	0.030 (16%)
32×32×32×7 (8×)	1.073 (5.6×)	0.703 (65%)	0.247 (23%)	0.123 (12%)
64×64×64×7 (8×)	8.795 (8.2×)	5.114 (58%)	2.656 (30%)	1.025 (12%)
128×128×128×7 (8×)	71.61 (8.1×)	40.36 (56%)	24.05 (33%)	7.200 (11%)

reuse. With the deformable convolution, we show that we can indeed capture the application-specific data reuse. We also compare it with SODA [29], a state-of-the-art stencil compilation framework. We target Xilinx Alveo U280 Accelerator Card and use Xilinx Vitis 2020.2 to get post-synthesis results and synthesize the bitstream.

5.6.1 Scalability

To evaluate the scalability of the profiling algorithm, we measure the run time it takes to analyze the entire trace and mutate the problem size. We show the results in Table 5.2, where we select Jacobi 3D as the problem. The first column shows problem size, which is the number of accesses. We also show the increment in the parenthesis. In the second column, we show the total run time in seconds, where the increment is also shown in the parenthesis. From the table, we can observe that the run time indeed increases linearly to the problem size.

We also break down the run time into three parts to further analyze which part is the most time consuming. From the results, we see that we spend most of the run time on producing trace and parsing the trace, which is as expected.

Table 5.3: Evaluation on deformable convolution with and without data reuse.

	II	#LUTs	#FFs	#DSPs	#BRAMs	Latency (ms)	Speed Up
No Optimization	9	4,375	8,528	0	9	652.4	1.0
With Data Reuse	1	5,484	9,884	0	28.5	33.45	19.5

5.6.2 Case Study: Deformable Convolution

In this example, we demonstrate the effectiveness of data reuse by showing the speed up before and after introducing reuse buffers. We use deformable convolution with rounded offsets as a case study. As can be seen in Figure 5.7, the offset is an input to the program. In other words, static analysis cannot make any assumption on the offset, which results in hardware without any reuse buffer. We show the results in Table 5.3.

The first thing to be noticed is the change in II. Without applying data reuse, the II is much larger. After introducing data reuse with the profiling technique, we reduce the II to 1. Moreover, by introducing only slightly more resources, we can achieve a 19.5× speed up.

5.6.3 Comparison with SODA

Finally, we compare our results with SODA in Table 5.4. We select three benchmarks, which are all perfect stencils. First of all, since the SODA DSL is incapable of describing applications with a stride greater than one, it cannot handle 2D maximum pooling. For 2D convolution and 3D Jacobi, we run faster than SODA. There are two reasons. First, as mentioned in Section 5.4, SODA may introduce redundant computations at the margins of the inputs. Second, for the

Table 5.4: Performance and resource comparison with SODA [29]. Note that the resource usage shown here only includes the compute kernels.

	Design	#LUTs	#FFs	#DSPs	#BRAMs	#URAMs	Latency (ms)
2D Convolution 4098×4098	SODA	1,693	2,134	0	0	2	59.8
	Ours	1,738	1,768	0	26	0	57.0
3D Jacobi 258×258×258	SODA	2,452	3,673	12	129	0	73.6
	Ours	1,578	1,354	0	193	0	57.2
2D Max Pool 4096×4096	SODA	N/A					
	Ours	599	572	0	0	1	56.4

Jacobi benchmark, some of the SODA-generated dataflow pipeline modules do not reach an II of one, which might be a bug of the SODA compiler.

As for the resource comparison, we only show the resource usage for compute kernels. The reason is that SODA implements the I/O interface between host and accelerator directly using RTL, which is more resource efficient compared with direct synthesis with HLS. As for the comparison for compute kernels, from Table 5.4, we can see that we use fewer LUTs and FFs but use more BRAMs. This may come from the implementation details of the dataflow pipelines with line buffers. With SODA, the line buffers are implemented as arrays while in this work, we implement the line buffers with `hls::stream`.

CHAPTER 6

CONCLUSION

There is an increasing need for FPGA-based computation recently from both cloud and edge computing. However, programming FPGA remains challenging because of the low productivity when achieving high performance at the same time. The main reason is that for existing programming frameworks, the hardware customizations are closely entangled with the algorithm, which makes the programs/designs less portable and less maintainable.

To tackle the challenges, this dissertation presents two programming frameworks targeting FPGAs and decouple essential hardware customizations from the algorithm. With the presented frameworks, we show that the programs can achieve the same level of performance while improving the productivity and portability at the same time. Moreover, we introduce trace-based profiling techniques that help validate and recommend application-specific data reuse, which further improves productivity and performance.

