COMPILING APPLICATIONS
TO RECONFIGURABLE PUSH-MEMORY ACCELERATORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Qiaoyi Liu

December 2023

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Fredrik Kjoelstad

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Priyanka Raina

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

The slowing down of Moore's law and evolving of applications has underscored the increasing significance of domain specific architectures. Programmable domain-specific accelerators, such as coarse-grained reconfigurable arrays (CGRAs), have emerged as a promising middle-ground between efficiency and flexibility, but they have traditionally been difficult compiler targets since they use a different memory system. In contrast to general purpose compute platform, the memory hierarchies of reconfigurable accelerators use *push memories*: memories that send input data streams to computation kernels or to higher or lower levels in the memory hierarchy, and store the resulting output data streams. To address the compilation challenge caused by push memories, in this thesis, we have introduced a novel abstraction, namely, the **unified buffer**, designed to support the application compilation process to reconfigurable accelerators as well as facilitating physical hardware design for reconfigurable architecture.

The unified buffer abstraction enables the compiler to separate generic application scheduling optimizations from the mapping to specific memory implementations in the backend. This approach automates push memory scheduling optimization through a collection of compiler techniques, including polyhedral analysis and software pipelining, effectively shielding users from the low-level hardware details. This separation also allows our compiler to bridge the gap between resource-agnostic application description and resource-constrained hardware implementation, mapping applications to different CGRA memory designs, including some with a ready-valid interface. Furthermore, the separation also opens the opportunity for optimizing push memory elements on reconfigurable arrays. Our optimized memory implementation, the Physical Unified Buffer (PUB), uses a wide-fetch, single-port SRAM macro with built-in address generation logic to implement a buffer with two read and two write ports. It is 18% smaller and consumes 31% less energy than a physical buffer implementation using a dual-port memory that only supports two ports.

Finally, our system evaluation shows that enabling a compiler to support CGRAs leads to performance and energy benefits. Over a wide range of image processing and machine learning applications, our CGRA achieves 4.7× better runtime and 3.5× better energy-efficiency compared to an FPGA.

Acknowledgments

I want to extend my gratitude to the individuals who have been instrumental in shaping my academic journey and personal growth during the course of my PhD adventure.

First of all, I would like to express my thankfulness to my advisor, Mark Horowitz. I would say that Mark is not one of the most hands-on advisor I have seen. However, when I ask for help during the challenging times, Mark is always there and provides me with immense support. I enjoy those heated discussions during our meetings' whiteboard sessions and am truly appreciated that he can go over every details with great patience. He thoughtfully tailor his advising style to match my personal characteristics, emphasizing tactfulness and encouragement while keep challenging me, has built up my confidence and profoundly influenced my thinking process. I learnt that he always challenging me not because he thinks that I am incorrect but because he wants me to figure out the intrinsic characterization of the problem. His influence on my behavior, prompting deeper reflection and consideration of underlying reasons, has been transformative.

I have the privilege to collaborate with several Stanford faculties through the comprehensive scope of the Agile Hardware (AHA) project. Working under Priyanka Raina's advising and with her team has been amazing. The weekly discussion when leading up for our chip tapeout has been an enlightening experience. Her rigorous approach to research, meticulous attention to detail, and guidance through complex questions have been pivotal. Next, I would also like to thank the mentorship from Fredrik Kjolstad. I often wish that I had met Fred earlier in my PhD journey. His insights from the compiler perspective and guidance through various challenges have been invaluable. Beyond academic, hiking with Fred has provided not only enjoyable moments but also opportunities for rejuvenation. Lastly, I want to express my gratitude to Christos Kozyrakis. His support and inspiration left an indelible mark. Despite taking different paths, Christos's academic insights and positive attitude continue to inspire me.

I extend my sincere appreciation for the invaluable support and mentorship I received beyond Stanford, particularly from Nvidia Architecture Research. I am deeply grateful to Angshuman Parashar and Joel Emer for their guidance and supervision during my summer internship. Their openness and encouragement to collaborate on diverse research projects have been instrumental in broadening my perspective. Engaging in their weekly meetings significantly expanded my understanding of industrial architecture and compiler research, enriching my experience and knowledge in this field.

My greatest fortune was encountering a number of great colleagues. During my early PhD journey, Xuan Yang and Mingyu Gao gave me a lot of academic guidance and paved the road for my own research project. Later on, Dillon Huff became my polyhedral analysis guru, teaching me not only the secrets of compilers but also the geekiest coding styles. Jeff Setter provided incredible support on coming up the idea of unified buffer, as well as setting up and conducting experiment when I was heads-down on implementation. It's my great honor to have collaborators that easy to approach and always stay positive when our paper was rejected multiple times. I also want to express my gratitude for the memory backend team member, Max, Kavya, Taeyoung and Jake, our CGRA chief architect Alex, the software magician Keyi, and the rest of AHA project member, Kalhan, Jack, Yuchen, Kathleen and Po-han. Without your all's support and help, I cannot accomplish such research and much of the experiment presented here would not have existed. It's truly a great teamwork.

Despite the geographical distance and the time zone differences, my parents are always there for me providing guidance, though non-technical, and moral support. Thanks mom and dad for supporting me and believing me in every step I choose. Your encouragement and boundless love have been the source of power that helped me navigate and conquer every challenge on this roller-coaster ride called my PhD. It's unfortunate that the pandemic has disrupted our physical closeness, and I eagerly await the day when I can show both of you the place I call home in the last five years.

Lastly, I want to express my immense gratitude to my girlfriend, Iris Zhai. Our paths crossed during the pandemic, on the ski slop. And she became an unexpected source of strength. We share tears and laugh. Her continuous encouragement has been the greatest gift, and I am truly grateful for her understanding and unconditional support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, high-resolution embedded imaging sensors have become ubiquitous components in numerous commercial hardware products. They have revolutionized our ability to capture images and videos with unprecedented clarity and detail, opening the door to applications such as autonomous vehicles and robotics. These autonomous systems rely on real-time environment perception and rapid decision-making, placing a need for low-latency image processing. Simultaneously, the realm of deep learning and artificial intelligence (AI) has undergone a remarkable transformation. Groundbreaking research in this field has yielded algorithms that deliver exceptional performance improvements [32, 19, 81]. These algorithms have proven to be powerful in tasks like image recognition and language understanding, enabling machines to perform with a level of sophistication previously unattainable.

However, while these cutting-edge algorithms excel at processing high-resolution images, they demand higher power consumption and impose a substantial memory footprint on embedded systems. This conundrum presents a significant challenge as we strive to bring up the full potential of image processing and machine learning in embedded devices. This thesis addresses some of these challenges and helps empower embedded systems to deliver efficient, high-performance image processing and machine learning capabilities.

To fully understand what the challenges are, let's start the discussion with the limitations of general purpose platforms, such as CPUs. As the clock frequency plateaued following the end of Dennard Scaling, the microprocessor industry pivoted towards multi-core architectures. The motivation was clear: to leverage the increasing transistor count, provided by Moore's Law. While multi-core processors offered improved performance through parallelism, adding processors also increased power dissipation. Power constraints and the complexity of exploiting parallelism effectively [23] led to diminishing returns from increasing processor count. To increase performance at constant power requires minimizing the energy overheads of moving data to the computation. [75].

To illustrate this point, consider a scenario where a processor must access data from a distant cache level or external memory. The energy expenditure for data movement, in terms of fetching data from a remote location, can overshadow the energy required for the actual computation. For example, in 45nm technology, fetching 32-bit data from the L1 cache consumes ten times more energy than performing a floating-point addition operation. This discrepancy escalates to a factor of 1000 if the data fetch occurs from off-chip DRAM [36]. This growing disparity between computation and data movement cost raises a fundamental question: Can we find a more efficient way to orchestrate computation, particularly in scenarios where data movement is the dominant factor influencing energy efficiency and overall performance?

Fortunately, both image processing and machine learning workloads share the characteristics of high locality, statically analyzable access patterns as well as massive parallelism, which make hardware accelerator an ideal computation platform. Hardware accelerators have dedicated data path and control logic scheduled at circuit level, which is customized to specific applications. Thus, the specialization removes the instruction overhead in general purpose computation. Besides data parallelism inherent in the form of computation, the application's pipeline stages can be spatially mapped to hardware modules, yielding pipeline parallelism. In terms of data movement, the explicit software managed memory hierarchy is decoupled from the computation instructions, which can overlap the data fetch with computation, resulting in shorter latency. Moreover, the memory controller can be configured to reuse the data in local on-chip memory which further reduces energy overhead for moving data around.

Hardware accelerators implemented as fixed-function ASICs achieved best energy efficiency and performance. However, this efficiency comes at the cost of loss of flexibility. With limited reconfigurability, the fixed-function ASICs easily become obsolete as image processing and machine learning applications evolve. Accordingly, there is significant interest in building hardware accelerators with programmability. FPGAs' flexible reconfigurable routing network and fine-grained on-chip resources are capable of quickly adapting to different application algorithms. However, the overhead of routing network and building logic with Look-up Tables (LUTs) results in lower resource density and low clockwork frequency, making FPGA system less performant and energy-efficient than ASIC. [15, 66] Coarse granularity reconfigurable arrays (CGRA) address the energy efficiency issue by raising the abstraction level of the hardware building block. Instead of building memory control logic and compute engine out of LUTs and FFs, dedicated access pattern generators and processing elements(PEs) with domain specification are distributed as building blocks on the chip. With a reconfigurable routing network connecting all the resources, CGRA architecture is able to form a customized computation pipeline that achieves higher resource density and better energy efficiency while maintaining programmability.

However, along with the flexibilty and programmability, reconfigurable hardware accelerators also bring us a unique challenge: to compile applications to the accelerator and achieve good performance and resource utilization. The key compiler challenge is that efficient domain-specific accelerators use

Table 1.1: The push memories in many programmable accelerators account for a large percentage of chip area and power.

Domain	Accelerator	Area	Power
Multiple	Plasticine CGRA [69]	30%	Not specified
DNN	TPU [42]	37%	Not specified
DNN	Eyeriss [12]	67%	36–44%
DNN	Simba PEs [76]	41%	56%
Sparse DNN	EIE [30]	93%	59%

a different memory abstraction than CPUs and GPUs. General purpose hardware architectures, like CPUs, issue load/store instructions to *pull* the needed data from hardware managed on-chip memory system (known as cache) for computation, as is shown in Figure 2.1a. However, a reconfigurable accelerator like CGRA uses software-managed on-chip memories distributed across the chip to 'push' data to the computing units and store the data stream they produce. The compiled program comprises not just a single piece of code but a set of configurations running on different memory controllers distributed across various memory hierarchy levels. Since push memories control both temporary storage and the flow of data, they account for a large fraction of the chip area and power in accelerators, as shown in Table 1.1. Thus creating efficient execution schedules for these memories, which prefetch the data and maximize reuse, is crucial to achieve higher performance and better energy efficiency. Last but not least, unlike a cache used by CPUs, programmable push-memory accelerators do not have a widely adopted hardware implementation for their memory system. Compilation for push memories requires the compiler to retain the capability to target different memory hardware implementations. As a result, this compiler must support backend portability to enable exploring the design space of memory hardware to achieve the best energy efficiency.

To create a compiler that can efficiently target accelerators with push memories requires:

- A unified abstraction that covers a variety of applications' behavior while maintains the support for different backend memory architectures.
- A compiler which maps applications to spatially distributed memory resources on the reconfigurable hardware accelerator.
- A collection of optimizations to automatically schedule the applications and program the flexible resource to reduce energy consumption and latency.

Neither conventional software compilers nor existing hardware compilers meet our requirements for targeting these types of push memory. Conventional compilers for imperative programming languages that assume a Von-Neumann machine, such as LLVM, are built around an intermediate representation that separates control, data access, and arithmetic. They assume that the most important piece of architectural state that the compiler must manage is the register file. However, when using efficient push memories, memory, address generation, and control are grouped into a hardware unit, and the compiler must configure each of these units. High level synthesis (HLS) tools such as Vivado [87], LegUp [7], Catapult [56] and others [52, 40], are designed to solve scheduling and resource binding problems at a finer granularity than those seen when compiling to push memories. Their strategy works well when targeting FPGAs or ASIC technology libraries, because the architectural primitives (such as registers and LUTs) are more fine-grained than the compiler IR instructions. To efficiently execute DNN work loads, Interstellar[88], Timeloop[65] , Marvel[9] build an architecture-aware schedule optimizer to search the optimal DNN execution schedule. While this type of tool is essential for overall loop optimization, they cannot map to hardware, and do not support fusing applications with multiple stages. Accelerator generators, for instance, DNNBuilder[91], DNNWeaver[77], VTA[59] take a different approach and create an architecture template[1] for an accelerator and build a library based method to efficiently map certain application kernels down to this template. Compiler for reconfigurable architectures, including SARA[92], StreamDataFlow[63], and Exo [39] offer more abstract programming models for this specialized distributed push memory. However, their frameworks heavily rely on the backend implementation and require expert to create high efficient schedule. Extending these frameworks to support new applications or different hardware implementations would require significant development effort from domain experts.

We address these challenges by creating a new push memory abstraction that we call a *unified buffer*[51], so named because it generalizes push memories for different application domains (such as image processing and machine learning) and different reconfigurable targets (our custom CGRA[8] as shown in Figure 7.2, accelerator memory with ready-valid[67], and FPGAs[38]). In particular, we propose that the representation of push memories in the compiler must combine storage, address generation, and sequencing control logic. Unified buffers serve as the interface inside the compiler between the application and the architecture. They define both the abstraction used by the compiler during push memory mapping and the logical behavior that the hardware architects must implement. The unified buffer abstraction allows us to compile a program to a single understandable intermediate representation (IR), perform application-specific yet hardware agnostic optimization at that level, and then tailor mapping to different hardware targets.

1.1 Thesis Outline

Chapter 2 reviews the **background** of the thesis by describing the characteristics and challenges of the three major domains that the compiler must interact with:

[1] These templates can vary from a fine-grained layer-based pipeline architecture with row buffers to a multi-PE hierarchical architecture with local and global buffers.

- A push memory, the unique type of memory that the compiler must configure
- A coarse grained reconfigurable array(CGRA), the hardware target of the compiler
- Scheduling space and optimization methods, including Polyhedral analysis, software pipelining, and High-Level Synthesis (HLS) style scheduling, the tools the compiler will use to accomplish its task.

The chapter also explores relevant academic research in this field, assessing their contributions and identifying their limitations.

Based on the compilation techniques and requirements elucidated in background chapter, Chapter 3 first provides an **overview** of the application's scheduling space when deployed onto a push memory reconfigurable architecture, outlining the objectives and major tasks of an effective schedule. It then introduces the concept of the unified buffer abstraction. This abstraction serves as a foundational element in our compiler framework, which is designed to support tensor computations for machine learning and stencils pipeline for image processing. The overall compilation process is outlined in this chapter. The following three chapters then provide more information about the detailed scheduling, mapping, and optimizations through the lowering passes.

Chapter 4, **Buffer Extraction**, introduces the extraction of access patterns and iteration domains from application descriptions in the form of loop nests. It places particular emphasis on leveraging polyhedral and software pipelining techniques to effectively schedule workloads and establish a hardware pipeline.

Chapter 5, **Buffer Mapping**, explains how the compiler analyzes the information extracted in the previous stage and maps it onto the physical hardware. Within this lowering pass, various transformations are applied to reduce resource utilization, enhancing data reuse and optimizing for the constraints of physical memory.

Chapter 6, **Backend Portablity**, addresses the backend of the compiler. It first introduces the optimized physical unified buffer (PUB) implementation and various hardware backends. This chapter demonstrates the generality of our unified buffer abstraction, showcasing its capacity to generate code for various push memory implementations, thus achieving backend portability.

Finally, Chapter 7, **Evaluation**, analyzes the benefit brought by the scheduling algorithm introduced in Chapter 4 and optimization passes introduced in Chapter 5, and evaluates the full system energy efficiency and performance on our CGRA against FPGA and CPU.

Chapter 2

Background

The fundamental goal of accelerating applications using specialized hardware is to exploit the parallelism and locality within the applications. These hardware accelerators are often equipped with a massive amount of functional units, facilitating the extraction of data parallelism within each compute kernel and task parallelism across different compute kernels. However, this hardware parallelism comes with a cost. To efficiently leverage it demands precise coordination of data flow to ensure that the large number of functional units remain fully engaged and stall-free. This careful orchestration of data means these machines use a different type of memory system than normal processors, which stream the data to the functional units. These push memories are described in more detail in Section 2.1. In response to the evolving algorithms and the need to support a wide range of applications, enabling programmablity for hardware accelerators has become a prominent and actively researched field. To efficiently implement push memories in hardware, accelerators generally bundle memory and control logic together, creating the address and sequence information right where it is needed. Similarly, reconfigurable hardware platforms like Field-Programmable Gate Arrays(FPGAs) continue to incorporate specialized functional units such as DSP blocks or AI engines to improve efficiency. Following this trend, Coarse-Grained Reconfigurable Arrays (CGRAs) emerge as a compelling alternative to traditional FPGAs. To better understand the hardware our compiler will need to target, Section 2.2 provide a thorough review of programmable spatial accelerators.

Although building a specialized memory storage/controller block improves hardware efficiency, it brings significant challenges as the hardware target of an application compiler because it increases the granularity of the hardware target. Traditional application mapping approaches, such as High-Level Synthesis, stop being effective because they target resources with much finer granularity, making them incompatible with the coarser-grained nature of CGRAs. The compilation challenges encountered when mapping applications for CGRAs with push memory are discussed in detail in Section 2.3.

The last section of this chapter, Section 2.4, explores the scheduling space the compiler needs to consider when mapping applications to spatial accelerators. The spectrum of factors influencing this space is extensive, including considerations like loop transformations, dataflow optimization, memory hierarchy management, and the binding with specific hardware resources. Given the complexity of schedule optimization, we divided the scheduling process into two distinct steps, a global and a local process. In this section, we first introduce the tools developed by other research groups, which aid in finding efficient global schedules. This result can be leveraged by our flexible compiler interface, Halide scheduling primitive, introduced in Section 3.2. Next, our proposed method will transition from the global schedule to the local loop nest. To facilitate the understanding of the proposed compilation flow, two compiler techniques, polyhedral analysis and software pipelining, will be introduced, which play a pivotal role in creating optimized schedules and mapping applications onto CGRAs with push memory.

2.1 Push Memories

As we described in the introduction, massive hardware parallelism comes with a cost. To fully utilize its compute power, accelerators need a memory system to keep all function units busy. However, the latency of getting data from a processor's memory system to an accelerator is inherently non-deterministic: the accelerator's requests are queued with other processor requests to access the main memory (DRAM) or processor cache. To shield the compute functional units from this uncertainty, data must be meticulously managed. Data will be prefetched from off-chip DRAM to on-chip SRAM, pushed through the computation units, and the stream of computational outputs are subsequently collected into a memory before being flushed back to the main memory. Concurrently, given the accelerator's focus on extracting parallelism, staging buffers are inserted between compute kernels to accommodate intermediate data and provide enough bandwidth when feeding data to the functional units. Usually a specialized controller, associated with the memory, takes on the responsibility of orchestrating the dataflow between memory hierarchy levels or delivering it to the compute units, as shown in Figure 2.1b. This unique memory architecture, where data is actively pushed to its destinations, is what we call **push memory**.