6.1 Dissertation Summary and Contributions

In this dissertation, we first introduce the essential techniques for building a high-performance FPGA accelerator, including compute, memory, and data type customizations. We also summarize a rich spectrum of work on programming abstractions and optimizing compilers that provide different trade-offs between performance and productivity.

After that, we present SuSy, a programming model for productively building high-performance systolic arrays. With SuSy, programmers can describe

any systolic algorithm with UREs and also efficiently explore different spatial optimizations, such as space-time transformation and reuse buffer insertion. Moreover, we provide an end-to-end compilation flow targeting Intel FPGAs. Experiment results show that we can indeed achieve high performance on not only dense tensor kernels but also bioinformatics benchmarks. We believe SuSy can bridge the gap between productivity and quality of the development of systolic arrays on FPGAs. SuSy is now available on GitHub.

Next, we present HeteroCL, a multi-paradigm programming infrastructure for heterogeneous platforms integrating CPUs and FPGAs. HeteroCL not only provides a clean abstraction that decouples the algorithm from compute/data customization, but it also captures the interdependence among them. Moreover, HeteroCL incorporates spatial architecture templates including systolic arrays and stencil with dataflow architectures. We believe HeteroCL can help developers to focus more on designing efficient algorithms rather than being distracted by low-level implementation details. HeteroCL is now publicly available on GitHub. Moreover, industry companies such as Intel have started to develop their tools using HeteroCL.

Finally, we introduce DrTrace, a trace-based profiling technique for application-specific data-dependent data reuse. By combining with HeteroCL using a new API `profile()`, the profiling technique helps users validate specified data reuse schemes and recommend potential data reuse if not specified.

6.2 Future Directions

Automation to Generate Customization Primitives. So far both HeteroCL and SuSy rely on user's inputs for specifying the customization primitives. However, as we have seen in Chapter 5, user-specified primitives may not always be valid or may have better usage. Thus, it is vital to have a way to automatically generate customization primitives which are not only valid but also help improve the performance. By automatically generating the primitives, programmers can further focus on the design of the algorithm itself, which also improves productivity.

There are several promising approaches. We have already seen one example in Chapter 5, where we profile the input program and exploit the possible optimization opportunities either with run-time or compile-time analysis. However, sometimes the solution space for all possible primitives is too large. In this case, we can introduce other techniques such as autotuning to help prune out less effective solutions.

As for the applications, in addition to regular stencils as we have shown in Chapter 5, there still exist cases that are not strictly stencils. In these cases, we need to generate more flexible hardware architectures such as caches. One example is the histogram, which we show the code in Figure 6.1. To show the effectiveness of a custom cache, we compare the resource usage and performance with and without cache in Tables 6.1 and 6.2. To conduct this simple experiment, we set the size of input data to be 2^{20}, with a direct-mapped cache having 32 lines, where each line can store four 32-bit numbers.

From the table, we can see that by using slightly more resources, we can

```
void top(int* feature, int* histogram) {
    for (int i = 0; i < N; i++) {
        #pragma HLS pipeline
        m = feature[i];
        histogram[m] = histogram[m] + 1;
    }
}
```

Figure 6.1: Histogram

Table 6.1: Resource comparison for histogram.

	II	#LUTs	#FFs	#BRAMs	Frequency
Without Cache	141	4,603	8,711	9	300
With Cache	2	5,304	10,558	12	300

largely reduce the II from 141 to 2. Depending on the access patterns (e.g., the number of unique elements from the input data set), we can achieve as high as 58× speedup. The access patterns can be easily analyzed by our online trace-based analysis. The major issue here is the larger design space, where we have many choices on what types of caches to generate (e.g., set-associative or direct-mapped, least frequently used or least recently used).

In addition to generating caches, trace-based analysis is capable of performing more analysis. Following we list some examples. First, we can analyze the access pattern and generate multi-buffers such as a double buffer, which is useful for automatically generating the I/O modules for SuSy. Second, we can analyze whether the consecutive compute kernels can be connected with FIFOs or not by checking the access order. Similar to what DrTrace can do now, in addition to validation, we can also provide recommendations for potential FIFO connections. Finally, we can use the FIFO information to estimate the latency and throughput of a design, which can be useful to auto-tuning and auto-scheduling tools.

Table 6.2: Run time comparison for histogram.

#Unique Elements	#Misses	Run Time (ms)
Without Cache		513
32	8	8.9
128	32	9.2
129	12,944	9.3
256	524,296	9.7
4096	1,040,483	14.2

Verification and Debugging. Once we generate the customization primitives automatically from profiling and autotuning, it is important to validate the generated primitives. It would be even better if we can pinpoint the incorrect primitives. In other words, we need a verification and debugging flow, which can benefit both HeteroCL and SuSy.

For verification, we have already seen how DrTrace uses run-time profiling analysis to validate custom data reuse. There are more ways we can perform the verification and debugging. One naive way of performing verification is via software simulation. In other words, given a set of test inputs, we compare the results of an input program before and after applying the primitives. With this method, we need to carefully design the test inputs to guarantee high coverage. One major issue with simulation is that usually it is not scalable. In this case, for some primitives such as loop transformations, we can rely on static analysis such as polyhedral analysis.

For debugging, it is important to provide programmers a better debugging interface, where they can insert probes to observe any part of a program or even insert breakpoints and step through the code. With the latest HeteroCL, we have already enabled simple debugging features such as printing intermediate

values and asserting the values. These values can also be useful for the trace-based analysis, which provides feedback for automatic generation of primitives.

Integration and Interoperability of Accelerator Programming Models. HeteroCL, SuSy, and DrTrace are not standalone tools. There are already projects built on HeteroCL such as HeteroHalide [135] and HeteroFlow [205]. Another obvious integration is between HeteroCL and SuSy. To be more specific, the HeteroCL compiler first identifies sub-programs that are systolic algorithms. Then, we rely on SuSy to generate high-performance systolic architectures. Both the identification of sub-programs and the suggestion of primitives can be achieved by DrTrace.

To further improve the interoperability, one direction is to integrate with other open-source frameworks. One example is MLIR [130], which is an intermediate representation for building reusable and extensible compiler infrastructure. MLIR enables a multi-level compilation stack with the concept of dialects, where each dialect may target for a particular application domain or specialize in a set of compilation tasks. To integrate HeteroCL with MLIR, we can implement our own HeteroCL dialect, where we keep all HeteroCL features such as decoupled customization, custom data types, and mixed-paradigm programming. Similarly, we can create a dialect for SuSy and we can partially lower a program in HeteroCL dialect to SuSy dialect. This is also where we can make use of existing dialects such as LinAlg for linear algebra optimizations, Affine for polyhedral optimizations, and Loop for loop optimizations. There are also some existing efforts on extending HLS with MLIR such as ScaleHLS [213]. By lowering to their dialects, we do not need to develop our own HLS back ends (or just implement some simple extensions).

BIBLIOGRAPHY

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Michael Adler, Kermin E Fleming, Angshuman Parashar, Michael Pellauer, and Joel Emer. Leap Scratchpads: Automatic Memory and Cache Management for Reconfigurable Logic. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[3] Jason Agron. Domain-Specific Language for HW/SW Co-Design for FPGAs. *IFIP Working Conf. on Domain-Specific Languages*, 2009.

[4] Mythri Alle, Antoine Morvan, and Steven Derrien. Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis. *Design Automation Conf. (DAC)*, 2013.

[5] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. Cλash: Structural Descriptions of Synchronous Hardware Using Haskell. *Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. *Design Automation Conf. (DAC)*, 2012.

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. *Int'l Symp. on Code Generation and Optimization (CGO)*, 2019.

[8] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A Code Optimization Framework for High Performance Systems. *arXiv preprint arXiv:1804.10694*, 2018.

[9] Samridhi Bansal, Hsuan Hsiao, Tomasz Czajkowski, and Jason H Anderson. High-Level Synthesis of Software-Customizable Floating-Point Cores. *Design, Automation, and Test in Europe (DATE)*, 2018.

[10] Cedric Bastoul. Code Generation in the Polyhedral Model is Easier Than You Think. *Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[11] Shuvra S Bhattacharyya, Gordon Brebner, Jörn W Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, and Mickaël Raulet. OpenDF: A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. *ACM SIGARCH Computer Architecture News*, 2009.

[12] Robert D Blumofe and Charles E Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of the ACM (JACM)*, 1999.

[13] David Boland and George A Constantinides. Automated Precision Analysis: A Polynomial Algebraic Approach. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2010.

[14] David Boland and George A Constantinides. A Scalable Approach for Automated Precision Analysis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2012.

[15] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2008.

[16] Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. Automatic Mapping of Nested Loops to FPGAs. *ACM SIGPLAN Conf. on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[17] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The Essence of Bluespec: A core Language for Rule-Based Hardware Design. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2020.

[18] Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood. High Level Modeling of Channel-Based Asynchronous Circuits Using Verilog. *CPA*, 2005.

[19] Cadence. Stratus high-level synthesis, 2020.

[20] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski.

LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[21] Zachariah Carmichael, Hamed F Langroudi, Char Khazanov, Jeffrey Lillie, John L Gustafson, and Dhireesha Kudithipudi. Deep Positron: A Deep Neural Network Using the Posit Number System. *Design, Automation, and Test in Europe (DATE)*, 2019.