General purpose hardware architectures, like CPUs, issue load/store instructions to their memory system to *pull* the needed data for computation, as is shown in Figure 2.1a. These memory instructions usually coupled with compute and they orchestrate data implicitly [67]. Normally these instructions contain the global address of the data, and rely on hardware-managed caches to interpret the address and wait for requested data to be fetched from any of their memory hierarchy levels. When using efficient push memory architectures, memory management, address generation, and control are all grouped into a separate control unit. This decoupled control unit has the ability to manage memory access independently from the compute unit. One key feature that makes push memory stand out from its counter-part **pull memory** is that the dedicated controller can leverage domain knowledge of the applications to prefetch the data to hide latency or reuse data to extract data locality. For instance, a double buffer utilizes the dedicated controller and assigns extra space in the memory to overlap the memory loads with computation which hides the data movement latency and reduces the hardware overhead introduced by cache-like pull memory. Moreover, in image processing applications, data is loaded in raster-scan order and reused within neighbor convolution windows. Therefore the locality can be captured in small memory component such as row buffer and shift register. Such an optimized architecture is called a line buffer.

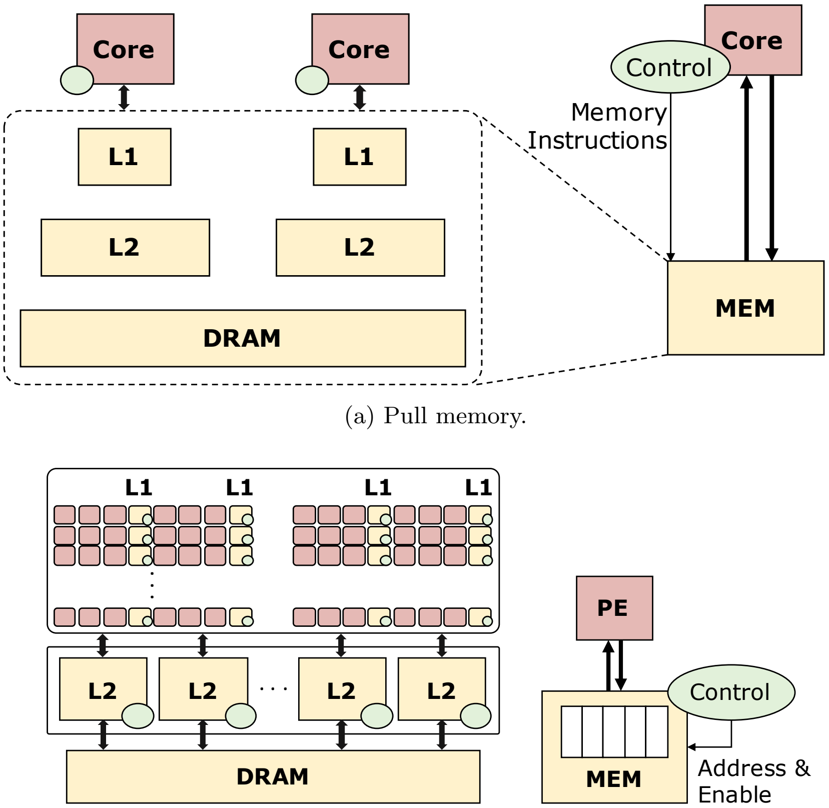
Furthermore, it's essential to note that the SRAM macros in this scenario offer limited bandwidth. Increasing the number of ports on these SRAM modules results in a quadratic growth in terms of complexity and area usage. In scenarios where highly parallel compute units are involved, memory access can become a significant bottleneck, particularly when there's a need to access multiple data elements from the same array simultaneously. Compilers need to take the responsibility to reduce the bandwidth requirement or partition data into different banks of the memory to provide enough bandwidth.

What is worse, unlike a cache used by CPUs, programmable push-memory accelerators do not have a widely adopted hardware implementation for its memory system. These accelerators typically use a custom implementation of push memory hardware optimized for specific applications, or classes of applications, to minimize area and energy. Thus, compilation and optimization for push memories require the compiler to target a different memory abstraction for every application. More details about the push memory implementations on the target reconfigurable accelerators will be introduced in the Section 2.2. To avoid rebuilding a mapping process for each unique memory implementation, a compiler abstraction that can be ported to multiple backend will be beneficial.

To generalize push memory hardware that can be reused across multiple domains, Buffets [67], a buffer implementation idiom with explicit decoupled data orchestration (EDDO), was proposed by NVIDIA research. It can be utilized as a substitution for other on-chip memory, such as cache, double buffered scratchpad or FIFO. It has a dedicated state machine for memory management. Combined with address generation and decoupled functional unit, it is more energy efficient with lower latency. We leverage many of the concepts of Buffet, a hardware primitive, when we develop our compiler abstraction in the next chapter.

2.2 Coarse-Grained Reconfigurable Architectures

In the quest to optimize hardware acceleration, the choice of underlying architecture also plays a pivotal role. Coarse-Grained Reconfigurable Architectures (CGRAs) represent a distinct class of computing platforms that sit between general-purpose processors and Application-Specific Integrated Circuits (ASICs). These architectures are designed to provide a balance between the flexibility of general-purpose processors and the efficiency of ASICs for specific application domains.



(a) Pull memory.

(b) Push memory.

Figure 2.1: Pull style memory in a CPU with centralized memory hierarchy versus push style memory in a CGRA with distributed memory and its control.

The concept of CGRAs originated in the early 1990s[31, 10] as a response to the growing demand for specialized computational accelerators. These accelerators aimed to provide close to ASIC performance and energy efficiency with the programmability typically provided by software.[36, 62]. Over the years, CGRAs have seen numerous research and development efforts to refine their architectures and programming models[63, 69]. One of the key advantages of CGRAs is their suitability for accelerating applications that exhibit regular and repetitive computational patterns[50]. Examples of such applications include digital signal processing, image and video processing, machine learning and artificial intelligence workload, and many scientific computing tasks. By allowing for specialized hardware configurations, CGRAs excel in achieving high throughput and energy efficiency for these types of workloads.

Coarse-Grained Reconfigurable Arrays (CGRAs) can be broadly categorized into two types based on their architecture characteristics. The first type of CGRA system features processor like processing elements(PEs) [1]. In this design, the memory components are seamlessly integrated with the computational units, creating a unified system where memory and computation work closely together. This approach simplifies data movement and data access. However this tighten integration may limit the flexibility and incurs processor instruction overhead. On the other hand, the second type of CGRA system adopts a different approach. In this setup, there is a clear separation between the processing elements (PEs) and memory tiles. The memory tiles are responsible for pushing data to an array of processing elements. This decoupled architecture allows for greater flexibility in terms of memory access and data flow.

In this thesis, we adopt a recent CGRA architecture (Amber CGRA)[8], proposed by researchers at Stanford, shown in Figure 7.2 as a representative example of the second type of CGRA which decouple memory from compute. While some CGRA designs have proposed using PEs for calculating memory addresses [54, 24, 79], most systems, like our target CGRA Amber[8], create dedicated addressing units associated with their push memories to reduce energy and area overhead[68]. Unlike general-purpose hardware that has a centralized memory system, the memory systems on a CGRA, are distributed. For instance, Figure 2.1b demonstrates the memory hierarchy of a coarse-grained reconfigurable array (CGRA) with multi-bank global buffer (L2) and on-chip memory tile (L1). Typically a CGRA will contains an array of memory and compute units, connected by a flexible reconfigurable network on chip (NOC). With the goal of achieving optimal area efficiency and minimizing energy consumption per unit fetch, our CGRA push memory architecture employs a wide-fetch single-port SRAM units to emulate multi-ported memory (more details will be introduced in Section 6.1.3). However, this hardware optimization introduces added complexity for the compiler in terms of mapping access patterns and creating efficient configurations. This architecture is accompanied by a comprehensive toolchain flow (AHA Flow)[4] that manages placement, routing, and low-level hardware configuration, streamlining the application graph's mapping onto the reconfigurable architecture. Our exploration includes the compilation process and software-hardware co-design, with a particular focus on accelerating dense linear algebra workloads and optimizing energy-efficient computing solutions.

2.3 Compilation Delimma

CGRAs are built on the premise that reconfigurability can be strategically applied at a coarser granularity compared to Field-Programmable Gate Arrays (FPGAs). Instead of reconfiguring individual gates and flip-flops, CGRAs are tailored to reconfigure coarser gain blocks: functional units, data paths, or even entire processing elements. The rise in granularity of hardware logic units poses a dilemma. High level synthesis (HLS) tools such as Vivado [86], LegUp [7], Catapult [55] and others [52, 40], are designed to solve scheduling and resource binding problems at a finer granularity than those seen on a CGRA. They schedule the instructions in the standard software IR, following by binding instructions to functional units, and finally emit code to synthesize the hardware. Their strategy works well when targeting FPGAs or ASIC technology libraries, because the architectural primitives (such as registers and LUTs) are more fine-grained than the compiler IR instructions. Programmable push memory accelerators requires the compilation process to generate instructions or configuration for the specialized controller rather than synthesizing the hardware using reconfigurable logic.

Apart from the challenges brought by raising the abstraction level, another compiler challenge comes from the push memory backend. Conventional compilers for imperative programming languages, such as LLVM[47], are built around an intermediate representation that separates control, data access, and arithmetic and thus are not well-suited to target push memory which integrate these components.

Luckily, researchers in this area have proposed various solutions to tackle these compilation challenges. We describe results in two key areas: one focuses on generating accelerators on reconfigurable fabric, the other focuses on compilation to push memories.

2.3.1 Domain-Specific Accelerator Generators

Previous researchers sought to automate the domain-specific accelerator generation process and create an implementation on a reconfigurable architecture or as an ASIC. Image processing accelerator generation languages such as Darkroom [33], Rigel [34], Aetherling [20], Hetero-Halide [48], HIPACC-FPGA [73], PolyMage-FPGA [14], SODA [13] and Halide-HLS [70] automatically generate FPGA implementations of image processing pipelines. These systems either target FPGAs that have large overheads, or ASICs that are inflexible. AutoSA [83], AutoDSE[78], and Clockwork[38] are some systems that use polyhedral analysis for scheduling, but they do not consider CGRAs.

To efficiently execute DNNs, Zhang et al. [90] optimize DNN data blocking using double buffer structures and synthesize a pipelined FPGA accelerator from Caffe [41]. DNNWeaver [77] also generates synthesizable designs automatically from Caffe, with support for more types of layer implementations. DNNBuilder [91] proposes a fine-grained layer-based pipeline architecture with a line-buffer-based scheme to reduce FPGA on-chip memory usage. VTA [58, 59] provides a full hardware-software stack for DNN acceleration using a modified version of Halide IR. It proposes an ISA to map DNN layers onto optimized operators on their proposed FPGA accelerator. These domain-specific hardware generators reduce design effort when mapping a DNN to an accelerator. However, their frameworks heavily rely on the backend implementation. With the architectures determined, extending them to support new applications or more efficient hardware implementations would require significant development effort from domain experts.

2.3.2 Compilation to Push Memory

Reconfigurable accelerators like CGRA often have parallel computation units. In order to fully utilize the resources, staging buffers are inserted between functional units to accommodate intermediate data and provide enough bandwidth when feeding data to the compute units. This yields a unique programming challenge: the compilation target is not just a single piece of code, but a set of programs (or configuration bit-streams) running on every memory's controller (green ovals in Figure 2.1b) that manage the data movement. Not only must the programs contain information on which data should be accessed (the addresses), but they must also align the timing of read and write events inside the buffers to synchronize the flow of data from buffers, through processing elements for computation, to another buffer.

While HLS tools can translate the compiler IR directly to hardware, they do not support the memory optimizations as described above. Modern HLS tools such as Vivado HLS or Catapult HLS are well suited to arithmetic mapping and exploiting pipeline parallelism within the bodies of individual loops [53]. However, they perform limited memory [70] and cross-loop optimizations [94]. As a result, they are not good at exploiting pipeline parallelism across different loop nests in a computation, and require a great deal of manual effort by users to create high quality code for deep pipelines. HeteroCL [45] uses a unified DSL frontend to describe their memory optimization and spatial architecture description, But its backend implementation still depends on separate frameworks for different categories of applications. More importantly, all of these HLS based framework can only synthesize accelerator design on an FPGA, which is non-trivial to extend to target accelerators with coarser level of hardware granularity.

Academic compiler systems tailored for reconfigurable accelerators, including Spatial [43] and Exo [39] offer more abstract programming models for this specialized distributed push memory. However, they often require users to explicitly define memory micro-architecture optimizations. Spatial [43] provides a high-level programming language for Plasticine, but requires users to explicitly orchestrate data movement between different memories. SARA [92] improves upon the Plasticine compiler by scaling applications to utilize most of the hardware resources, but leaves resource binding optimizations to the user. Exo empowers users with a general programming language for efficient kernel generation on both CPU and hardware accelerators, but the memory management optimizations and micro-architecture configurations remain dependent on the user-defined libraries. Nowatzki [63] proposed a low-level programming model for stream dataflow CGRAs, which features a global scratchpad memory architecture and dynamic scheduling in their memory ISA. While this approach is suitable for the CGRA architecture with a centralized global buffer, it may not align well with other accelerators equipped with distributed push memories.

In summary, these academic compilation systems for push memory on CGRAs provide abstract programming models but often necessitate user involvement in memory architecture optimizations and micro-architecture configurations. We extend this work to remove the needed user involvement by developing a novel compiler abstraction tailored to efficiently and automatically compile for such architectures.

2.4 Scheduling Techniques

Because of the explicit memory management and the large sum of resource on the push memory accelerators, achieving high performance and energy efficiency in mapping application on accelerator architecture involves considering multiple factors, resulting in an extensive design space. This diverse design space encompasses a wide range of scheduling and architectural features, spanning from schedule optimizations (the inner loop of the process) like loop tiling, loop ordering and parallelization, to more architectural considerations (the outer loop), which includes memory size, Processing Element (PE) array size, and interconnect network topology.

In this section, we first introduce the tools that have been developed by other research groups, as they provide valuable insights into finding efficient global schedules. It's important to note that coming up with a global schedule is a formidable challenge, and there is no one-size-fits-all approach that can be universally applied. Instead, we rely on Halide's scheduling primitives, as introduced in Section 3.2, as a flexible interface, which we use to express these schedules. This allows us to simplify the complexity of scheduling, enabling the compiler to automatically develop efficient local schedules tailored to specific application workloads on the push memory backend. This local schedule generation leverages two compiler techniques: polyhedral analysis and software pipelining. Traditionally, these techniques have been closely associated with conventional CPU compilation. However, in the context of this thesis, we adapt and extend these methods to address the unique challenges arising in the scheduling of operations for push memory accelerators.

2.4.1 Accelerator Scheduling Framework

Prior scheduling frameworks like Timeloop [65], Interstellar [88], and Marvel [9] have created analytical models with specific hardware constraints to formalize the large design space. They then search the scheduling space to minimize the architecture model's cost. However, this brute-force approach becomes impractical for large Deep Neural Network (DNN) layers, leading most of these works to focus on single DNN layers and neglect the potential for fusing multiple operations.

In contrast, Tangram [26] and recent research [6] have modeled complex multi-layer DNN workloads with optimized inter-layer fusion scheduling, eliminating excessive data duplication and memory access overhead for intermediate data. However, their search space remains constrained to specific data flows, employing heuristics to target particular schedule spaces due to the increased complexity introduced by fusing multiple layers.

To avoid brute-force searching, approaches like Naas[49] and Spotlight[37] leverage feedback-driven methods, such as genetic evolution algorithms and Bayesian optimization, to navigate the search space more efficiently. However, these frameworks, while capable of analyzing the mapping of tiled computations to spatial accelerators, do not provide a programming interface for users to create arbitrary affine loop nests. What is worse, they also could not offer a fully functional backend for generating code for reconfigurable accelerators with heterogeneous resource.

To implement a complete compiler, previous works like [11, 60, 2] incorporate auto-tuning, beam search, and machine learning to auto-schedule applications and generate efficient code for execution on CPUs or GPUs. We will use the insights and schedules generated by these tools to set the global schedule of our applications.

2.4.2 Polyhedral Analysis

The Polyhedral model is a powerful algebraic framework that has significantly advanced the analysis and optimization of affine programs. It achieves this by precisely capturing the compile-time execution order of operations in a program. Traditional architectures, such as CPUs and GPUs, employ a pull memory system characterized by data loads and stores that are implicitly managed by the hardware.

Since in a cache based system, the exact timing of when the data returns is not known, most polyhedral scheduling algorithms like Feautrier's algorithm[25] and PLUTO[5], transform loops, and only track the relative order of the operations. These algorithms map elements of the iteration domain to a sequence of execution during compile time. Essentially, they transform the original loops within a program into a new set of loops that implement these optimized execution orders, while the hardware underneath handling the scheduling. This program execution order can be used to reason about behavior of a hardware's compute and memory units and help enhance improve the program locality and data caching.

In contrast, reconfigurable accelerators with push memory system are mostly operated in a statically programmed and explicitly managed manner. This crucial distinction allows a compiler framework to comprehensively analyze the precise timing behavior of the operations within an affine program. This analysis occurs during compile-time and becomes invaluable in generating configurations that maximize performance and energy efficiency for all programmable units within a coarse-grained reconfigurable architecture. Thus, when scheduling and mapping for push memory accelerators, our approach diverges from conventional polyhedral scheduling algorithms. Instead of focusing on loop transformations, we concentrate on mapping these loop nests to a wide array of distributed hardware units that operate individually and possibly concurrently. To accommodate this distinction, our proposed scheduling approach maps the operations of the multidimensional iteration domain within the application program to scalar values. These scalar values represent the *number of unstalled cycles after reset when the operation begins.* Given the pipelined nature of the hardware design, multiple operations may share the same timestamp. This distinction is essential to effectively schedule for hardware accelerator with parallel compute units and maximize its performance.

2.4.3 Software Pipelining

Software pipelining is an optimization technique used in compilers for VLIW processor to improve instruction-level parallelism of loop-intensive algorithms[46]. By overlapping the execution of loop iterations, it minimize stall cycle and improve hardware utilization. This technique orchestrates a transformation of the operations nested within the innermost loop, rendering them as a series of stages in a pipeline. Such an arrangement enables each stage to process different loop iterations concurrently.

Iterative modulo scheduling, on the other hand, is a fundamental approach within software pipelining that determines the initiation interval (II) of operations within a loop[72]. The II defines the minimum number of clock cycles required between starting of an operation in the next loop iteration. By iteratively scheduling operations to minimize II while meeting resource constraints and dependencies, this technique plays a pivotal role in achieving high throughput and optimal hardware resource utilization.

Software pipelining was utilized in HLS tools [64] to generate efficient hardware implementations from high-level descriptions. Specifically, loop pipelining is an essential technique in high-level synthesis to increase the throughput and resource utilization of reconfigurable accelerators. It relies on iterative modulo schedulers to compute an operator schedule that allows subsequent loop iterations to overlap partially when executed, while honoring all dependencies and resource constraints. The HLS's scheduler employs these techniques to automatically create a schedule that adheres to specified constraints, such as clock frequency and number and types of compute resources available. However, it's worth noting that the subsequent stages in HLS, such as hardware binding and logic synthesis, cannot be directly applied in the context of an accelerator backend featuring coarse-grained push memory or functional units. Nevertheless, the frontend scheduling, facilitated by software pipelining, offers an viable approach to scheduling high-level algorithms and structuring compute pipelines.