[22] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A Cloud-Scale Acceleration Architecture. *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[23] Tao Chen, Shreesha Srinath, Christopher Batten, and G Edward Suh. An Architectural Framework for Accelerating Dynamic Parallel Algorithms on Reconfigurable Hardware. *Int'l Symp. on Microarchitecture (MICRO)*, 2018.

[24] Tao Chen and G Edward Suh. Efficient Data Supply for Hardware Accelerators with Prefetching and Access/Execute Decoupling. *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[26] Yu Ting Chen and Jason H Anderson. Automated Generation of Banked Memory Architectures in the High-Level Synthesis of Multi-Threaded Software. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2017.

[27] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. Bring your own codegen to deep learning compiler. *arXiv preprint arXiv:2105.03215*, 2021.

[28] Jianyi Cheng, Lana Josipovic, George A Constantinides, Paolo Ienne, and John Wickerson. Combining Dynamic & Static Scheduling in High-level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[29] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: Stencil with

Optimized Dataflow Architecture. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[30] Yuze Chi, Licheng Guo, Young-kyu Choi, Jie Wang, and Jason Cong. Extending High-Level Synthesis for Task-Parallel Programs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[31] Jongsok Choi, Stephen Brown, and Jason Anderson. From Software Threads to Parallel Hardware in High-Level Synthesis for FPGAs. *Int'l Conf. on Field Programmable Technology (FPT)*, 2013.

[32] Jongsok Choi, Kevin Nam, Andrew Canis, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2012.

[33] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[34] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. *Design Automation Conf. (DAC)*, 2016.

[35] Young-Kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 2019.

[36] Eric S Chung, James C Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[37] Alessandro Cilardo and Luca Gallo. Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning. *ACM Trans. on Architecture and Code Optimization (TACO)*, 2015.

[38] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2017.

[39] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the Search for Compositions of Program Transformations. *Int'l Symp. on Supercomputing (ICS)*, 2005.

[40] J. Cong and J. Wang. PolySA: Polyhedral-Based Systolic Array Auto Compilation. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[41] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. Best-Effort FPGA Programming: A Few Steps can Go a Long Way. *arXiv preprint arXiv:1807.01340*, 2018.

[42] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding Performance Differences of FPGAs and GPUs. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[43] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. Source-to-Source Optimization for HLS. *FPGAs for Software Programmers*, 2016.

[44] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, 2016.

[45] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2011.

[46] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Nonuniform Partitioning of Data Reuse Buffers. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[47] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[48] Jason Cong and Jie Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[49] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. Automated Ac-

celerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. *Design Automation Conf. (DAC)*, 2018.

[50] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. Bandwidth Optimization Through On-Chip Memory Restructuring for HLS. *Design Automation Conf. (DAC)*, 2017.

[51] Philippe Coussy, Cyrille Chavet, Pierre Bomel, Dominique Heller, Eric Senn, and Eric Martin. GAUT: A High-Level Synthesis Tool for DSP Applications. *High-Level Synthesis*, 2008.

[52] Tomasz S Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P Singh. From OpenCL to High-Performance Hardware on FPGAs. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2012.

[53] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph Processing framework on FPGA a Case Study of Breadth-First Search. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[54] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable Convolutional Networks. *Int'l Conf. on Computer Vision (ICCV)*, 2017.

[55] Steve Dai, Gai Liu, Ritchie Zhao, and Zhiru Zhang. Enabling Adaptive Loop Pipelining in High-Level Synthesis. *Asilomar Conf. on Signals, Systems, and Computers*, 2017.

[56] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, 2014.

[57] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[58] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing. *Advances in Systems Science*, 2014.

[59] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers,

Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. Pushing the Limits of Narrow Precision Inferencing at Cloud Scale with Microsoft Floating Point. *Advances in Neural Information Processing Systems*, 2020.

[60] Florent De Dinechin and Bogdan Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers*, 2011.

[61] Luiz Henrique De Figueiredo and Jorge Stolfi. Affine Arithmetic: Concepts and Applications. *Numerical Algorithms*, 2004.

[62] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[63] Johannes de Fine Licht, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *arXiv preprint arXiv:1805.08288*, 2018.

[64] Jan Decaluwe. MyHDL: A Python-Based Hardware Description Language. *Linux journal*, 2004.

[65] Li Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research. *IEEE Signal Processing Magazine*, 2012.

[66] Steven Derrien, Thibaut Marty, Simon Rokicki, and Tomofumi Yuki. Toward Speculative Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[67] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 2018.

[68] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-Directed Scheduling of Streaming Accelerators. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2020.

[69] Stephen A Edwards, Richard Townsend, Martha Barker, and Martha A

Kim. Compositional Dataflow Circuits. *ACM Transactions on Embedded Computing Systems (TECS)*, 2019.