While software pipelining traditionally operated at the level of individual operations, primarily enhancing instruction-level parallelism within VLIW processors, its application in the context of accelerators, particularly those adopting dataflow execution models with coarse grained hardware like CGRA-based implementations, takes on a different form. Here, a compute pipeline is generated where discrete functional units process a continuous stream of data, collectively executing a series of operations. Our approach extends the abstraction level of pipeline stages beyond individual operations to encompass groups of operations or entire computation kernels. This adjustment aligns with the organizational structure of accelerators, enable software pipelining to create hardware pipeline on reconfigurable accelerator architecture.

By integrating these compiler techniques into our compiler framework, we aim to bridge the gap between software and hardware, addressing the unique challenges presented by push memory accelerators while striving for efficient and high-performance outcomes. In the chapters that follow, we will give a brief introduction of our compiler flow and the novel compiler abstraction, the unified buffer.

Chapter 3

Compiler Overview

In the preceding chapter, we introduced the compilation challenge of mapping a wide range of applications to different efficient hardware implementations of push, or streaming memories. After examining conventional compiler approaches and recent research efforts, it was apparent that none offers a straightforward solution for automating the compilation process for push memory accelerators. To address this gap, this chapter first explores the design space when scheduling a workload on push memory accelerators. Next we will introduce our compilation flow and describe which part of the scheduling task each compilation stage handles. Finally we will dive into the detail of the heart of our compiler, the unified buffer abstraction, serving as the intermediate representation (IR) of our compilation framework.

3.1 Accelerator Scheduling Space

The push memory model of accelerators places the responsibility on programmers or compilers to develop software that manages data movements. This model operates in an architecture with a substantial number of on-chip memories and compute units distributed on chip, enabling parallelism and the creation of computation pipelines. However, this abundance of resources also presents scheduling challenges. Scheduling applications on such architectures requires us to specify the mapping of computation operations onto the processing elements, the placement of data into memories, as well as the communication of data between memory hierarchy levels. Drawing insights from prior architectural work[88, 42, 36], we recognize the importance of saving energy associated with data movement during accelerator scheduling. Therefore, when scheduling an application on reconfigurable accelerator, we follow the concept proposed by Hagedorn [28, 29] where both data movement and computation are regarded as first-class operation objects. Within this setup, two distinct spaces come into play: the **operation space** and the **data space**.

It's essential to note that the mapping from operation space to the data space depends on the application's access pattern, which is a characteristic independent of the schedule itself. The major task of scheduling is

1. Figuring out the order of all operations' execution.

2. Mapping points in operation space and data space to their corresponding physical hardware elements.

The first step defines the execution order of operations within each kernels, which subsequently yields the iteration space. For instance, if we adopt the loop nest representation of the application, the iteration space is the Cartesian product of all loop iterators. The sequence and dataflow execution will be shaped by loop tiling factors and the loop order. If the workload contains multiple operations, the interleaving granularity also need to be defined. An alternative strategy involves mapping the iteration space to time, which simplifies the identification of overlapping execution among different operations.

The next step is hardware mapping. Following the establishment of an iteration's temporal behavior, we proceed to associate each operation with a hardware resource, thereby introducing parallelism. Previous work [88, 70] leverage the Halide `unroll` scheduling primitive to create parallel compute unit. Alternatively, [65] annotates the `for` loop using `spatial` and `temporal` keyword, which indicates whether the loop will be executed in parallel or sequentially. Similarly, when dealing with iteration space points which belong to distinct operations, one can map the operation to the same hardware resource, facilitating shared computation. On the other hand, these operations can be mapped to separate hardware. In cases where no dependencies exist between these operations, parallel execution becomes feasible when mapped to separate units. If dependencies are present, the dedicated processing elements are employed to express pipelined parallelism.

A similar mapping strategy can be extended to the data space. For instance, a single data array or tensor may be partitioned into multiple physical tiles. Conversely, different data arrays can be consolidated and stored into the same physical memory. It is important to note that data space mapping differs from operation space mapping in that data values can reside in multiple physical location across various memory hierarchy levels.

3.2 Compiler Overview

In order to handle the application scheduling challenge mentioned in previous section, a multi-level compilation flow was developed, which is illustrated in Figure 3.1. Users of our system specify their applications in Halide, a high-level domain-specific language (DSL). Halide separates the application algorithm from its schedule to isolate computation from optimizations in execution [71]. The algorithm specifies the computation of an output, while the schedule specifies the order in which the computation should be performed. Our compiler divides the problem of compiling data array in Halide's algorithm to push-memory implementations into three steps (shown in Figure 3.1):

1. **Halide Scheduling** leverages the Halide scheduling system, whose scheduling language controls loop transformations and which we extend with accelerator commands. We support a subset of the Halide input language that includes stencils, tensor computations, and lookup tables.

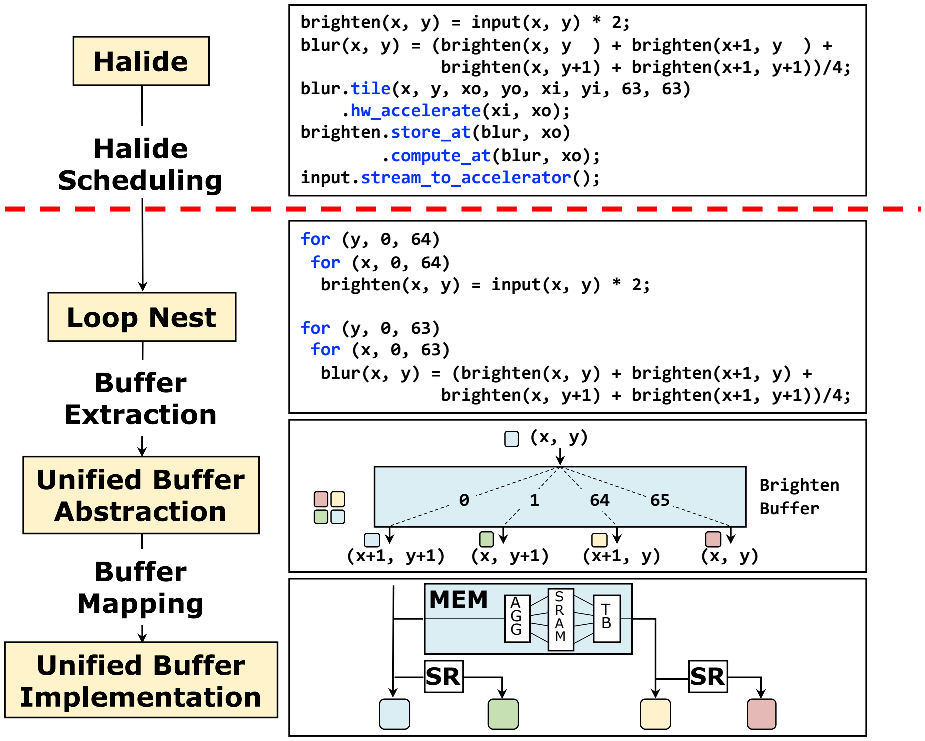
2. **Buffer extraction** uses polyhedral scheduling and software pipelining techniques to map the multidimensional iteration spaces of loop nests into one-dimensional cycle times, thus yielding pipeline parallelism. The same step then extracts the full specification of each buffer port in the unified buffer abstraction, as shown in Figure 3.2.

3. **Buffer mapping** converts each abstract unified buffers into a set of simpler buffers which can be one-to-one mapped to the low-level hardware primitives, based on the low-level hardware constraints. Through this mapping process the compiler also generates the code or configuration for specific hardware primitives. This step guarantees the compiler optimized schedule can be executed on the efficient physical memory primitives built separately.

We chose to keep the Halide scheduling language for tiling instead of placing it in the second step (like the PLUTO scheduling algorithm [5]). The reason is that a high-quality, general-purpose tiling algorithm for all dense linear algebra applications has not yet been found. As a result, we believe tiling is best left to either performance experts through a scheduling language or to domain-specific search procedures such as [88]. Thus, we limit our use of polyhedral techniques to memory analysis and semantic-preserving loop fusion. Besides tiling, the first Halide scheduling stage also responsible to define the sequencing of iteration space, including loop order, interleaving granularity as well data duplication for memory hierarchy.

The following chapters of this thesis focused on the compilation flow starting from the loop nest representations. The frontend scheduling language utilized can be any language capable of transforming the application algorithm into a loop nest representation. For instance, it is possible to extend domain-specific compilers used in machine learning, such as TVM [11] or XLA [74], to accommodate our unified buffer compilation flow. In practice, we have chosen Halide as our frontend language due to its concise scheduling language, which provides a formal definition of the scheduling space.

Consequently, the subsequent content will center on the compiler's lowering pass below the Halide Intermediate Representation (IR) depicted in Figure 3.1. The input for the compiler comprises a loop nest description language that can be targeted by the domain-specific Halide language. The compiler's output constitutes a hardware graph, comprising compute and memory modules, annotated with configurations tailored to execute the specific workload. In the next section, we will dive


```
brighten(x, y) = input(x, y) * 2;
blur(x, y) = (brighten(x, y  ) + brighten(x+1, y  ) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
blur.tile(x, y, xo, yo, xi, yi, 63, 63)
    .hw_accelerate(xi, xo);
brighten.store_at(blur, xo)
        .compute_at(blur, xo);
input.stream_to_accelerator();
```

```
for (y, 0, 64)
 for (x, 0, 64)
  brighten(x, y) = input(x, y) * 2;

for (y, 0, 63)
 for (x, 0, 63)
  blur(x, y) = (brighten(x, y) + brighten(x+1, y) +
              brighten(x, y+1) + brighten(x+1, y+1))/4;
```

Figure 3.1: The three compiler steps for a brighten-then-blur example. Halide scheduling generates tiled loops, from which buffer extraction emits the brighten unified buffer. This buffer is then mapped to shift registers (SR) and our optimized memory tile (MEM) with aggregator (AGG) and transpose buffer (TB).

into the fundamental intermediate representation used in this compilation flow: the **unified buffer abstraction**.

3.3 Unified Buffer Abstraction

In contrast to the centralized instruction issue unit employed by a CPU's execution model, which is control-flow oriented, push memory accelerators employ distributed memory controllers for data synchronization, operating in a data-flow manner. A push memory is fundamentally defined by the data stream they produce (output) or consume (input). To effectively capture and analyze this data stream behavior, we require a data-centric intermediate representation.

In our compiler framework, we introduce a novel compiler abstraction, called the *unified buffer*. This abstraction allows us to separate the part of the compiler that analyzes the application's data flow from the task of mapping the needed storage onto an efficient hardware implementation. Each port of the unified buffer is specified with respect to an iteration domain that enumerates all the elements of the data stream associated with that port. This specification also includes addressing information for write and read operations, allowing re-ordering and data reuse between input and output streams, as well as precise timing / ordering details for data arrival and departure. Essentially, for each statement that uses the port, the following information is provided:

- **The iteration domain** of the operations (statement instances) that use the port. The domain is defined by the bounds of loops in the loop nest and can be used to compute the length of the stream.
- **The access map** of the operations that maps each iteration domain point to a value read or written on the port.
- **The schedule** of the operations in the iteration domain. It doesn't need to be optimal, but it does need to preserve data dependencies in the computation. This schedule specifies when the value at each location arrives.

For this representation we use the polyhedral model [5], which provides a compact way to represent schedules and memory access patterns as integer sets and relations. Figure 3.2 shows the unified buffer that is generated to communicate between the brighten and blur stages of the example in Figure 3.1. The unified buffer interface is a list of ports. Each port is either an input port or an output port. In this case there is one input port for the values written by the brighten stage, and there are four output ports, one for each pixel in the 2x2 window read by the blur stage. This buffer accepts one pixel per cycle from the brighten compute kernel and delivers a 2×2 window of pixels per cycle (after a startup delay) to the blur kernel. To accommodate the required bandwidth, this unified buffer has 5 ports: 1 input port and 4 output ports. The iteration domain integer set, the access map, and schedule relations are implemented using the polyhedral analysis tool ISL [82]. For the input port in our example, the iteration domain is the integer set

$$\{(x, y) \mid 0 \leq x \leq 63 \wedge 0 \leq y \leq 63\}.$$

Since the brighten operation, which is the only user of the input port, is surrounded by a two-dimensional loop, the iteration domain has two index variables: an outermost index variable y and an innermost index variable x.

The unified buffer does not merely specify what operations use a port. To synthesize address generation code and optimize memory sizes, it must also specify what memory locations are accessed by those operations. To specify the memory locations, each port has an access map. For example, the second output port of the brighten buffer has the access map $(x, y) \rightarrow \text{brighten}(x + 1, y)$, which means the accessed value is to the right of each point in the operation's iteration space. The other output ports have different maps, thus collectively fetching the 2×2 stencil required by the blur kernel.

The last component in a port's specification is the polyhedral schedule which maps loop nests to

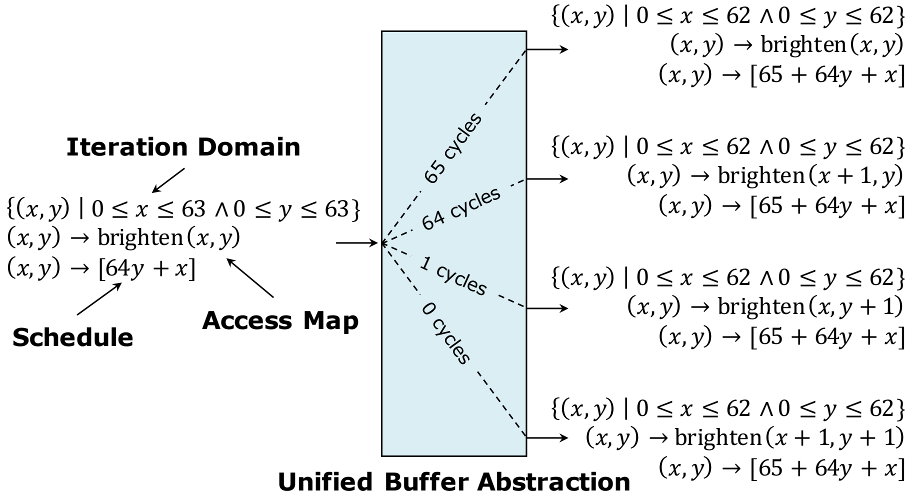

Iteration Domain
{(x, y) | 0 ≤ x ≤ 63 ∧ 0 ≤ y ≤ 63}
(x, y) → brighten(x, y)
(x, y) → [64y + x]
Schedule
Access Map
65 cycles
64 cycles
1 cycles
0 cycles
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x, y)
(x, y) → [65 + 64y + x]
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x + 1, y)
(x, y) → [65 + 64y + x]
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x, y + 1)
(x, y) → [65 + 64y + x]
{(x, y) | 0 ≤ x ≤ 62 ∧ 0 ≤ y ≤ 62}
(x, y) → brighten(x + 1, y + 1)
(x, y) → [65 + 64y + x]
Unified Buffer Abstraction

Figure 3.2: The unified buffer specifies the data movement between the brighten and blur kernels in Figure 3.1. Each port is defined by a polyhedral iteration domain and an access map that describe the data written to and read from the buffer. The schedule describes the cycle at which those values arrive at the port.

cycle times in a hardware design. This contrasts with conventional polyhedral schedule algorithms which is doing loop transformation, our schedules map loop nests to time. Specifically, it's the number of unstalled cycles after reset when each operation begins. And to accommodate pipelined hardware designs, our schedules map several operations to the same timestamps.

The schedule is used to calculate when reads and writes occur. In our example, the schedule for the input port is the integer function $(x, y) \to 64y + x$. It specifies that the first write to the brighten buffer input port, at coordinate $(0, 0)$, happens $64 * 0 + 0 = 0$ cycles after execution begins and that the second brighten operation, at coordinate $(1, 0)$, happens after $64 * 0 + 1 = 1$ cycle. Furthermore, the output ports emit their first value after $65 + 64 * 0 + 0 = 65$ cycles, which is the time the buffer must delay the first value to generate the correctly aligned output. The internal distances refer to the number of cycles from when a value arrives at an input port to when it leaves an output port.

Unified buffers separate the part of the compiler that analyzes programs to determine and optimize data movement, from the part that implements the data movement by configuring physical memories. Therefore, they have two objectives:

1. provide a precise description of the requirements of each push memory at its interface and

2. maximize opportunities for independent optimization on each side of the interface.

The first objective preserves the functionality of the application, while the second ensures the freedom for hardware designer to explore an efficient implementation. The unified buffer interface describes the observed behavior of the memory at its interfaces, in terms of the operations in the original program. The unified buffer does not specify the internal implementation of its behavior and can be used to map to different hardware backends. Only externally visible scheduling and binding decisions are expressed. Crucially, unified buffers omit the physical capacity of the memory and the physical mapping of data to locations in memory. Thus, it is a precise specification in terms of familiar data structures for a compiler—the sets and relations of the polyhedral model—and leaves the architects considerable room to optimize the design. Next, we describe how we extract this unified compiler abstraction from the frontend loop nest description.

Chapter 4

Buffer Extraction

This chapter discusses in detail the buffer extraction process, starting with the domain specific language that generates the loop nest AST. It then shows how to extract iteration domain and access map information from this information. Then we will examine the techniques and algorithms used to schedule the operations in the loop nest, including loop pipelining to optimize throughput (initiation interval of the loop) (Section 4.3) and loop fusion (Section 4.4) which optimizes latency.

4.1 Loop Nest Description Language

The frontend of the buffer extraction process is a domain specific language (DSL) which creates a representation of each application loop nest generating an output data block. Its output represents the memory load and store addresses of the data arrays used, and the order of operations set by loop tiling and other ordering transformations. This language constructs the loop nest by adding two kinds of nodes, *loop* and *op* node, in the loop nest abstract syntax tree (AST). We modified the Halide compiler, which we use for our applications, to generate this description.

The *loop* node in the AST encodes loop information such as the starting index and loop bound. The operation node encodes a number of memory load operations (getting the needed inputs), one memory store operation (storing the generated output), and a function that calculates the output from its inputs. Here are the language primitives to construct the input loop nest to the buffer extraction process.

- `c_node = p_node.add_loop(name:str, start:int, bound:int)` creates a loop node and attaches it to the parent node (`p_node`). The argument `str` specifies the name of the newly added loop iterator, and the `start` and `bound` indicates the beginning value and the loop bound of the corresponding loop.

- `op_node = p_node.add_op(name:str)` creates a op node and attaches to its parent node. The argument `str` defines the name of the operation.

- `op_node.add_load(name:str, addr:expr)`, `op_node.add_store(name:str, addr:expr)` are used to add memory load and memory store instructions to the operation. The `str` argument specify the name of the data array being accessed, which will be used as the name of the extracted unified buffer later on. The addressing argument `expr` is constructed as a high-dimensional expression that uses an affine map to calculate the memory address at which a load or store operation should occur. The iterator names in the address expression correspond to the names of the *loop* nodes in the loop nest. Thus, the address expression argument is a pointer to the *loop* nodes that are on top of the current operation.

- `op_node.add_function(name:str)` declares the computation function that the operation applies on the input data to get the output. It worth noting that the sequence of the function's input argument corresponds to the declaration order of the memory load operations. This alignment establishes a clear connection between the specific memory port and the input port of the functional unit. The term `str` will also served as a reference to the compute hardware unit. Mapping different operation nodes to the same hardware indicates compute hardware sharing, which will be discussed in Section 4.3.3.

The pseudo code in Figure 4.1a outlines an image pipeline that brightens and blurs an image by breaking it into 16 tiles and then operating on each tile. The first kernel performs a multiplication of every pixel by two within a 64x64 tile, followed by a 2x2 average blur. And the code in Figure 4.1b demonstrates the utilization of the AST constructing domain specific language to generate the two-stage processing pipeline depicted in Figure 4.4, which served as the input to our compiler. To be specific, on line 3 of the loop nest shown in Figure 4.1b, a tiling loop with a loop bound of 16 is created, followed by two children nodes representing the two loop nodes *y_br* and *y_bl*. Underneath these sub-loop nests, the operations for brighten and blur are executed. As we can see from lines 11 to 13, the brighten operation is associated with one memory load to the *input* and one memory store to the buffer *brighten*, whereas the blur operation has four memory load operations so that it can compute the blurred pixel and store back into the *blur* memory. The data load from those four memory operations will be passed to the blur kernel's input arguments according to their adding sequence.

The buffer extraction step analyzes the loop nest generated from this domain specific language to turn both loop iterations and data arrays into push memories expressed using the unified buffer abstraction presented in Chapter 3. That is, this application specific loop nest describes computation as operations on arrays over iteration domains defined by loop variables. The iteration domain is the Cartesian product of the bounds of the loops surrounding the operations in the loop nest. For instance, according to the code in Figure 4.1a line 4-5, the `brighten` operation is encapsulated under a two level loop nest with loop bound equal to 64. So the end of the loop will be $start + bound - 1 =$

```
//Outer tiling loop
for t in [0:16]
  //Brighten
  for y in [0:64]:
  for x in [0:64]:
    brighten(x, y) = brighten_kernel( input(x, y, t) );

  //Blur
  for y in [0:63]:
  for x in [0:63]:
    blur(x, y, t) = blur_kernel( brighten(x, y),
                                 brighten(x+1, y),
                                 brighten(x  , y+1),
                                 brighten(x+1, y+1));
  //Definition of blur kernel
  function blur_kernel(arg0, arg1, arg2, ar3):
    arg_st = (arg0 + arg1 + arg2 + arg3)/4;
    return arg_st;

  //Definition of brighten kernel
  function brighten_kernel(arg0):
    arg_st = arg0 * 2;
    return arg_st;
```

(a)

```
//Outer tiling loop
prog prg("br-bl");
ir_node* t = prg.add_loop("t", 0, 16);

//Brighten sub-loop
ir_node* y_br = t->add_loop("y_br", 0, 64);
ir_node* x_br = y_br->add_loop("x_br", 0, 64);
ir_node* brighten_op = x_br->add_op("Brighten");

//Brighten OP
brighten_op->add_function("brighten_kernel");
brighten_op->add_load("input", "x_br","y_br", "t");
brighten_op->add_store("brighten", "x_br","y_br");

//Blur sub-loop
ir_node* y_bl = t->add_loop("y_bl", 0, 63);
ir_node* x_bl = y_bl->add_loop("x_bl", 0, 63);
ir_node* blur_op = x_bl->add_op("Blur");

//Blur OP
blur_op->add_function("blur_kernel");
blur_op->add_load("brighten", "x_bl", "y_bl");
blur_op->add_load("brighten", "x_bl+1", "y_bl");
blur_op->add_load("brighten", "x_bl", "y_bl+1");
blur_op->add_load("brighten", "x_bl+1", "y_bl+1");
blur_op->add_store("blur", "x_bl", "y_bl", "t");
```

(b)

Figure 4.1: (a)Loop nest for brighten blur pipeline. The brighten kernel is multiplying the input by two, and the blur kernel is calculating average among all four input arguments. (b) The corresponding DSL that constructs the loop nest in Figure 4.1a

.

$0 + 64 - 1 = 63$. Therefore the iteration domain of brighten operation is defined as

$$\{\texttt{Brighten}(x, y, t) | 0 \leq x \leq 63, 0 \leq y \leq 63, 0 \leq t \leq 15\}$$

whereas the consumer operation `blur` has a smaller iteration domain with one less on the boundary because of the halo of 2x2 window,

$$\{\texttt{Blur}(x, y, t) | 0 \leq x \leq 62, 0 \leq y \leq 62, 0 \leq t \leq 15\}$$

Each memory read operation or write operation is given a unique logical port on the corresponding unified buffer. The number of logical ports indicates the bandwidth required to schedule the operations. For each logical port, buffer extraction then extracts an access map from the buffer accessing address expression. For example, the `brighten` kernel will store the calculated pixel value into the *brighten* buffer while the `blur` kernel will load four pixels from the same buffer to compute the final output of the pipeline. From the address expression denoted in Figure 4.1b, the access map for the *brighten* buffer write is

$$\{\texttt{Brighten(x,y,t)} \rightarrow brighten(x, y)\} \tag{4.1}$$

The Blur kernel read operations go through four logical ports to the brighten buffer, and the access maps are

$$\{\texttt{Blur(x,y,t)} \rightarrow brighten(x, y)\} \tag{4.2}$$

$$\{\texttt{Blur(x,y,t)} \rightarrow brighten(x + 1, y)\} \tag{4.3}$$

$$\{\texttt{Blur(x,y,t)} \rightarrow brighten(x, y + 1)\} \tag{4.4}$$

$$\{\texttt{Blur(x,y,t)} \rightarrow brighten(x + 1, y + 1)\} \tag{4.5}$$

Using this DSL and information extracted from the Halide compiler makes it easy to generate the iteration domain and access map for each of the *brighten* unified buffer ports depicted in Figure 4.2. The main work of unified buffer extraction, however, is computing the cycle-accurate schedule that maps operations in the loop nest to the cycle times specifying when they will happen in hardware.

4.2 Scheduling

Unlike traditional polyhedral schedulers that create an optimized set of loops, our scheduling approach is designed to capture the runtime behavior of individual computation operation. This

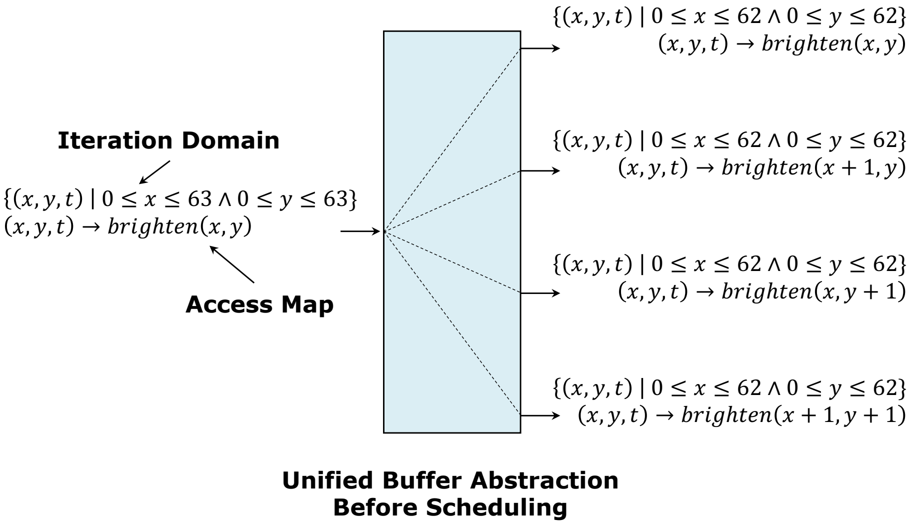

Iteration Domain
$\{(x,y,t) \mid 0 \le x \le 63 \land 0 \le y \le 63\}$
$(x,y,t) \to brighten(x,y)$
Access Map
$\{(x,y,t) \mid 0 \le x \le 62 \land 0 \le y \le 62\}$
$(x,y,t) \to brighten(x,y)$
$\{(x,y,t) \mid 0 \le x \le 62 \land 0 \le y \le 62\}$
$(x,y,t) \to brighten(x+1,y)$
$\{(x,y,t) \mid 0 \le x \le 62 \land 0 \le y \le 62\}$
$(x,y,t) \to brighten(x,y+1)$
$\{(x,y,t) \mid 0 \le x \le 62 \land 0 \le y \le 62\}$
$(x,y,t) \to brighten(x+1,y+1)$
Unified Buffer Abstraction
Before Scheduling

Figure 4.2: Iteration domain and access map for unified buffer *brighten*.

choice steams from the fact that hardware accelerators typically have a massive amount of concurrent compute resources that can be executed in overlap with others, which are challenging to represent efficiently using imperative loop. To address this challenge, our scheduler maps each operation's iteration space to a one-dimensional timestamp. This timestamp-based schedule simplifies the representation of pipeline parallelism, enabling each computation stages to execute on a separate hardware unit in a pipelined fashion.

4.2.1 Scheduler Setup

Many image processing and machine learning applications have statically analyzable access patterns. This characteristic enables our scheduler to schedule all memory accesses at compile time. The primary objective of our scheduler is to minimize the application's latency while adhering to essential constraints related to dependencies and resource availability. For instance, consider a scenario where a consumer operation, denoted as op_{cons}, depends on data produced by another operation, op_{prod}. In this case, the scheduler must allocate a start time for op_{cons} after the completion of op_{prod}. This relationship between the start time T_{start} and completion time T_{end} of an operation can be expressed as follows, with L representing the latency of the specific node within the loop nest:

$$T_{end} = L + T_{start} \tag{4.6}$$

As we described in Section 4.1, there are two type of nodes, a *loop* and an *op*, in the loop nest that the frontend compiler generates. The latency of a *op* node can be defined by the following equation, where $L_{compute}$ represents the compute latency, and L_{mem_ld}, L_{mem_st} represent the memory load and store latency respectively.

$$L_{op} = L_{mem\_ld} + L_{mem\_st} + L_{compute} \tag{4.7}$$

In order to form a cycle accurate schedule, each node was assigned a delay within its parent node in the loop nest IR. It specifies the absolute time offset with respect to the start of its parent node. There is another attribute that is specific to the *loop* node, the initiation interval (II), which defines the time duration before starting of the next loop iterations. These attributes are essential components that the scheduling algorithm utilize to orchestrate the precise timing of the operation. Computing these II values is one of main tasks for the scheduling algorithm.

From these two attributes, we can calculate the latency of the sub-loopnest under this node. The latency of a loop is defined as II multiplied with loop trip count minus one added with single loop iteration latency. And the loop iteration latency is the max of delay + latency among all children under this loop.

$$L_{loop} = II * (trip\_cnt - 1) + L_{loop\_iter} \tag{4.8}$$

$$L_{loop\_iter} = \max_{\forall i \in loopchildren} (D_i + L_i) \tag{4.9}$$

It is important to note that the child node could be either a loop or an operation. Consequently, the term L_i will be substituted with equation specified in Equation 4.7 when it refers to an *op* node, or will apply the Equation 4.8 recursively if it's a *loop* node. The next several sections will provide insights into how our scheduler enhances performance by reducing total latency through the computation of optimal delay and initiation interval (II) using software pipelining and polyhedral analysis. Importantly, this is achieved while maintaining resource constraints and preserving operation dependencies.

4.2.2 Hardware Constraints

An essential input to the scheduler, which significantly influences its schedule process, are the physical hardware constraints. These constraints are vital during the compilation process, as they guide the scheduler in making optimized schedule decisions tailored to the specific target hardware. These hardware specific constraints encompass a number of factors, including

- Latency of each mapped compute kernel (including both processing element and interconnect)
- Number of physical memory ports
- Load, store latency of memory and fetch width of the SRAM macro used by the memory
- Lowest initiation interval(II) of compute, memory and interconnect devices

This information helps determine the innermost initiation interval of each operation and delay needed before each operation's output becomes valid. More details about how these constraints are passed to the scheduler are given in Section 6.3. By incorporating such constraints, the unified buffer extracted in this stage can be configured to map to the target physical hardware in a unified process. This configuration flexibility ensures that the scheduler's decisions align with the hardware's capabilities. As a result, not only does it ensure correct functionality, but it also leads to improved performance and resource utilization.

4.2.3 Dependency Graph

During the buffer extraction stage, we generated a dependency graph to reason about the data dependencies among operations. This graph is a directed data structure, illustrating how one operation in the system relies on data produced by another operation, with the consumer pointing to the producer. Given that tensor operation access patterns are statically analyzable, we can compute data dependencies by intersecting the access maps of various operations that access the same data array. For example, in the brighten-blur application, `brighten` operation write to the intermediate buffer while `blur` operation will consume it. By applying the access relation from Equation 4.1 on the right hand side of the access map of blur Equation 4.5, we can derive the data dependency as follow

$$\{\texttt{Blur(x,y,t)} \rightarrow \texttt{Brighten(x,y)}\} \tag{4.10}$$

$$\{\texttt{Blur(x,y,t)} \rightarrow \texttt{Brighten(x+1,y)}\} \tag{4.11}$$

$$\{\texttt{Blur(x,y,t)} \rightarrow \texttt{Brighten(x,y+1)}\} \tag{4.12}$$

$$\{\texttt{Blur(x,y,t)} \rightarrow \texttt{Brighten(x+1,y+1)}\} \tag{4.13}$$

As a result, the graph representation of the data dependency is shown in Figure 4.4. This data structure is useful when optimizing the schedule for hardware pipeline.

4.2.4 Variable Latency Support

Our schedules guarantee that data dependencies are not violated, assuming the input data is valid and the output data can be stored. Although our application is statically analyzable, all hardware accelerators have to deal with variable-latency operations like main memory accesses. To remove these issues, almost all accelerators transfer memory data to a local buffer before starting operation. However, if the operation is tiled, the accelerator is usually scheduled to transfer the next tile into the buffer while it is operating on the current buffer. In these situations it is still possible for this memory transfer to run "late" and not be ready at the start of the next tile iteration which stalls the start of the next statically scheduled tile. To accomodate these possible stalls, the schedules count unstalled cycles instead of absolute clock cycles.

In our target CGRA, the interface between the accelerator and the host memory system uses latency-insensitive channels, while the memory inside the CGRA uses (gated) cycle counters. Our Halide program tiles the inputs to create execution blocks that our compiler statically schedules. All statements within the tiled execution block are assigned a timestamp consistent with the global cycle accurate schedule. This context information ensures the scheduler adds enough buffering to allow internal compute kernel nodes to read the data from multiple predecessors simultaneously even if they are produced at different times. Between the tiled execution, we use double-buffered ready-valid channels. Thus, we only need to stall if the next tile has not been prefetched from DRAM into the accelerator, or the previous tile output has not been stored in DRAM before the current tile stops execution. For a CGRA implementation using ready-valid channels with buffet [67] style memory blocks that contain dependency checking capabilities, our compiler outputs address patterns and drops the schedule information for read after write dependency[1]. It thus lets the hardware handle execution timing and potential port conflicts.

4.3 Loop Pipeline

Figure 4.1a gives the pseudo code for brighten-and-blur kernel. A valid but not fully-optimized schedule is to run this loop nest sequentially, just like running it on a single threaded CPU. In this scenario, the scheduler assigns a delay to each child node equal to the cumulative latency of its predecessor. Simultaneously, it sets the initiation interval (II) of the loop to match the total latency of its loop body. As an example, considering the brighten and blur workload, we cannot start the next loop iteration t until the previous blur computation has completed. Assuming a zero latency when computing the brighten and blur kernels (which of course is unrealistic , but simplifies the discussion), this lead to a delay of $64 \times 64 + 63 \times 63 = 8065$ cycles before starting the next t iteration. We refer to this schedule strategy as a sequential schedule, which is the baseline for scheduling optimizations. Later in this section we will use this to show how iterative modulo scheduling, loop fattening, and loop perfection can be used to transform this base into an optimized schedule.

Different from general purpose platform, domain specific accelerators have massive amount of computation resources, and dedicated compute units can be assigned for different computation operations. For most systems the brighten kernel and blur kernel would be implemented using distinct hardware component. Therefore, a hardware pipeline is more efficient, which overlaps the brighten kernel from iteration $t = t_0 + 1$ with blur kernel from iteration $t = t_0$.

In practice, the frontend compiler will generate a loop nest with a loop level to identify the target pipeline interleaving granularity. By default, it will be the for loop with more than one child. To

[1] The read after update dependency is still kept, since buffet does not monitor these dependency in its hardware controller. We simply count the number of update then enable the read.

identify the tile loops that can be overlapped, the buffer extraction process walks from the root of the program to the leaves, collecting all the encountered loop nodes until it gets the first loop whose body is not a perfect loop (contains more than one child). These perfectly nested loops form the outer loops of the coarse-grained pipeline. We refer to the operations inside the pipeline loop as pipeline stages, but these stages are themselves typically an operation wrapped by a number of loops. Therefore, we formally define the pipeline stage as the block of operations wrapped by the subloops between the coarse-grained pipeline loop and the operation statement.

For instance, in the brighten-blur pipeline pseudo code shown in Figure 4.1a, the outer coarse-grained loop, the `for` loop on line 2, is pipelined and it contains two pipeline stages, brighten and blur sub-loops respectively. Since the brighten kernel produce the input for blur kernel, the dependency graph has two nodes with brighten kernel feed into the blur kernel as is illustrated in Figure 4.4. A more complex example is shown in Figure 4.3. This figure shows the loop structure in a multi-channel 3x3 convolution loop nest in used in many DNNs. In this example, the pipeline loop encompasses six stages, including initiation of the output value, loading of input from the global buffer (GLB) into memory, loading of weights from the GLB into memory, two parallel convolution computations and transfer of the output value stored in local memory to the GLB, which is shown in the dependency graph in Figure 4.10.

4.3.1 Iterative Modulo Scheduling

To create pipelined hardware schedules from this information, we extended iterative modulo scheduling [72], a compilation technique to enable software pipelining. It constructs a static schedule for all operations under a loop iteration so that the same schedule can be repeated with respect to resource constraints and data dependence. The constant interval between the start of successive iterations is the initiation interval (II), which is one of the parameters needed to compute the cycle accurate schedule.

In order to model the resource utilization within this static schedule, a resource reservation table is built to record that a specific resource is used by particular operation at every timestamp. Normally the modulo scheduling target operations under single loop iteration. In our cases, we are dealing with operations wrapped by multi-dimensional loop nests, which is defined as pipeline stages. Therefore, we make the assumption that **once the resource is assigned to that operation, it will be reserved for the entire duration of execution period**. To keep track of the reservation duration, we maintain a column in the table that records the latency after the initiation of that particular operation.