[70] Johan Eker and J Janneck. CAL Language Report: Specification of the CAL Actor Language. *ERL Technical Memo UCB/ERL*, 2003.

[71] Zhenman Fang, Farnoosh Javadi, Jason Cong, and Glenn Reinman. Understanding Performance Gains of Accelerator-Rich Architectures. *Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 2019.

[72] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI. *Int'l Symp. on Computer Architecture (ISCA)*, 2018.

[73] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. *arXiv preprint arXiv:1911.09925*, 2019.

[74] W Morven Gentleman and HT Kung. Matrix Triangularization by Systolic Arrays. *Real-Time Signal Processing IV*, 1982.

[75] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. ReBNet: Residual Binarized Neural Network. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[76] Marcel Gort and Jason H Anderson. Range and Bitmask Analysis for Hardware Optimization in High-Level Synthesis. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2013.

[77] Ian Gray, Yu Chan, Jamie Garside, Neil Audsley, and Andy Wellings. Transparent Hardware Synthesis of Java for Predictable Large-Scale Distributed Systems. *arXiv preprint arXiv:1508.07142*, 2015.

[78] Paul Grigoraş, Xinyu Niu, Jose GF Coutinho, Wayne Luk, Jacob Bower, and Oliver Pell. Aspect driven compilation for dataflow designs. *Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 2013.

[79] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 2012.

[80] Anne-Claire Guillou, Fabien Quilleré, Patrice Quinton, S Rajopadhye, and Tanguy Risset. Hardware Design Methodology with the Alpha Language. *FDL'01*, 2001.

[81] Sikender Gul, Muhammad Faisal Siddiqui, and Naveed Ur Rehman. FPGA Based Real-Time Implementation of Online EMD With Fixed Point Architecture. *IEEE Access*, 2019.

[82] Peng Guo, Hong Ma, Ruizhi Chen, Pin Li, Shaolin Xie, and Donglin Wang. FBNA: A Fully Binarized Neural Network Accelerator. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2018.

[83] Tae Jun Ham, Juan L Aragón, and Margaret Martonosi. Decoupling Data Supply From Computation for Latency-Tolerant Communication in Heterogeneous Architectures. *ACM Trans. on Architecture and Code Optimization (TACO)*, 2017.

[84] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of Hardware Accelerators for Multi-dimensional Dataflow-intensive Applications. *International Workshop on Applied Reconfigurable Computing*, 2008.

[85] Frank Hannig, Holger Ruckdeschel, Hritam Dutta, and Jürgen Teich. PARO: Synthesis of Hardware Accelerators for Multi-Dimensional Dataflow-Intensive Applications. *Int'l Workshop on Applied Reconfigurable Computing (ARC)*, 2008.

[86] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. *ACM Trans. Graph.*, 2014.

[87] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible Multi-Rate Image Processing Hardware. *ACM Trans. on Graphics (TOG)*, 2016.

[88] Timothy Hickey, Qun Ju, and Maarten H Van Emden. Interval Arithmetic: From Principles to Implementation. *Journal of the ACM (JACM)*, 2001.

[89] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient Realization of Streaming Applications on FPGAs. *Intl'l Conf. on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2008.

[90] Hsuan Hsiao and Jason Anderson. Thread Weaving: Static Resource Scheduling for Multithreaded High-Level Synthesis. *Design Automation Conf. (DAC)*, 2019.

[91] Qijing Huang, Dequan Wang, Zhen Dong, Yizhao Gao, Yaohui Cai, Tian Li, Bichen Wu, Kurt Keutzer, and John Wawrzynek. CoDeNet: Efficient Deployment of Input-Adaptive Object Detection on Embedded FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[92] Yuanjie Huang, Paolo Ienne, Olivier Temam, Yunji Chen, and Chengyong Wu. Elastic CGRAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2013.

[93] Mohsen Imani, Samuel Bosch, Sohum Datta, Sharadhi Ramakrishna, Sahand Salamat, Jan M Rabaey, and Tajana Rosing. QuantHD: A Quantization Framework for Hyperdimensional Computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[94] Mohsen Imani, Sahand Salamat, Behnam Khaleghi, Mohammad Samragh, Farinaz Koushanfar, and Tajana Rosing. SparseHD: Algorithm-Hardware Co-Optimization for Efficient High-Dimensional Computing. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[95] National Instruments. Labview, 2020.

[96] Intel. Intel Math Kernel Library. 2007.

[97] Intel. Accelerating Genomics Research with OpenCL, and FPGAs. 2017.

[98] Intel. Intel High Level Synthesis Compiler User Guide. 2017.