Similar to the resource reservation table for a CPU which contains of ALU and registers, our resource modulo table for hardware accelerators contains **compute elements** and **storage elements**. Figure 4.5 shows the resource reservation table for brighten and blur pipeline shown in Figure 4.1a.

```
//Outer tiling loop
for t in [0:16]
  //Initial output bias from GLB to CGRA Memory
  for y in [0:28]:
  for x in [0:28]:
  for k in [0:2]:
    output_mem(k, x, y) = init_bias_GLB(k, x, y, t);

  //load input from GLB to CGRA Memory
  for y in [0:30]:
  for x in [0:30]:
  for c in [0:4]:
    input_mem(c, x, y) = input_glb(c, x, y, t);

  //load weight from GLB to CGRA Memory
  for ky in [0:3]:
  for kx in [0:3]:
  for k in [0:2]:
  for c in [0:4]:
    weight_mem(c, k, kx, ky) = weight_glb(c, k, kx, ky, t);

  //3x3 Conv with 4 input channels and 2 output channels
  for ky in [0:3]:
  for kx in [0:3]:
  for y in [0:28]:
  for x in [0:28]:
    output_mem(0, x, y) =
        CONV_0(input_mem(0:4,x,y), weight_mem(0:4, 0, kx, ky), output_mem(0, x, y));
    output_mem(1, x, y) =
        CONV_1(input_mem(0:4,x,y), weight_mem(0:4, 1, kx, ky), output_mem(1, x, y);

  //drain output from CGRA Memory to GLB
  for y in [0:28]:
  for x in [0:28]:
  for k in [0:2]:
    output_glb(k, x, y, t) = output_mem(k, x, y);
```

Figure 4.3: Loop nest for the multi channel 3x3 convolution DNN layer. It has 4 input channels and 2 output channels. The computation is fully unrolled (parallel) along the input channel dimension `c` and the output channel dimension `k`. The 3x3 kernel dimension is calculated sequentially.

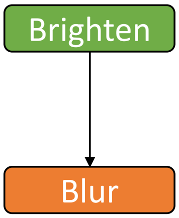


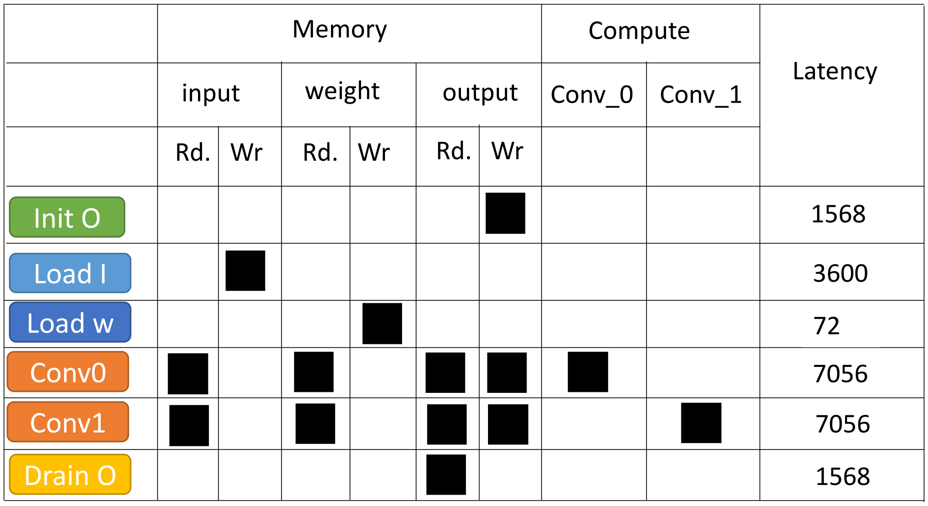
Figure 4.4: Dependency graph of brighten and blur pipeline.

	Storage						Compute		Latency
	input		brighten		blur		Brighten	Blur	
	Rd.	Wr	Rd.	Wr	Rd.	Wr			
Brighten	■			■			■		4096
Blur			■			■		■	3969

Figure 4.5: Resource Reservation Table for Brighten and Blur Pipeline.

This application has three unified buffers and two compute units. Accounting for the varying number of read and write ports present in memory devices, the resource reservation table distinguish read operations from write operations. Figure 4.5's, resource reservation table assumes that the underlying unified buffers have a single read port and a single write port. Thus we have one column for both read and write operation in each unified buffer. However, it's important to note that the blur operation, for instance, reads four pixels to produce a result, which imposes a higher memory port resource requirement to maintain that bandwidth. Given our reconfigurable hardware backend, we have the flexibility to partition data into multiple physical memories to increase bandwidth. Therefore, even for memory operations with high bandwidth requirements, a single entry in the resource reservation table is taken. Further details regarding memory bandwidth requirements will be discussed in the next section.

After constructing the resource reservation table, we will start scheduling for a specific hardware accelerator architecture model. We assign a timestamp to each operation's issuing and create a schedule reservation table, which records the usage of a particular resource by an operation at a specific time. This new data structure is distinct from the resource reservation table, which only captures the occupied duration of each resource. The schedule reservation table, on the other hand, clearly state the start and end time of resource reservation for all operations. The scheduling is considered as valid only if there are no resource conflicts. To ensure this, the schedule reservation



	Memory						Compute		Latency
	input		weight		output		Conv_0	Conv_1	
	Rd.	Wr	Rd.	Wr	Rd.	Wr			
Init O						■			1568
Load I		■							3600
Load w				■					72
Conv0	■		■		■	■	■		7056
Conv1	■		■		■	■		■	7056
Drain O					■				1568

Figure 4.6: Resource Reservation Table for Multi-Channel Convolution Layer.

table is used to check for overlapping resource usage between different operations. If there are no overlapping time period where the same resource is reserved by multiple operations, the scheduling is deemed legal and can proceed. Because this scheduled loop nest will be re-executed with period of initiation interval (II) of the loops outside of the pipelining, if scheduling the operation at a specific timestamp t involves the resource R, then the entry ($t \mod II, R$) of the table will be marked as occupied. As a result, the schedule reservation table only need to record the period of the II. Such scheduling reservation table is called modulo reservation table(MRT)[27].

To find the II of this outer loop, we will create an empty modulo reservation table with the minimum possible II. The lower bound of II will equal to the maximum occupied duration among all resources. In order to get the lower bound of II, we go over each column of the resource reservation table and calculate the total latency of all operations that use this resource. For instance, in the Brighten Blur pipeline shown in Figure 4.1a, according to the hardware constraints, we set the innermost loop $II = 1$, and the corresponding total latency of brighten kernel is

$$Loop\_bound_y * (II_x * Loop\_bound_x + D_x) + D_y = 64 \times (64 \times 1 + 0) + 0 = 4096$$

cycles and the latency of blur kernel is

$$Loop\_bound_y * (II_x * Loop\_bound_x + D_x) + D_y = 63 \times (63 \times 1 + 0) + 0 = 3969$$

cycles respectively. The blur kernel takes shorter period of time because the halo of the 2x2 convolution window. Then we can get the minimum II of the the pipeline loop is $max(4096, 3969) = 4096$. Next we perform a topological sort of all the operations in the dependency graph, and schedule

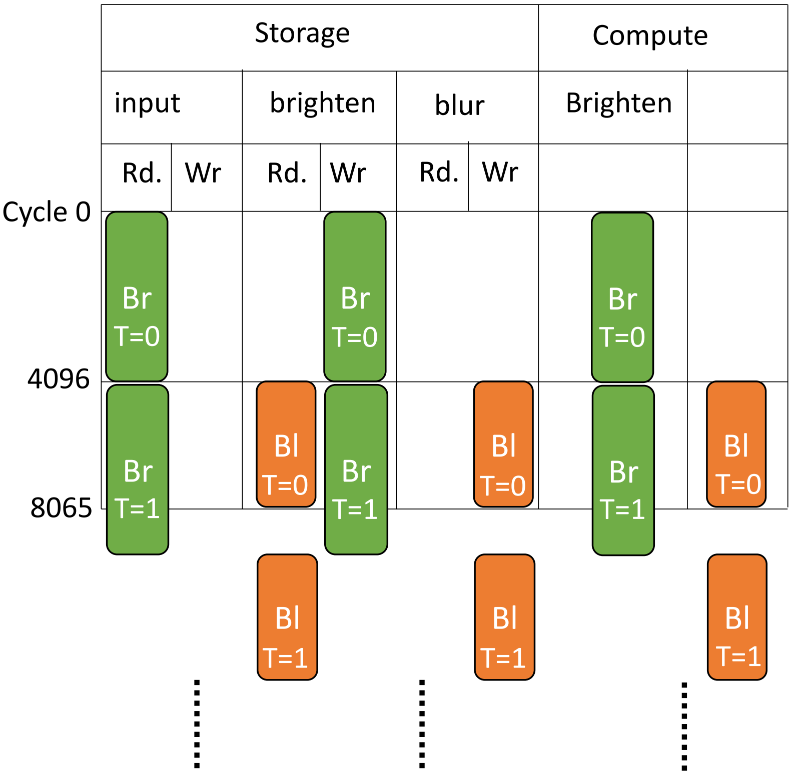


Figure 4.7: Schedule Reservation Table for Brighten and Blur Pipeline.

the operations at the earliest possible time. Once we have seen a resource conflict, we abort the current scheduling process and increase the II. For the brighten and blur pipeline example, we can sequentially schedule the brighten and blur kernel from the same iteration in loop t. The schedule reservation table is illustrated in Figure 4.7 Notice that the II of the t loop in Figure 4.1a is 4096, so the brighten computation for the next tile of input data will start instantly as we start the blur kernel from the previous iteration. The modulo reservation table in Figure 4.8 captures the resource occupation in steady state of the pipeline, which is much more succinct. Upon the successful scheduling of every pipeline stage without any resource conflict, the modulo schedule is completed and the cycle accurate schedule for each operation is generated. The schedule is then distributed to the ports of each unified buffer, allowing for the identification of the execution timing for each memory operation. The cycle accurate schedule we get is

$$Brighten(x, y) \rightarrow [4096t + 64y + x]$$

$$Blur(x, y) \rightarrow [4096 + 4096t + 63y + x]$$

The *Brighten* operation writes the unified buffer *brighten*, and the *Blur* operation reads from the

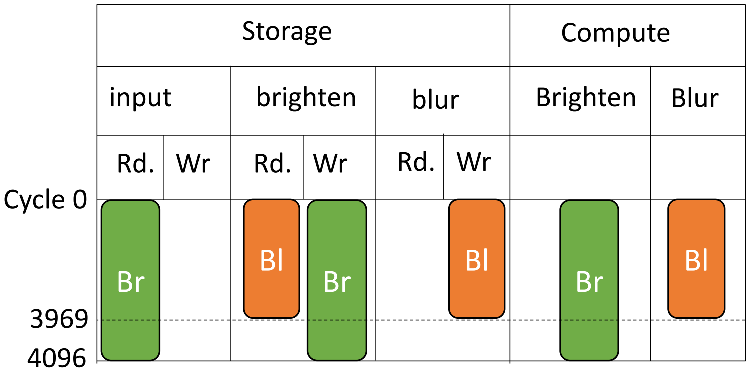


Figure 4.8: Modulo Reservation Table for Brighten and Blur Pipeline.

same buffer. Thus the schedule of *Brighten* operation is used to set the timing of the write port while the schedule of *Blur* operation sets the timing of all four read ports of unified buffer *brighten*, as is depicted in Figure 4.9.

4.3.2 Memory Bandwidth Requirement

To enable parallel computation, frequently more than one entry in the data array or tensor are accessed simultaneously. Thus the available memory bandwidth significantly impacts the latency of each operation that is scheduled. In our scheduler, we assume that our abstract unified buffers possess sufficient bandwidth to accommodate high-throughput data reads. The problem of supplying the needed bandwidth falls to the buffer mapping stage of the compiler pipeline. If the hardware units don't have sufficient bandwidth, it uses memory banking Section 5.2 to increase the effective storage element bandwidth.

These high bandwidth requests can originate from the **same pipeline stages** or from those that are **isomorphic** in the dependency graph. The first scenario is relatively straightforward to understand. For instance, considering the `Blur` kernel in Figure 4.5 and the memory implemented with dual port SRAM macro, which load four pixels from the brighten buffer to create the blurred output. Without the above assumption, data loads sequentially from the single memory read port. However, leveraging the flexible interconnect and abundant distributed on-chip buffers on the CGRA, data could be duplicated into four different memory banks to support concurrent read from the memory. Often, other optimizations which are described in Section 5.1 greatly reduce the number memory banks required. The scheduler is aware of these low level architecture optimizations, so that this high bandwidth memory operation only occupies a single resource entry in the resource reservation table shown in Figure 4.5.

A more complex example involve parallel memory access from multiple operations as occurs in the DNN convolution example shown in Figure 4.3. In particular, as depicted in Figure 4.6, the

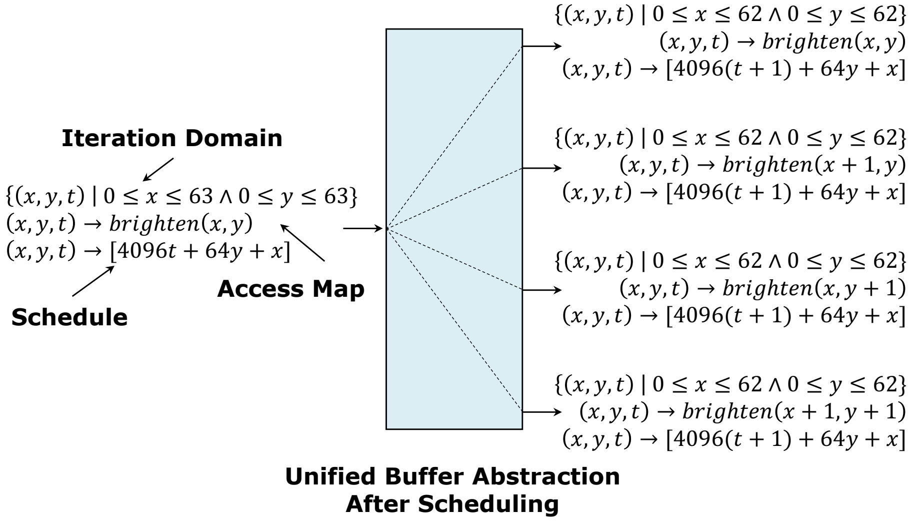

Iteration Domain
$\{(x,y,t) \mid 0 \le x \le 63 \wedge 0 \le y \le 63\}$
$(x,y,t) \to brighten(x,y)$
$(x,y,t) \to [4096t + 64y + x]$
Access Map
Schedule
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x,y)$
$(x,y,t) \to [4096(t+1) + 64y + x]$
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x+1,y)$
$(x,y,t) \to [4096(t+1) + 64y + x]$
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x,y+1)$
$(x,y,t) \to [4096(t+1) + 64y + x]$
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x+1,y+1)$
$(x,y,t) \to [4096(t+1) + 64y + x]$
Unified Buffer Abstraction
After Scheduling

Figure 4.9: The unified buffer *brighten* with schedule.

resource reservation table indicate that both pipeline stage `Conv0` and `Conv1` reserve both the read and write port of unified buffer *output* for accumulating the partial result. A naive approach would not allow the concurrent issuance of these two pipeline stages. The resulting schedule, is shown in Figure 4.11, forces the `Conv1` stage to be scheduled before `Conv2`.This schedule results in the minimum II equals to the total latency of two convolution stages and the draining of the output data.

However, as shown in Figure 4.10, which is the dependency graph of CNN pipeline after topological sort, pipeline stage *CONV0* and *CONV1* don't consume or produce each other's outputs. As a result, those statements can be schedule at the same timestamp as long as we can create memory with higher bandwidth[2] to service these kernel in parallel. However, reading and writing from non-isomorphic pipeline stages will require extra bandwidth that cannot be addressed by banking. For instance, the output memory must have two different writing phases, *InitO* phase is to initialize the output value; the *CONV0, CONV1* phase will write the updated output values into the same memory. These two operations require two unique write port for that storage element. As a result, after taking the memory banking into consideration, the schedule reservation table in Figure 4.12 indicates the optimized minimum II equals to the total latency of one convolution computation and the output data draining. Notice that the loading and draining of output could be further overlapped if we use a memory have two input ports and two output ports memory.

[2] Known as memory banking in computer architecture

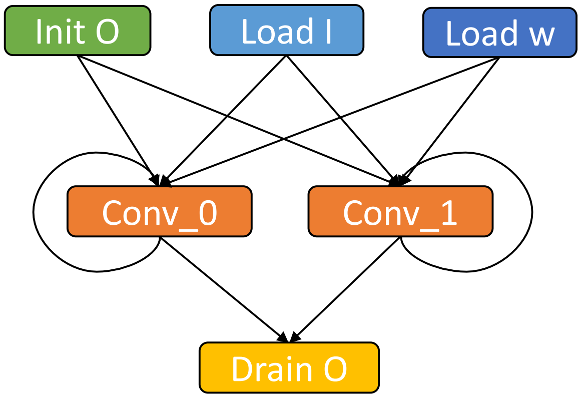


Figure 4.10: Topological Sort Result of the Convolution Layer Loop Nest in Figure 4.3.

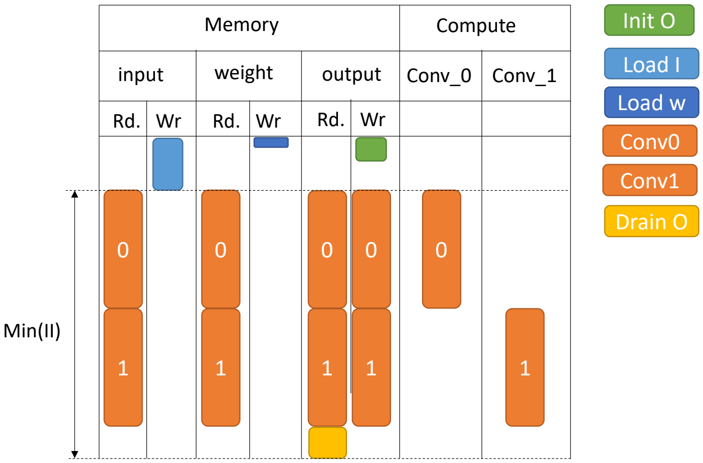


Figure 4.11: Schedule Reservation Table for Multi-Channel Convolution Layer.

4.3.3 Resource Sharing

During the procedure of loop pipelining, there is a significant challenge to balance the consumption of compute resource while optimizing for both latency and throughput. To achieve maximum throughput, it seems logical to allocate dedicate hardware for each operation in the loop nest. However, this approach can lead to situations where the design exceeds the available computation resources, or where certain hardware is under-utilized due to imbalances in workload across different pipeline stages. Thus, our compiler provides the user the ability to indicate when hardware resources should be shared, which can help reduce the overall hardware required for a pipeline and, in some cases, improve hardware utilization.