[99] Intel. Intel agilex f-series fpgas & socs, 2019.

[100] Intel. vLab Academic Cluster. *URL: https://wiki.intel-research.net*, 2019.

[101] Intel. Intel high level synthesis compiler pro edition: Reference manual, 2020.

[102] Intel. Intel soc fpgas, 2020.

[103] Intel. The oneapi specification, 2020.

[104] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is FIRRTL Ground: Hardware Construction Languages, Compiler Frameworks, and Transformations. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2017.

[105] Manish Kumar Jaiswal and Ray CC Cheung. Area-Efficient Architectures for Double Precision Multiplier on FPGA, with Run-Time-Reconfigurable Dual Single Precision Support. *Microelectronics journal*, 2013.

[106] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. *Int'l Conf. on Multimedia*, 2014.

[107] Lana Josipovic, Philip Brisk, and Paolo Ienne. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Transactions in Embedded Computing Systems (TECS)*, 2017.

[108] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically Scheduled High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[109] Lana Josipovic, Andrea Guerrieri, and Paolo Ienne. Speculative Dataflow Circuits. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[110] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-Datacenter Performance Analysis of a Tensor Processing Unit. *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[111] Juniper. Juniper: Java platform for high-performance and real-time large-scale data, 2020.

[112] Nachiket Kapre and Deheng Ye. GPU-Accelerated High-Level Synthesis for Bitwidth Optimization of FPGA Datapaths. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[113] Soguy Mak karé Gueye, Gwenaël Delaval, Eric Rutten, Dominique Heller, and Jean-Philippe Diguet. A Domain-Specific Language for Autonomic Managers in FPGA Reconfigurable Architectures. *Int'l Conf. on Autonomic Computing (ICAC)*, 2018.

[114] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM (JACM)*, 1967.

[115] Keras. Keras. simple. flexible. powerful., 2020.

[116] Ronan Keryell and Lin-Ya Yu. Early Experiments Using SYCL Single-Source Modern C++ on Xilinx FPGA: Extended Abstract of Technical Presentation. *Int'l Workshop on OpenCL*, 2018.

[117] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. Accelerating Graph Analytics by Co-optimizing Storage and Access on an FPGA-HMC Platform. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[118] Adam B Kinsman and Nicola Nicolici. Finite Precision Bit-Width Allocation Using SAT-Modulo Theory. *Design, Automation, and Test in Europe (DATE)*, 2009.

[119] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 2017.

[120] Ana Klimovic and Jason H Anderson. Bitwidth-Optimized Hardware Accelerators with Software Fallback. *Int'l Conf. on Field Programmable Technology (FPT)*, 2013.

[121] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A Language and Compiler for Application Accelerators. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2018.

[122] David Koeplinger, Raghu Prabhakar, Yaqi Zhang, Christina Delimitrou, Christos Kozyrakis, and Kunle Olukotun. Automatic Generation of Efficient Accelerators for Reconfigurable Hardware. *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[123] HT Kung and Charles E Leiserson. Systolic Arrays for VLSI. *Sparse Matrix Proceedings*, 1979.

[124] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining under Joint Optimization. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[125] S. Kung. VLSI Array Processors. *IEEE ASSP Magazine*, 1985.

[126] Jakub Kurzak, Piotr Luszczek, Mark Gates, Ichitaro Yamazaki, and Jack Dongarra. Virtual Systolic Array for QR Decomposition. *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2013.

[127] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[128] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2020.

[129] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined fpga acceleration: Status and future prospects. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2021.

[130] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[131] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. Advanced Systolic Design. *Digital Signal Processing for Multimedia Systems*, 1999.

[132] Hervé Le Verge, Christophe Mauras, and Patrice Quinton. The ALPHA Language and Its Use for the Design of Systolic Arrays. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 1991.

[133] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998.

[134] D-U Lee, Altaf Abdul Gaffar, Ray CC Cheung, Oskar Mencer, Wayne Luk, and George A Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2006.

[135] Jiajie Li, Yuze Chi, and Jason Cong. Heterohalide: From image processing dsl to efficient fpga acceleration. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[136] Peng Li, Louis-Noël Pouchet, and Jason Cong. Throughput Optimization for High-Level Synthesis Using Resource Constraints. *Int'l Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.

[137] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: Binarized Neural Network on FPGA. *Neurocomputing*, 2018.

[138] Amy W Lim and Monica S Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 1997.

[139] Junyi Liu, Samuel Bayliss, and George A Constantinides. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2015.

[140] Junyi Liu, John Wickerson, Samuel Bayliss, and George A Constantinides. Polyhedral-Based Dynamic Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017.

[141] Qiang Liu, George A Constantinides, Konstantinos Masselos, and Peter YK Cheung. Automatic On-Chip Memory Minimization for Data Reuse. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2007.

[142] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. *Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[143] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers Through Microbenchmarking. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[144] Xiaoyin Ma, Walid A Najjar, and Amit K Roy-Chowdhury. Evaluation and Acceleration of High-Throughput Fixed-Point Object Detection on FPGAs. *IEEE Transactions on Circuits and Systems for Video Technology*, 2015.

[145] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. TABLA: A Unified Template-Based Framework for Accelerating Statistical Machine Learning. *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2016.

[146] MathWorks. Mathworks hdl coder, 2020.

[147] Maxeler. Maxeler high-performance dataflow computing systems, 2020.

[148] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Hipacc: A domain-specific language and compiler for image processing. *IEEE Trans. on Parallel and Distributed Systems*, 2016.

[149] Mentor. Catapult high-level synthesis, 2020.

[150] Microchip. Legup 9.1 documentation, 2020.

[151] Microsoft. A microsoft custom data type for efficient inference, 2020.

[152] Peter Milder, Franz Franchetti, James C Hoe, and Markus Püschel. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2012.

[153] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. VTA: An Open Hardware-Software Stack for Deep Learning. *arXiv preprint arXiv:1807.04188*, 2018.

[154] Antoine Morvan, Steven Derrien, and Patrice Quinton. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2013.

[155] Duncan Moss, Srivatsan Krishnan, Eriko Nurvitadhi, and et al. A Customizable Matrix Multiplication Framework for the Intel HARPv2 Platform - A Deep Learning Case Study. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[156] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2016.

[157] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable Accelerator Design with Time-Sensitive Affine Types. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2020.

[158] Rishiyur Nikhil. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. *Int'l Conf. on Formal Methods and Models for Co-Design (MEMOCODE)*, 2004.

[159] Mostafa W Numan, Braden J Phillips, Gavin S Puddy, and Katrina Falkner. Towards Automatic High-Level Code Deployment on Reconfigurable Platforms: A Survey of High-Level Synthesis Tools and Toolchains. *IEEE Access*, 2020.

[160] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C Hoe, José F Martínez, and Carlos Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[161] William George Osborne, Ray CC Cheung, José Gabriel F Coutinho, Wayne Luk, and Oskar Mencer. Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2007.

[162] Ganda Stephane Ouedraogo, Matthieu Gautier, and Olivier Sentieys. A Frame-Based Domain-Specific Language for Rapid Prototyping of FPGA-

Based Software-Defined Radios. *EURASIP Journal on Advances in Signal Processing*, 2014.

[163] M. Akif Özkan, Oliver Reiche, Frank Hannig, and Jürgen Teich. Fpga-based accelerator design from a domain-specific language. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2016.

[164] Alexandros Papakonstantinou, Karthik Gururaj, John A Stratton, Deming Chen, Jason Cong, and Wen-Mei W Hwu. FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs. *Symp. on Application Specific Processors (SASP)*, 2009.

[165] Philippos Papaphilippou, Jiuxi Meng, and Wayne Luk. High-Performance FPGA Network Switch Architecture. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[166] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv preprint arXiv:1912.01703*, 2019.

[167] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo, and Marco D Santambrogio. OXiGen: A tool for Automatic Acceleration of C Functions into Dataflow FPGA-Based Kernels. *Int'l Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, 2018.

[168] Christian Pilato and Fabrizio Ferrandi. Bambu: A Modular Framework for the High Level Synthesis of Memory-Intensive Applications. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2013.

[169] Louis-Noël Pouchet. Polybench: The Polyhedral Benchmark Suite. *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[170] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. Polyhedral-Based Data Reuse Optimization for Configurable Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2013.

[171] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Trans. on Architecture and Code Optimization (TACO)*, 2017.

[172] Patrice Quinton. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. *ACM SIGARCH Computer Architecture News*, 1984.

[173] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2013.

[174] Oliver Reiche, M Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating FPGA-Based Image Processing Accelerators with Hipacc. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2017.

[175] Hongbo Rong. Programmatic Control of a Compiler for Generating High-Performance Spatial Hardware. *arXiv preprint arXiv:1711.07606*, 2017.

[176] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[177] Sahand Salamat, Mohsen Imani, Behnam Khaleghi, and Tajana Rosing. F5-HD: Fast Flexible FPGA-Based Framework for Refreshing Hyperdimensional Computing. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[178] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 2002.

[179] Jocelyn Sérot, François Berry, and Sameer Ahmed. CAPH: A Language for Implementing Stream-Processing Applications on FPGAs. *Embedded Systems Design with FPGAs*, 2013.