The loop nest in Figure 4.13 illustrates a simple gaussian pyramid application. These multi-scale data structures are often used in image processing applications. The pipeline starts with loading the

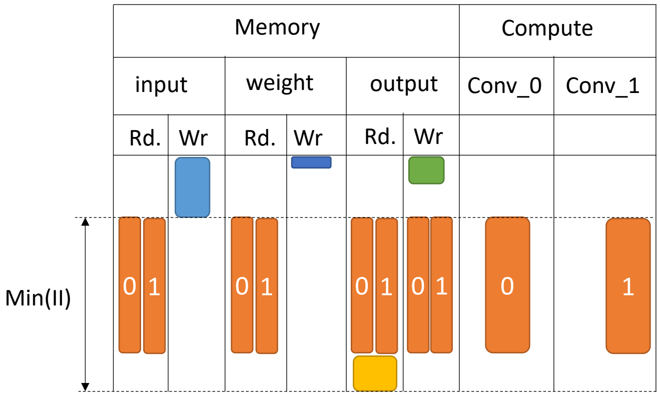


Figure 4.12: Schedule Reservation Table for Multi-Channel Convolution Layer with memory bandwidth optimization.

input onto CGRA memory tile, followed by three 2x2 average kernels that perform down sampling. Notably, the latency of each kernel are decreasing four times as we progress through the pipeline, as the image being down sampled by 2 both horizontally and vertically. Assigning dedicated compute hardware for each pyramid kernel, as is shown in Figure 4.14, results in a modulo schedule which indicates that the overall pipeline latency is dominate by data transfer, and the pyramid computation hardware is under utilized. In order to improve the hardware utilization and minimize the resource consumption, the three consecutive Gaussian pyramid kernel can share the same hardware, given that they employ the same arithmetic operations. As depicted in Figure 4.15, the modulo schedule after resource sharing remains unchanged in terms of the latency and throughput, while reducing the number of pyramid compute unit required from three, Figure 4.14, to one.

4.3.4 Optimizations: Loop Perfection and Flatten

Facilitating effective loop pipelining is critical while constructing a schedule using an HLS-style scheduler. A conventional HLS loop scheduler uses `for` loops as boundaries of pipelining[16]. Nested and imperfect loops contain multiple such boundaries, giving rise to the occurrence of redundant pipeline flush stages at the end of each loop [89, 70].

The same rationale applies to our coarse-grained pipeline. To mitigate the additional latency introduced by pipeline flushes, the buffer extraction stage incorporates loop perfection and loop flattening before applying the iterative modulo schedule, as described in the previous section. Loop perfection condenses all coarse pipeline stages within the innermost `for` loop with `if` guards. Meanwhile loop flattening consolidates all loops above the coarse-grained pipeline into one single, merged loop. For instance, as is depicted in Figure 4.16a, there is a pipeline flush overhead with each iteration of outer loop `x`. This example shows the simplest example, where each coarse grained pipeline stage is a single operation, in practice, this could be more complex, involving multi-level nested

```
//Outer tiling loop
for t in [0:16]
  //load input
  for y in [0:64]:
  for x in [0:64]:
    input(x, y) = GLB_I(x,y)

  //pyramid 1
  for y in [0:64]:
  for x in [0:64]:
    p1(x, y) = [input(2x,2y) + input(2x+1, 2y)
              + input(2x, 2y+1) + input(2x+1, 2y+1)]/4

  //pyramid 2
  for y in [0:32]:
  for x in [0:32]:
    p2(x, y) = [p1(2x, 2y) + p1(2x+1, 2y)
              + p1(2x, 2y+1) + p2(2x+1, 2y+1) ]/4

   //pyramid 3
  for y in [0:16]:
  for x in [0:16]:
    p3(x, y) = [p2(2x, 2y) + p2(2x+1, 2y)
              + p2(2x, 2y+1) + p2(2x+1, 2y+1) ]/4
```

Figure 4.13: Loop nest for 3 Level Gaussian Pyramid

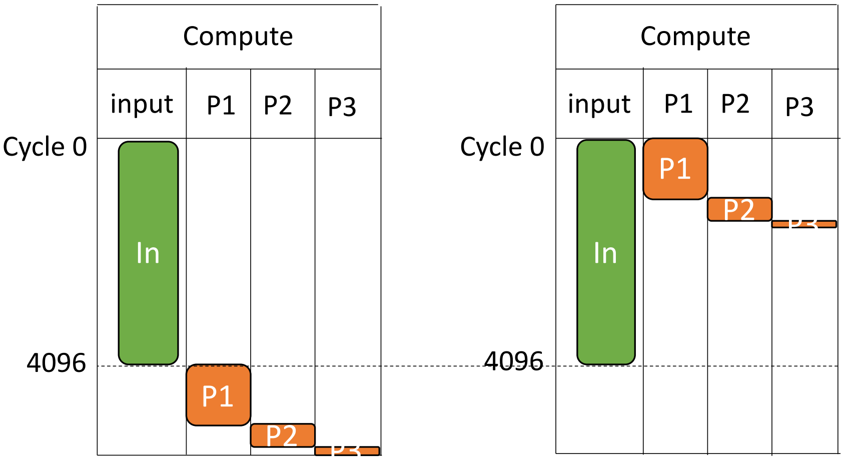


Figure 4.14: Modulo Reservation Table for Gaussian Pyramid Application in Figure 4.13. The Memory Part is left out. The left diagram depicts the schedule reservation table, while the right diagram depict the modulo reservation table with the coarse grained II = 4096.

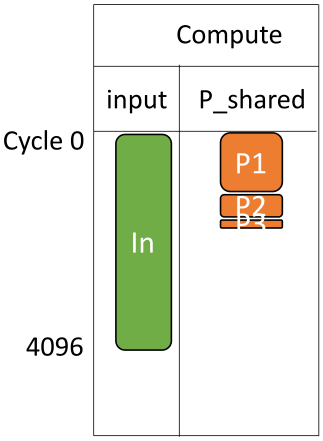


Figure 4.15: Modulo Reservation Table for Gaussian Pyramid Application with all pyramid kernels shared the same compute kernel.

```
for x in [0:32]:
  out[x] = 0;
  for dx in [0:5]:
    out[x] += filter[dx] * mem[x+dx];
  DRAM.store(out, x);
```

(a) Original loop nest before loop perfection.

```
for x in [0:32]:
  for dx in [0:5]:
    if (dx == 0)
      out[x] = 0;
    out[x] += filter[dx] * mem[x+dx];
    if (dx == 4)
      DRAM.store(out, x);
```

(b) Loop nest after loop perfection.

```
for x_flatten in [0:160]:

  //Reconstruct the original loop iteratiors
  x = x_flatten / 5;
  dx = x_flatten % 5;

  //Computation logic
  if (dx == 0)
    out[x] = 0;
  out[x] += filter[dx] * mem[x+dx];
  if (dx == 4)
    DRAM.store(out, x);
```

(c) Loop nest after loop flatten.

Figure 4.16: Example for a 1D convolution loop nest applying HLS scheduling optimizations, including loop perfection and loop flatten.

Algorithm 1: Loop Pipeline Algorithm

```
Function LoopPipeline($prg, cst$):
    $prg \leftarrow$ RunOpt($prg$, {LoopPerfection, LoopFlatten})
    $(V, E) \leftarrow$ BuildDependencyGraph($prg$)
    $rrt \leftarrow$ BuildResourceReservationTable($prg, cst$)
                    ▷ $rrt$ records all resource subscription and correponding latency
    $II \leftarrow$ GetPipelineII($rrt$)
    $mrt \leftarrow$ InitModuloReservationTable($rrt, II$)
                    ▷ $mrt$ records the schedule information
    for $kernel \in$ TopoSort($V, E$) do
        $est \leftarrow$ FindEarliestStart($kernel, (V, E), mrt.rrt$)
        ModuloIterativeSchedule($kernel, est, mrt, rrt$)
    end
    return CalculateSched($mrt$)
                    ▷ return the cycle accurate schedule from Modulo Reservation Table
```

loop. To establish a fully pipelined loop with a single level, both the initialization and the DRAM store operation was pushed into the inner most `dx` loop with an if guard as shown in Figure 4.16b. Finally, Figure 4.16c demonstrates the loop structure after loop flattening, where all operations are emcompassed within a single-level loop. It is important to mention that the original loop iterations are reconstructed using floor-divide and modulo operators. By combining loop perfection and loop flattening, our scheduler can generate a fully pipelined schedule with minimum pipeline flush overhead.

4.3.5 Putting it All Together

The Loop Pipeline Algorithm, illustrated in Algorithm 1, encapsulates a function that optimizes schedule, which takes the loop-level intermediate representations (prg) and hardware constraints (cst). This algorithm plays a pivotal role in reducing latency and improving resource utilization for scheduling an application on push memory backend.

Initially, the loop perfection and flatten optimization, described in Section 4.3.4 is applied on the loopnest's intermediate representation. Subsequently, a dependency graph (V, E), introduced in Section 4.2.3, is constructed based on the application's producer-consumer relations. Along with hardware constraint information, the data dependency graph is further employed to generate a resource reservation table (rrt). This table records resource requirements and corresponding latencies based on the hardware constraints, forming a critical auxiliary data structure in the following scheduling phases. A key step involves the determination of the coarse grained loop initiation interval (II) through analysis of the resource reservation table (rrt), followed by the initialization of the modulo reservation table (mrt). This table captures essential scheduling information, orchestrating the subsequent iterative modulo scheduling procedures.

The scheduling phase iterates through the coarse grained pipeline stages ($kernel$) of the program in topologically sorted order. For each stage, the algorithm identifies the earliest start time (est) based on dependencies and resource availability. Leveraging this information, the Modulo Iterative Scheduling determines the delay of each kernel, introduced in the Section 4.3.1, utilizing the established schedules in the modulo reservation table (mrt) and resource reservation table (rrt). Finally the schedule information in modulo reservation table(mrt) generates a cycle-accurate schedule via the `CalculateSched()` function, providing an optimized schedule for the application (prg).

Overall, the Loop Pipeline Algorithm serves as an important component in the unified buffer extraction process, which takes hardware constraints into consideration. It utilizes iterative modulo scheduling techniques, creating schedule which optimizes resource utilization in push memory accelerators.

4.4 Loop Fusion

While loop pipeline algorithm reduces the interval between consecutive loop iterations, it does not change the relative order between operations. To further reduce the computation latency and the intermediate memory capacity, loop fusion can be applied between producer consumer loop nests, which essentially brings the consumer closer to the producer in the same iteration.

Leveraging previous work proposed by Huff et al.[38], the fusion procedure combines loop nests in an application into a single loop nest using a static data flow style constraint problem. It sets the relative II (II_r) and delay of each operation to minimize dependence distances while ensuring uniformity. To be specific, a global schedule is constructed with the constraint that the start of a statement must happen after the latest end time among all of its producer statements. The end time of a statement is calculated using the start time, the latency of memory loads, the latency of memory store, and the latency of its computation. This is demonstrated in the following equations, where the start and end time of a operation is defined in Equation 4.6 and Equation 4.7:

$$Start(\texttt{stmt}) > End(\texttt{prod}) \quad \forall \texttt{prod} \in \texttt{stmt}'s\ producers \tag{4.14}$$

The fusion is done incrementally, starting from the outermost loop level to an inner loop level specified by the user, which is called *loop fusion level.* The loop fusion level determines the granularity of interleaving the pipeline stages. Once fusion is finished, we use the loop pipeline algorithm described previously to form a cycle-accurate schedule for the inner imperfect loop nest, which sets the II and delay of all loops and operations under fused loop level. Since the fused loop nest is pipelined at the imperfect loop level, a lower level means finer interleaving granularity, which reduces the pipeline warm up latency as well as reduce the intermediate storage requirements. Therefore, the default fuse loop level is set to the innermost loop.

For the brighten and blur example we have been using, the original loop nest without fusion optimization, shown in Figure 4.17a, will start the blur kernel after the 64×64 block is brightened, requiring a brighten buffer size of 4096 entries, and a latency of 4096 to obtain the first blurred pixel out of this image pipeline. By setting fusion at the y loop nest, shown in the Figure 4.17b, the polyhedral fusion analysis will set $delay = 1$ and $II_r = 1$ for the y loop in blur kernel. Therefore, the blur kernel can be initiated right after the second row of the brighten kernel is processed, reducing the latency to only 128 cycles. Similarly, only an intermediate buffer size of 128 is required. Furthermore, if the loop is fused at the innermost level x, the polyhedral analysis will set the $delay = 1$ and $II_r = 1$ for the x loop, resulting in the fused loop shown in Figure 4.17c. This loop allows the blur kernel start executed as soon as the computation of blur kernel on the second pixel of the second row, which further reduces the latency to 65 cycles and require only a buffer with 1 row of size 64 and one pixel. The updated schedule will assigned to the corresponding port in unified buffer, as depicted in Figure 4.18.

Regarding the initiation interval (II) and delay for outer loops above the fusion level, we will follow a standard High-level Synthesis (HLS) loop scheduler [93]. This HLS scheduler implements a simple rule that sets the II equal to the total latency of one inner loop iteration, thereby propagating the II to the outer level loop. This approach ensures that the outer loop executes with the same cadence of the fused inner loop, and avoids any unnecessary pipeline stalls or bubbles if the outer loop's II was set too high. By adhering to this standard HLS loop scheduler, we can optimize the performance of the fused loop nest and ensure efficient execution across all loop levels.

4.4.1 Loop Alignment

In the previous section, we mentioned that loop fusion algorithm can set the relative II and delay and bring the consumer to producer as close as possible. However, loop fusion will not be legal when it changes the execution order of dependent operations[17]. Moreover, loop fusion also can greatly increase the complexity of the addressing logic and schedule leading to increased hardware complexity and degraded performance. For the resulting fused loop to yield a more efficient hardware solution the producer and consumer should access the data in the same order and the loops to be fused should have the compatible iteration space. Thus, the decision to apply loop fusion or not strongly depends on the dependency between producer and consumer operations. This dependency determines the level at which the producer loop should fuse with the consumer loop. We use the following algorithm shown in Algorithm 2 to determine whether to fuse loops:

```
//Outer tiling loop
for t in [0:16]
  //Brighten
  for y in [0:64]:
  for x in [0:64]:
    brighten(x, y) = 2*input(x,y)

  //Blur
  for y in [0:63]:
  for x in [0:63]:
    blur(x, y) = [brighten(x, y)
                + brighten(x+1, y)
                + brighten(x  , y+1)
                + brighten(x+1, y+1) ]/4
```

(a) Loop nest for brighten blur pipeline

```
//Outer tiling loop after fusion
for t in [0:16]:
for y in [0:64]:
  //Brighten
  for x in [0:64]:
    brighten(x, y) = input(x,y)

  //Blur
  if (y>=1)
    for x in [0:63]:
      blur(x, y-1) = [brighten(x, y-1)
                    + brighten(x+1  , y-1)
                    + brighten(x, y)
                    + brighten(x+1  , y) ]/4
```

(b) Loop nest for brighten blur pipeline after loop fusion at the y loop nest.

```
//Outer tiling loop after fusion
for t in [0:16]:
for y in [0:64]:
for x in [0:64]:

  //Brighten
  brighten(x, y) = input(x,y)

  //Blur
  if (y >=1 && x>=1)
    blur(x-1, y-1) = [brighten(x-1, y-1)
                    + brighten(x  , y-1)
                    + brighten(x-1, y)
                    + brighten(x  , y) ]/4
```

(c) Loop nest for brighten blur pipeline after loop fusion at the x loop nest.

Figure 4.17: Loop nest before and after loop fusion optimization. The interleaving of brighten kernel and blur kernel changes. We assume buffers for `input`, `brighten` and `blur` are allocated at root level and data could be reused between iterations.

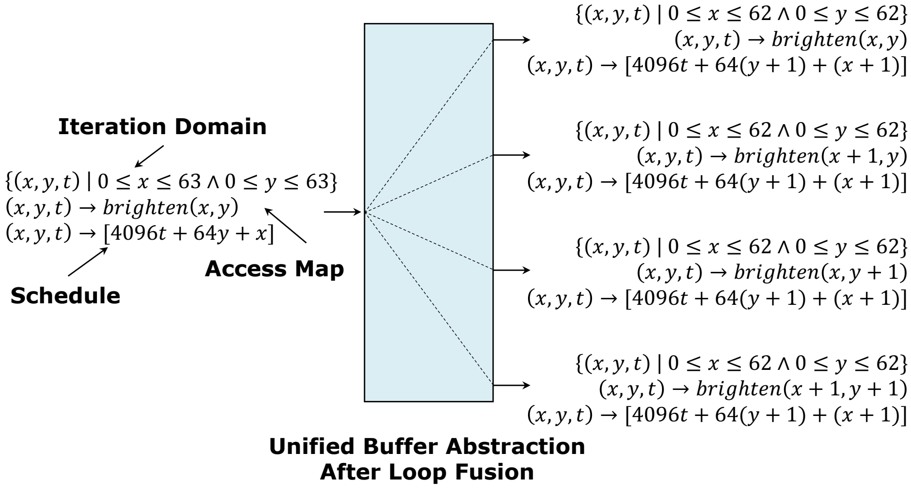

Iteration Domain
$\{(x,y,t) \mid 0 \le x \le 63 \wedge 0 \le y \le 63\}$
$(x,y,t) \to brighten(x,y)$
$(x,y,t) \to [4096t + 64y + x]$
Access Map
Schedule
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x,y)$
$(x,y,t) \to [4096t + 64(y+1) + (x+1)]$
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x+1,y)$
$(x,y,t) \to [4096t + 64(y+1) + (x+1)]$
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x,y+1)$
$(x,y,t) \to [4096t + 64(y+1) + (x+1)]$
$\{(x,y,t) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x,y,t) \to brighten(x+1,y+1)$
$(x,y,t) \to [4096t + 64(y+1) + (x+1)]$
Unified Buffer Abstraction
After Loop Fusion

Figure 4.18: The unified buffer *brighten* after loop fusion.

Algorithm 2: Loop Alignment Algorithm

```
$raw\_deps$ = ExtractReadAfterWriteDependencyMap();
$loop\_map = \{\}$;
for $(prod, cons) \in raw\_deps$ do
    for $lp_p \in prod$.loopnest() do
        $lp_c \leftarrow$ FindBestMatchingConsumerDimension($lp_p, cons$);
        $loop\_map[lp_p] = lp_c$;
        if InjectiveMapInOrder($loop\_map$) then
            PadLoopLevel($prod$, $cons$);
            LoopFusion($prod$, $cons$);
        end
    end
end
```

1. Extract the read-after-write dependency map from the producer iteration domain to consumers for all relevant pairs of loops in the application's loop nest.

2. For each pair of producer and consumer, iterate through all the producer iteration dimensions and identify the best matching consumer iteration dimension, based on the tripcount and stride in access pattern.

3. Create an injective map that associates the producer's iteration level with the corresponding consumer's iteration level. If the mapping's range preserves the same loop order as its domain,

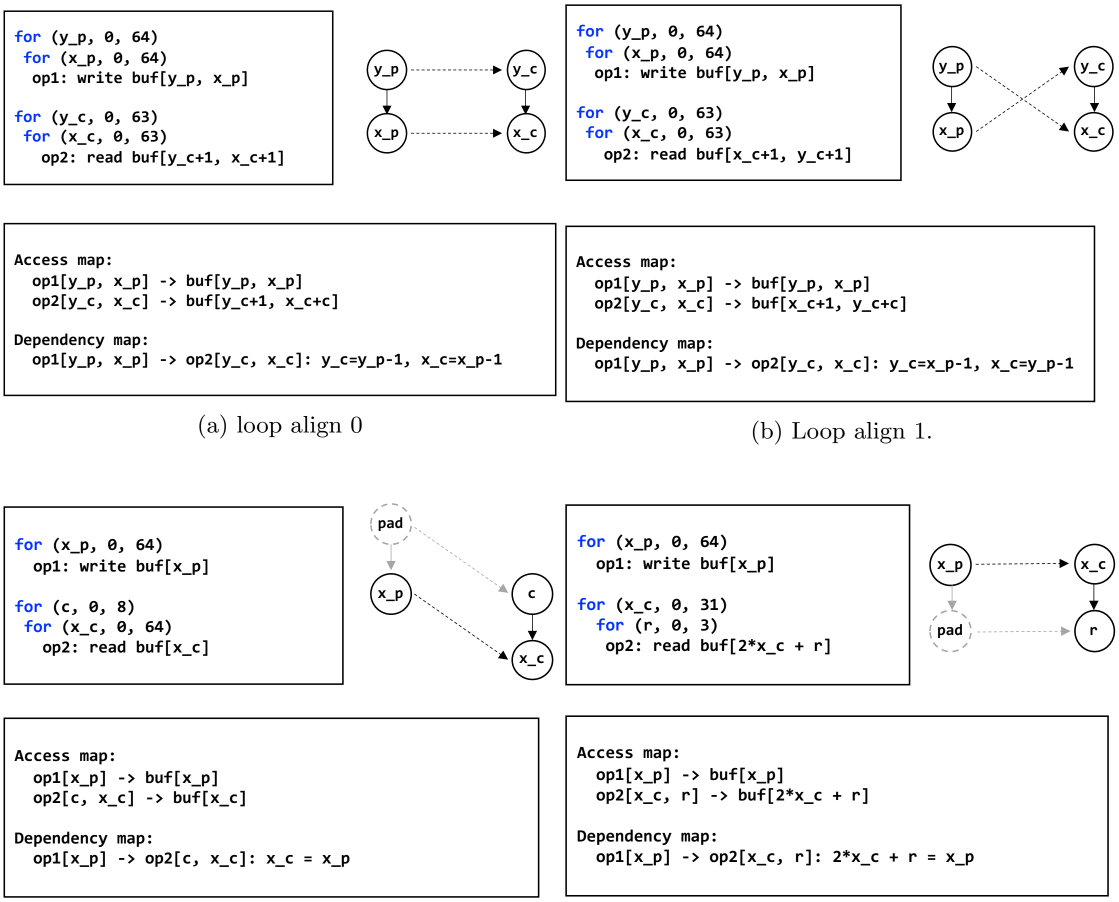
indicating alignment between the iterations of these producer-consumer pairs, And it means that these two loop can be fused.

4. If one loop level does not match any level of the other loop nest, pad the corresponding loop level in the other loop nest with a single iteration loop so that the two loop nests have the same depth and can be fused by loop fusion algorithm introduced in [38].

Figure 4.19 shows several examples to illustrate how the loop alignment finds corresponding consumer and producer loop iterators. The round nodes represent loop node in the loop nest abstract syntax tree, while the dash arrows indicate the corresponding relation between consumer producer loop nodes. Figure 4.19a shows the simplest example. Both the producer and consumer operations access the data in unified buffer buf in the same row major order. The operation dependency map has two constraints $y_c = y_p - 1$ and $x_c = x_p - 1$, which identifies the corresponding consumer iterator for producer iterator y_p and x_p are y_c and x_c, respectively. If we alter the data access pattern in consumer loop from row major order to column major order, which is shown in Figure 4.19b, the data dependency map constraints will change into $y_c = x_p - 1$ and $x_c = y_p - 1$, accordingly. This constraints indicates that iterator y_c consumes data along the direction where iterator x_p produced. Therefore the loop alignment result is flipped. Inner consumer loop x_c matches with the outer producer loop y_p and loop y_c matches with producer loop x_p. Noticed that the range of the map between producer loop iterators and consumer loop iterators is reordered from their original loop order. This loop nest is difficult to fuse and our algorithm won't fuse these loops.

Conventionally, loop fusion algorithms require loops to be fused with the same dimension in their iteration domains. However, in practical scenarios, consumer loops often possess a higher dimensionality compared to producer loops. This is especially common in image processing and machine learning applications where data reuse plays a significant role. Our loop alignment algorithm is capable of accommodating loops with different iteration space dimensions. To make the iteration space compatible, a dummy iteration dimension will be padded. Figure 4.19c depicted a loop nest that reuse data in buf for multiple output channels in dimension c. According to the dependency map, the channel dimension c is irrelevant to the producer loop iterator x_p. We pad the producer loop with a loop with single iteration on top of the loop x_p so that it can pair with the consumer loop iterator c in the fused loop nest.

In some access patterns, there will be multiple loop iterators in the consumer loop to access the same dimension that the producer iterator writes to. For instance, Figure 4.19d illustrates a sliding window access pattern with stride two on the 1D data sequence. The dependency map constraint $2 \times x_c + r = x_p$ specifies that both the iterator x_c and r are dependent on the producer loop iterator x_p. In order to determine which consumer loop level x_p should be fused with, we utilized a straightforward rule that the consumer loop having the most similar data space range to the producer loop will be selected for the fusion. In Figure 4.19d, the data space range of x_c loop is $2 \times 31 = 62$, whereas the range of loop r is only 3. Thus, we pair the producer loop, x_p, with the



```
for (y_p, 0, 64)
 for (x_p, 0, 64)
  op1: write buf[y_p, x_p]

for (y_c, 0, 63)
 for (x_c, 0, 63)
   op2: read buf[y_c+1, x_c+1]
```

```
Access map:
  op1[y_p, x_p] -> buf[y_p, x_p]
  op2[y_c, x_c] -> buf[y_c+1, x_c+c]

Dependency map:
  op1[y_p, x_p] -> op2[y_c, x_c]: y_c=y_p-1, x_c=x_p-1
```

(a) loop align 0

```
for (y_p, 0, 64)
 for (x_p, 0, 64)
  op1: write buf[y_p, x_p]

for (y_c, 0, 63)
 for (x_c, 0, 63)
   op2: read buf[x_c+1, y_c+1]
```

```
Access map:
  op1[y_p, x_p] -> buf[y_p, x_p]
  op2[y_c, x_c] -> buf[x_c+1, y_c+c]

Dependency map:
  op1[y_p, x_p] -> op2[y_c, x_c]: y_c=x_p-1, x_c=y_p-1
```

(b) Loop align 1.

```
for (x_p, 0, 64)
  op1: write buf[x_p]

for (c, 0, 8)
 for (x_c, 0, 64)
   op2: read buf[x_c]
```

```
Access map:
  op1[x_p] -> buf[x_p]
  op2[c, x_c] -> buf[x_c]

Dependency map:
  op1[x_p] -> op2[c, x_c]: x_c = x_p
```

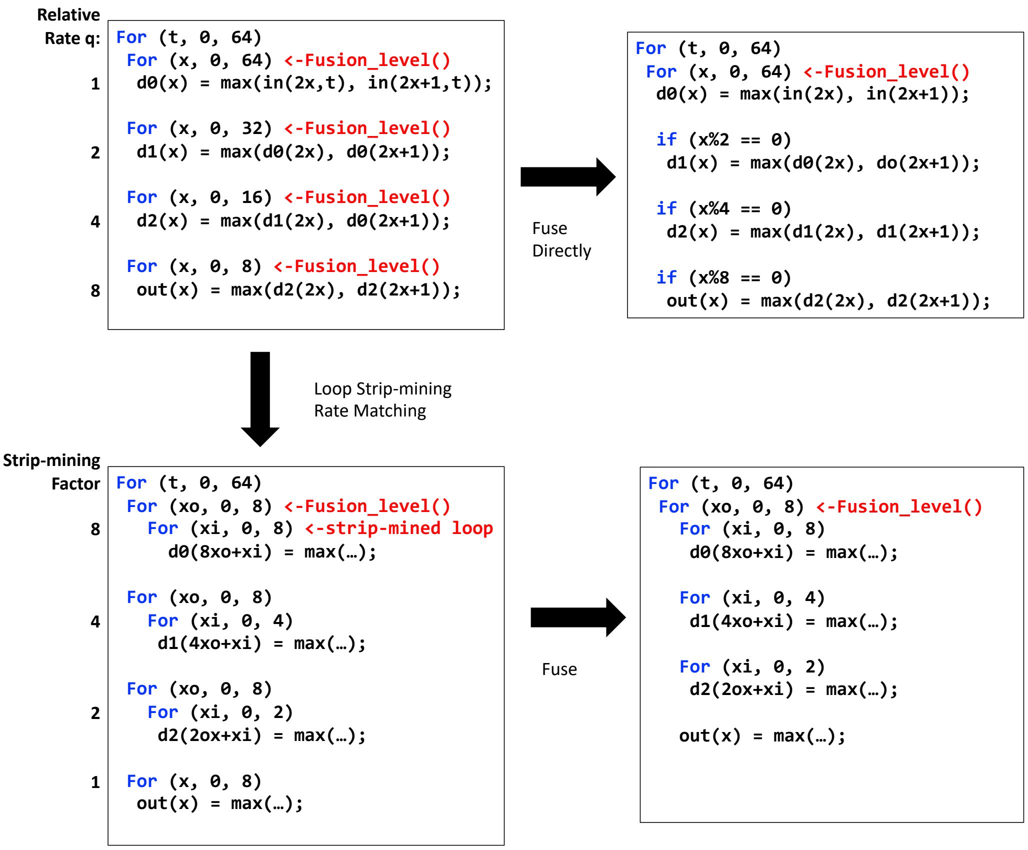
(c) Loop align 2.

```
for (x_p, 0, 64)
  op1: write buf[x_p]

for (x_c, 0, 31)
  for (r, 0, 3)
   op2: read buf[2*x_c + r]
```

```
Access map:
  op1[x_p] -> buf[x_p]
  op2[x_c, r] -> buf[2*x_c + r]

Dependency map:
  op1[x_p] -> op2[x_c, r]: 2*x_c + r = x_p
```

(d) Loop align 3.

Figure 4.19: Loop alignment examples.



```
For (t, 0, 64)
 For (x, 0, 64) <-Fusion_level()
  d0(x) = max(in(2x,t), in(2x+1,t));

 For (x, 0, 32) <-Fusion_level()
  d1(x) = max(d0(2x), d0(2x+1));

 For (x, 0, 16) <-Fusion_level()
  d2(x) = max(d1(2x), d0(2x+1));

 For (x, 0, 8) <-Fusion_level()
  out(x) = max(d2(2x), d2(2x+1));
```

```
For (t, 0, 64)
 For (x, 0, 64) <-Fusion_level()
  d0(x) = max(in(2x), in(2x+1));

  if (x%2 == 0)
   d1(x) = max(d0(2x), do(2x+1));

  if (x%4 == 0)
   d2(x) = max(d1(2x), d1(2x+1));

  if (x%8 == 0)
   out(x) = max(d2(2x), d2(2x+1));
```

```
For (t, 0, 64)
 For (xo, 0, 8) <-Fusion_level()
   For (xi, 0, 8) <-strip-mined loop
     d0(8xo+xi) = max(…);

 For (xo, 0, 8)
   For (xi, 0, 4)
    d1(4xo+xi) = max(…);

 For (xo, 0, 8)
   For (xi, 0, 2)
    d2(2ox+xi) = max(…);

 For (x, 0, 8)
  out(x) = max(…);
```

```
For (t, 0, 64)
 For (xo, 0, 8) <-Fusion_level()
   For (xi, 0, 8)
    d0(8xo+xi) = max(…);

   For (xi, 0, 4)
    d1(4xo+xi) = max(…);

   For (xi, 0, 2)
    d2(2ox+xi) = max(…);

   out(x) = max(…);
```

Figure 4.20: During loop fusion, the scheduling algorithm will check the relative rate of each subkernel to be fused, if they share the same hardware resource, it will apply an automatic loop strip-mining algorithm to match the data processing rate in every coarse grain iteration, removing the pipeline stall and improving the computation utilization.

consumer loop, x_c, and insert an additional loop iterator into the producer loop's innermost level to align it with the window dimension r for fusion.

4.4.2 Optimization: Loop Stripmining to Increase Compute Utilization

After fusion is done, the operation subloops will be placed under the target fused level. It worth noting that not all the compute units will be executed in every outer loop, due to the data dependency and different data consuming rate among various pipeline stages. It's common that kernels will be guarded by an *if* condition after fusion as is shown in Figure 4.17. That will result in a low compute utilization ratio. For instance, consider an application depicted in Figure 4.20 which contains three consecutive stride-two max pooling kernels along a 1D data sequence. If the loops are fused at the innermost level, the resulting pipeline will cause the later stages in the pipeline to fire less frequently.

While this schedule is acceptable if each kernel is associated with a dedicated compute hardware, as the pipeline throughput is bounded by the hardware used most frequently, the first max pooling kernel, a more efficient hardware mapping strategy is to share the hardware between all compute kernel. This approach can significantly reduce the hardware resource consumption as discussed in Section 4.3.3. However, if the operations share the hardware resource, this fused schedule will cause the hardware unit to stall, since the second, third and last stage will execute only once for every two four, and eight time the first down sample kernel proceed, respectively.

However the loop fusion algorithm can analyze data dependency and determine the relative initiation interval of each pipeline stage. We leverage this relative II information to strip-mine the fusing loop and ensure that the data processing rate matches in every coarse-grained iteration. This optimization can eliminate the pipeline stalls and enhance the hardware utilization when multiple kernels share the same hardware resource. Specifically, the loop fusion process will produce a initiation interval II_i for each loop i under the fusion level. This information indicates that the consumer loop c will increment

$$\frac{II_p}{II_c} \tag{4.15}$$

steps for every iteration of producer loop p. To achieve a balanced data processing rate across all the pipeline stages, we can increase workload for a loop l by factor of

$$\frac{\forall_{i \in subloop} LCM(II_i)}{II_l} \tag{4.16}$$

where $\forall_{i \in subloop} LCM(II_i)$ represents the least common multiple of all the kernel's relative rate. This factor is the bound of the inner level of the strip-mining. For example, in the cases shown in Figure 4.20, where the relative rate of different down sample kernels are 1, 2, 4, 8. We can strip-mine the fusion level by 8, 4, 2, 1 respectively. Finally, we obtain a pipelined loop without stalls after fusion, which is demonstrated in the bottom right corner of the Figure 4.20.

4.5 Summary

The unified buffer abstraction decoupled the frontend of the compiler, responsible for analyzing data movement, dependency, and optimizing schedules, from the backend, which is specific to hardware implementation. With this clean separation, the compiler only needs a smaller number of parameters, which represent the latency of the compute and memory units, to schedule the compute operations. This separation also streamlines the compilation process and facilitates the optimization of cycle-accurate schedules. These schedules, derived from high-level scheduling primitives and compact physical characteristics of the memory implementation, provide valuable insights and accurate latency estimations. The next chapter dives into the buffer mapping stage, where we focus on generating valid hardware mappings that align with the schedules derived during unified buffer extraction.

It will also explore how the unified buffer abstraction enables energy and area optimizations of the underlying hardware.

Chapter 5

Buffer Mapping

With scheduling finished, all operations have been assigned to non-stalled clock cycles and the bandwidth of each memory is known. The next task of the compiler is to map the abstract unified buffers into implementations built out of the available physical primitives. This mapping produces the configuration bitstream for each physical unified buffer used in the design. In principle, the unified buffers can be mapped directly to physical unified buffers on the target accelerator. In practice, however, this is rarely possible for the following reasons:

- **Limited buffer bandwidth.** The physical unified buffers on the accelerator may not have sufficient bandwidth. For example, the target CGRA [8] only has a single four-word-wide SRAM in each physical unified buffer, as described in Section 6.1.3, meaning that each buffer can only support up to four memory operations per cycle. However, the unified buffer from our *brighten blur* example needs to perform five memory operations per cycle, and many access pattern in common image processing applications need even larger bandwidth.

- **High capacity.** The cycle-accurate scheduler reduces storage requirements by bringing the consumer closer to the producer, but unified buffers may still need more space than what is available in any one physical unified buffer.

- **Wide fetch width.** The accesses in the loop nest may have a narrower bitwidth compare to the available bitwidth in the physical unified buffers. This necessitates additional transformations to convert individual words into wider ones for storage in the memory. Similarly, data serialization is required for loading data from this memory.

This chapter describes the method we use to bridge the gap between software defined unified buffer and its implementation in physical hardware. The next section explores the optimizations employed by our compiler to reduce the necessary memory bandwidth. If the resulting bandwidth still exceeds the capability of the physical buffers, Section 5.2 describes how we utilize multiple memories to create implementation with more physical ports, a technique known as banking. To address memory capacity issues, Section 5.3 introduces how we chain multiple physical banks to create a memory buffer with higher capacity. Next, Section 5.5 shows how we approach the configuration of wide-word memories, to convert their multi-word access into more effective ports, by framing it as a vectorization problem. To address the physical limitations of the underlying hardware, the compiler takes input from a file that specifies constraints for buffer mapping. This file includes details such as input and output port numbers, crucial for banking. Additionally, the specified capacity is employed to guide the memory tile chaining. Furthermore, the file incorporates sub-block information related to memory construction to guide the vectorization process. After introducing all the lowering processes in buffer mapping, we introduce the compiler output after the buffer mapping stage, the unified buffer implementation. This output serves as the input for the subsequent code generation phase, where hardware configurations corresponding to this implementation will be generated.

5.1 Port Reduction Optimization

Our compiler uses two strategies for servicing high bandwidth accesses: **port reduction optimization** and **banking**. Port reduction optimization reduces the required memory bandwidth, while banking partitions the unified buffer into separate physical hardware resources to provide enough bandwidth to meet the targeted throughput.

The port reduction optimization aims to identify opportunities for reusing data within the memory access pattern. When data fetched by one port is subsequently required by another, we employ temporary storage, a delay buffer, and a forwarding mechanism to eliminate the costly memory refetch by the other port. The following section outlines how we identify these reuse patterns using information obtained from the unified buffer extraction through dependence distance analysis. Ports which have a fixed dependency distance relationship can be replaced by delay buffers, and depending on dependence distance, these buffers are implemented as shift registers, or as a memory. After performing the port reduction optimization pass, an unified buffer is generated with the elimination of the shared output port and the implementation of delay buffers between pairs of input and output ports(in-to-out), or between pairs of output ports(out-to-out).

5.1.1 Dependence Distance Analysis

In compiler terminology, dependence distance is the distance between two program's elements that have a dependency relationship, measured by the number of other code elements between them. To determine if some of the unified buffer ports can be simplified, it's necessary to determine the dependence distance between each read and write port. To measure the distance, we need time difference between the read and write operations of the same address in the data space of the unified

Port ID	0	1	2	3	4
Operation	Brighten	Blur			
Wr/Rd	Wr	Rd			
Schedule	(x,y)→[x+64y]	(x,y)→[65+x+64y]			
Access Map	(x,y)→[x,y]	(x,y)→[x,y]	(x,y)→[x+1,y]	(x,y)→[x,y+1]	(x,y)→[x+1,y+1]
t2addr	[x+64y]→[x,y]	[65+x+64y] →[x,y]	[65+x+64y] → [x+1, y]	[65+x+64y] → [x, y+1]	[65+x +64y] → [x+1, y+1]

Table 5.1: Time to address map for all ports in the unified buffer shown in Figure 3.2. The write operation schedule expression is $t_{write} = x + 64y$ and for the read operation, the expression is $t_{read} = 65 + x + 64y$. Since the read and write operation access the same data address space. In order to derive the dependency distance between read write we use address x and y to substitute t in the expression.

buffer.