[180] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From High-Level Deep Neural Models to FPGAs. *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[181] Jiang Shunning, Christopher Torng, and Christopher Batten. An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework. *Workshop on Open-Source EDA Technology (WOSET)*, 2018.

[182] Sam Skalicky, Joshua Monson, Andrew Schmidt, and Matthew French. Hot & Spicy: Improving Productivity with Python and HLS for FPGAs. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[183] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. End-to-End Optimization of Deep Learning Applications. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[184] Roman A Solovyev, Alexandr A Kalinin, Alexander G Kustov, Dmitry V Telpukhov, and Vladimir S Ruhlov. FPGA Implementation of Convolutional Neural Networks with Fixed-Point Calculations. *arXiv preprint arXiv:1808.09945*, 2018.

[185] Lukas Sommer, Lukas Weber, Martin Kumm, and Andreas Koch. Comparison of Arithmetic Number Formats for Inference in Sum-Product Networks on FPGAs. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2020.

[186] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[187] Robert Stewart, Kirsty Duncan, Greg Michaelson, Paulo Garcia, Deepayan Bhowmik, and Andrew Wallace. RIPL: A Parallel Image Processing Language for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)*, 2018.

[188] Synthesijer. Synthesijer github, 2020.

[189] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. Elasticflow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2015.

[190] James Thomas, Pat Hanrahan, and Matei Zaharia. Fleet: A Framework for Massively Parallel Streaming on FPGAs. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[191] Shervin Vakili, JM Pierre Langlois, and Guy Bois. Enhanced Precision Analysis for Accuracy-Aware Bit-Width Optimization Using Affine Arith-

metic. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2013.

[192] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A Rapid Prototyping Framework for p4. *Symp. on SDN Research*, 2017.

[193] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[194] Shibo Wang and Pankaj Kanwar. BFloat16: The Secret to High Performance on Cloud TPUs. *Google Cloud Blog*, 2019.

[195] Xiaojun Wang and Miriam Leeser. VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2010.

[196] Yu Wang, James C Hoe, and Eriko Nurvitadhi. Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[197] Yuxin Wang, Peng Li, and Jason Cong. Theory and Algorithm for Generalized Memory Partitioning in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2014.

[198] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. *Design Automation Conf. (DAC)*, 2013.

[199] Andrew Shell Waterman. Design of the RISC-V Instruction Set Architecture. *UC Berkeley*, 2016.

[200] MS Waterman. Identification of Common Molecular Subsequence. *Mol. Biol*, 1981.

[201] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[202] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang,

Han Hu, Yun Liang, and Jason Cong. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. *Design Automation Conf. (DAC)*, 2017.

[203] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 2009.

[204] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-A Free Verilog Synthesis Suite. *Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[205] Shaojie Xiang, Yi-Hsiang Lai, Yuan Zhou, Hongzheng Chen, Niansong Zhang, Debjit Pal, and Zhiru Zhang. Heteroflow: An accelerator programming model with decoupled data placement for software-defined fpgas. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2022.

[206] Xilinx. SDAccel: Enabling Hardware-Accelerated Software. 2020.

[207] Xilinx. Sdnet packet processor user guide, 2020.

[208] Xilinx. Vitis high-level synthesis user guide, 2020.

[209] Xilinx. Zynq ultrascale+ mpsoc, 2020.

[210] Xilinx Inc. Vivado Design Suite User Guide: High-Level Synthesis. 2012.

[211] Jingling Xue. Formal Synthesis of Control Signals for Systolic Arrays. *University of Edinburgh*, 1992.

[212] Li Yang, Zhezhi He, and Deliang Fan. A Fully Onchip Binarized Convolutional Neural Network FPGA Implementation with Accurate Inference. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, 2018.

[213] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. Scalehls: Scalable high-level synthesis through mlir. *arXiv preprint arXiv:2107.11673*, 2021.

[214] Cody Hao Yu, Peng Wei, Max Grossman, Peng Zhang, Vivek Sarker, and Jason Cong. S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters. *Design Automation Conf. (DAC)*, 2018.

[215] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 2010.

[216] Hanqing Zeng and Viktor Prasanna. GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2020.

[217] Chen Zhang, Guangyu Sun, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2019.

[218] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[219] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2021.

[220] Shijie Zhou, Rajgopal Kannan, Viktor K Prasanna, Guna Seetharaman, and Qing Wu. HitGraph: High-Throughput Graph Processing Framework on FPGA. *IEEE Trans. on Parallel and Distributed Systems*, 2019.

[221] Yuan Zhou, Khalid Musa Al-Hawaj, and Zhiru Zhang. A New Approach to Automatic Memory Banking Using Trace-Based Address Mining. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[222] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[223] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving Polyhedral Code Generation for High-Level Synthesis. *Intl'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013.