Fortunately, the unified buffer abstraction provides all the information needed for this analysis. Specifically, the schedule maps the iteration domain of the operations to the executed timestamp, while the access map projects the operations from the same domain to the corresponding address in the data space. By applying the access map to the range of the inverse of schedule map, we can establish a relationship between cycle-accurate time and the address accessed by every port. The integer set transformation can be represent by following formulation:

$$t_{write}2t_{read}[p_i, p_o] = t2addr[p_i].\texttt{Apply}(\texttt{Inv}(t2addr[p_o])) \tag{5.1}$$

Let $access\_map[p]$ and $sched[p]$ be the access map and schedule of port p respectively, Then, we can calculate the map from cycle-accurate time to the address of that port $t2addr[p]$ by using the following transformation.

$$t2addr[p] = \texttt{Inv}(sched[p]).\texttt{Apply}(access\_map[p]) \tag{5.2}$$

Where, `Inv(m)` is the function to obtain an inverse of a map `m`, and `m1.Apply(m2)` represents the transformation that apply map `m2` on the range of map `m1`. Table 5.1 demonstrate the *t2addr* map for all ports in the unified buffer abstraction in Figure 3.2.

Then, for each pair of input and output ports in the unified buffer, we apply the time to address

Port Pair / [Wr, Rd]	$t_{write}2t_{read}$	t_{diff}
[0, 1]	[x+64y]→[65+x+64y]	{ 65}
[0, 2]	[x+64y]→[65+(x-1)+64y]	{ 64}
[0, 3]	[x+64y]→[65+x+64(y-1)]	{ 1 }
[0, 4]	[x+64y]→[65+(x-1)+64(y-1)]	{ 0 }

Table 5.2: The map from time-to-read to time-to-write, which calculated from Table 5.1. The set of dependence distance is calculated by substracting the RHS of the map from the LHS of the map. As we can see from the table all read ports in unified buffer Figure 3.2 has constant dependence distance from its write port.

map of the read port p_o, to the inverse map of the write port p_i to obtain a map between the time-to-read and the time-to-write the similar address, as is shown in Equation 5.1.

By applying the `Delta(m)` operator[1] , we can get the time difference between the same data accessed by pair of read write ports.

$$t_{diff}[p_i, p_o] = \texttt{Delta}(t_{write}2t_{read}[p_i., p_o]) \tag{5.3}$$

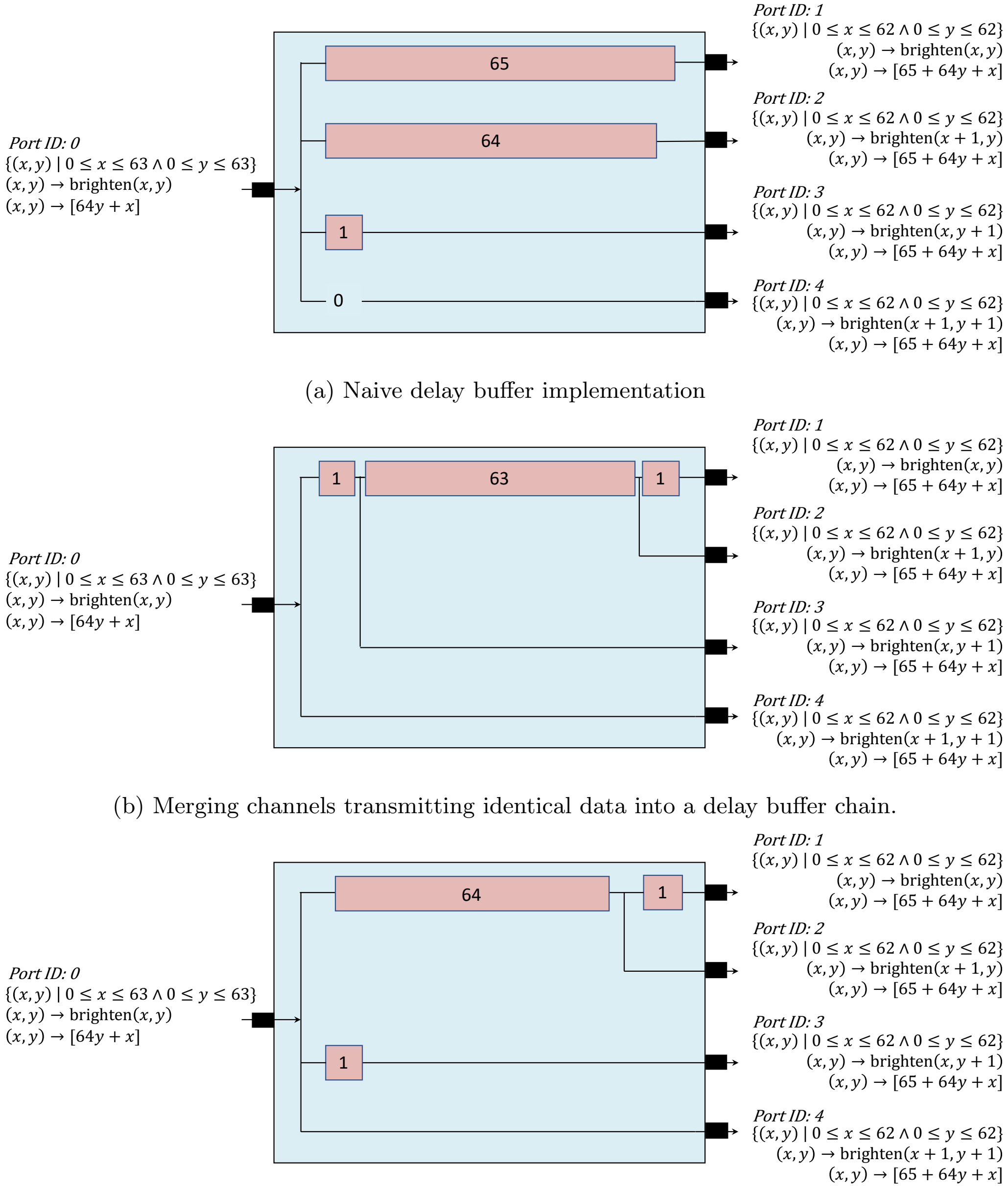
If the time difference set contains only a constant integer, it means every data will be read only once by that read port and the time difference between each data write and read will be a constant value. Therefore, the buffer between this read and write port can be optimized as a channel forwarding data with constant delay. This memory is conceptually a shift register.

After the dependence distance analysis, our compiler creates a data structure which saves all pairs of input output ports that can be optimized into delay memory with their constant dependence distance. As for illustration, the dependence distance map for the unified buffer listed in Figure 3.2 is shown in Table 5.2. This data structure will be used in the next step to create the delay memory implementation.

5.1.2 Create Shift Register Implementation

Now that dependency analysis has identified all the ports in a system that access data with a constant time delay, we can simply create a buffer for each pair of input and output ports that have a constant dependence distance, as identified in the analysis in Section 5.1.1. The delay of each buffer can be set to the corresponding time difference recorded in the data structure in Table 5.2. However, this simple approach, may result in significant waste of hardware resource due to duplication of

[1] This function returns a set containing the differences between range elements (right value of the map) and corresponding domain elements(left value of the map) in the input.


Port ID: 0
$\{(x, y) \mid 0 \le x \le 63 \wedge 0 \le y \le 63\}$
$(x, y) \rightarrow \text{brighten}(x, y)$
$(x, y) \rightarrow [64y + x]$
65
64
1
0
Port ID: 1
$\{(x, y) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x, y) \rightarrow \text{brighten}(x, y)$
$(x, y) \rightarrow [65 + 64y + x]$
Port ID: 2
$\{(x, y) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x, y) \rightarrow \text{brighten}(x + 1, y)$
$(x, y) \rightarrow [65 + 64y + x]$
Port ID: 3
$\{(x, y) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x, y) \rightarrow \text{brighten}(x, y + 1)$
$(x, y) \rightarrow [65 + 64y + x]$
Port ID: 4
$\{(x, y) \mid 0 \le x \le 62 \wedge 0 \le y \le 62\}$
$(x, y) \rightarrow \text{brighten}(x + 1, y + 1)$
$(x, y) \rightarrow [65 + 64y + x]$
1
63
1
64
1
1

(a) Naive delay buffer implementation

(b) Merging channels transmitting identical data into a delay buffer chain.

(c) Group the read port by the dependence distance, which creating a tree structure.

Figure 5.1: Different Shift Register Optimization Implementations

data across multiple channels, as shown in Figure 5.1a. To avoid this issue, channels transmitting identical data can be merged into a delay buffer chain with read ports connected to the middle of the chain. This allows data to be read with a specified delay without duplicating it or creating hardware redundancy. The buffer implementation after delay buffer merging is shown in Figure 5.1b, where all four channels are merged into one chain that serves the read ports from this unified buffer.

In order to implement the delay buffer chain on a coarse-grained reconfigurable array(CGRA), the set of delay buffers in the chain will be mapped onto two distinct distinct memory resources available on the CGRA routing network, **memory tiles** and **shift registers**. To facilitate this mapping process, a straightforward rule is utilized to determine the appropriate memory resource to be assigned to each delay buffer in the chain. Specifically, the decision is based on the cycle delay between consecutive delay buffers. When the delay exceeds a predetermined threshold, the corresponding delay buffer is assigned to a memory tile. Conversely, if the delay falls below this threshold, the delay buffer is assigned to a chain of shift registers. In practice, the threshold is set to 20.[2] For the example shown in Figure 5.1b, the buffer with delay equal to one will be map to a single shift register, and the delay buffer with delay of 63 cycle will be mapped to a memory tile.

5.1.3 CGRA Routing Optimization

Our compiler performs another backend-specific optimization to accommodate the physical layout of resource on the CGRA. As the shift registers are located within the routing network and memories are positioned in middle of them, implementing a chain of memories and shift registers can result in some long routing paths, such as the connection wired with port 3 and port 4 in Figure 5.1b. To alleviate routing congestion, it's necessary to improve the locality of resources for realizing the shift register. Instead of creating a delay buffer chain, a more efficient implementation for reconfigurable architecture, such as a CGRA, involves a tree structure as is depicted in Figure 5.1c. Paths with significant variance in dependence distance can diverge early in the tree style implementation. To create this tree structure, a compiler pass will separate the input-output port pair into different groups based on the variance of the dependence distance. The compiler pass first sorts all port pairs by their dependency distance and then assigns them to previously created groups or creates new ones as necessary. If the variance in dependence distance between a given port pair and the previously grouped port pairs is greater than the threshold used to differentiate between shift register chains and memory tiles, then a new group is created to ensure the segregation of ports with memory connections between them into distinct groups.

In the example shown in Figure 5.1c, the sorted dependency distance are $\{port1 : 0, port2 : 1, port3 : 64, port4 : 65\}$ respectively. According to the grouping rule, they will be separated into two groups $\{port1, port2\}, \{port3, port4\}$. Finally each group will be merged as a chain of memory

[2] This threshold is set arbitrarily. For our examples the shift registers are delays of small integers, while the memories generally are much larger. 20 fits nicely in this gap

and shift register independently as we described in Section 5.1.2.

5.1.4 Output Port Sharing

While the preceding optimizations have focused on enhancing buffer implementation between input and output ports with the same access pattern, output port sharing is another optimization technique that aims to optimize data reuse between different output ports, regardless of how the input port behaves. Output port sharing involves sharing the output ports of a buffer between multiple consumers, such as computation units or data transfer to another memory hierarchy level. This technique is useful in scenarios where the same data needs to be processed by multiple engines, such as in deep learning multi-channel convolution or matrix multiply. Therefore, this technique reduces the number of ports required for read operations and results in improved energy efficiency by reducing the number of memory fetch operations required.

To determine the output ports which can be shared, we perform the same dependence distance analysis mentioned in Section 5.1.1 between all pairs of output ports that cannot be optimized as shift register outputs. If these pairs of output ports have constant dependence distance, we utilize shift registers to transfer data from one memory fetch to all consumer ports that use it. Additionally, we incorporate this information into the out-to-out delay data structure after implementing port sharing optimization.

5.2 Memory Banking

After shift register optimization, the remaining ports must be serviced from a physical memory with a limited number of read and write ports. The effective number of ports can be increased by dividing the memory into multiple physical partitions so the system can access multiple memory locations simultaneously, a technique called banking. This increased memory bandwidth can fulfill the throughput requirement in the schedule we created in Chapter 4. In a banked memory system, each bank is accessed through a separate memory channel or port. The number of banks in the system depends on the memory architecture and the pattern of simultaneous memory access. Each bank has its own address range, and the system uses the bank number to determine which bank to access. Our compiler uses a simplified version of an optimal banking algorithm for stencil computations [21]to find legal banking schemes for the remaining ports. The basic idea is described next.

5.2.1 Banking Methodology

Cyclic banking is one of the technique used in memory optimization to maximize memory bandwidth by reducing memory bank conflicts. It involves dividing the memory into multiple banks and configuring them to allow accesses to occur in parallel. Consider the scenario of executing a vector

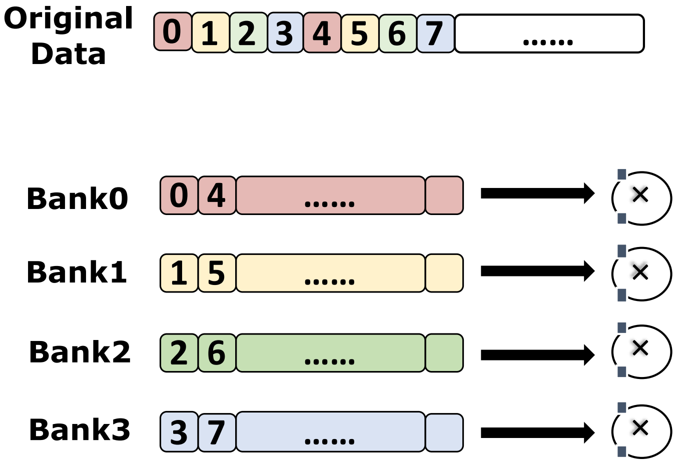


Figure 5.2: Cyclic banking partitions data in four different banks and sustains the bandwidth requirement for four parallel compute engines.

and scalar multiplication as an example. The schedule specifies the simultaneous computation of four multiplications in parallel. To meet the throughput requirement and supply data to the parallel compute unit efficiently, the memory must support parallel access for four consecutive data elements. As a result, the memory system adopts a cyclic partition strategy to store successive data elements in alternating banks. This strategy effectively reduces potential access bottlenecks, as illustrated in Figure 5.2, where each of the four distinct colors represents a different memory bank. Next we will introduce the algorithm that deduce this partition scheme from the information extracted in unified buffer.

Generally speaking, a memory banking procedure will create a transformation $f(\vec{x})$ from original address vector $\vec{x}$ in the data space to the bank ID as well as another function $g(\vec{x})$ which transforms the original address to offset inside each bank. The bank ID determines which partition each memory operation will fetch from, and the inner bank offset will determine the address within each partition. The cyclic bank mapping function is a key component in memory banking optimization. This function can be represented mathematically as

$$f(\vec{x}) = (\vec{\alpha} \cdot \vec{x}) \mod N$$

where α is a mapping vector and N is the number of banks. From geometry perspective, $\vec{\alpha} \cdot \vec{x}$ represents a sequence of hyperplanes in the high dimensional data space. And the function $f(x)$ assign the data points on hyperplane to different memory banks. The goal of this function is to avoid access conflicts between any two parallel memory references, which is achieved by verifying that $f(\vec{x_p}) \neq f(\vec{x_q})$ for every pair of parallel accesses $\vec{x_p}$ and $\vec{x_q}$. To accomplish this, a polyhedral analysis technique known as the polytope emptiness test is used. In order to find the optimal banking, prior work [84] have used an integer linear programming solver to determine the mapping vector $\vec{\alpha}$, ensuring that there are no bank access conflicts while minimizing the number of banks required, N. This process usually use exhaustive enumerating the possible candidate and could be time-consuming[85].

To optimize memory banking, a more recent study [22] proposed a graph-based approach that extracts the optimal banking strategy by coloring an graph data structure. This algorithm primarily focused on data partitioning for overlapping stencils with data reuse across different memory ports. However, the port reduction optimization already captures data reuse through the introduction of shift registers. Consequently, the graph-based approach becomes redundant. Hence we created a simplified alternative to derive the banking function $f(x)$ which is heuristic-based and has proved to be highly effective in optimizing memory banking.

The heuristic based method computes a vector of cyclic partition factors $(f_0, f_1, ...f_n)$ for n dimensional data array. This vector indicates that the corresponding dimension i of the data array will be cyclically partitioned into f_i banks. The high dimensional bank ID will be calculated by the function:

$$f(\vec{x}) = (x_0 \mod f_0\ , x_1 \mod f_1\ , ..., \ x_n \mod f_n) \tag{5.4}$$

which is an elementwise mod operation. The inner bank offset function will be an elementwise floor division by the cyclic partition factors,

$$g(\vec{x}) = (\lfloor x_0/f_0 \rfloor, \lfloor x_1/f_1 \rfloor, ..., \lfloor x_n/f_n \rfloor) \tag{5.5}$$

Our problem now is to efficiently determine the cyclic partition factors. Our heuristic first groups the ports that have overlapping schedules, meaning that they access the buffer at the same time. For every group of ports with high bandwidth requirements, we initially fully partition the address space into different banks for dimensions where all ports have constant indexing, by setting the cyclic partition factor as the dimension size. This step captures the case where the ports access different data sets. Then, we iterate over the other dimensions in the data space and compute the greatest common divisor (GCD) of non-zero strides in the affine expression of the access map between all ports in the group. We set the cyclic partition factor for that dimension to the computed GCD. This step captures the type of partitioning possible when each bank is supplying a different element in a vector.

For example Figure 5.3 depicts a loop nest in the convolutional neural network(CNN) layer where the kernel loads the four-dimensional weight from global buffer onto the on-chip memory tile from line 5 to line 10, and the convolution kernel from line 13 to 21 consumes the weight respectively. The computation is completed in parallel along the input channel c dimension, while two computation engines are executed simultaneously to produce two partial result in the output.

```
//Outer tiling loop
for t in [0:16]
  //Load of input and initialization of output is ignored for simplicity

  //load weight from GLB to CGRA Memory
  for ky in [0:3]:
  for kx in [0:3]:
  for k in [0:4]:
  for c in [0:4]:
    weight_mem(c, k, kx, ky) = weight_glb(c, k, kx, ky, t);

  //3x3 Convolution layer with 4 input channels and 4 output channels
  for ky in [0:3]:
  for kx in [0:3]:
  for y in [0:28]:
  for x in [0:28]:
  for k in [0:2]:
    output_mem(2k, x, y) =
        CONV_0(input_mem(0:4,x,y), weight_mem(0:4, 2k, kx, ky), output_mem(0, x, y));
    output_mem(2k+1, x, y) =
        CONV_1(input_mem(0:4,x,y), weight_mem(0:4, 2k+1, kx, ky), output_mem(1, x, y);

  //Drain of the output value is ignored for simplicity
```

Figure 5.3: Loop nest for the a multi channel 3x3 convolution DNN layer. It has 4 input channels and 4 output channels. The computation is fully unrolled (parallel) along the input channel dimension `c`. The output channel dimension `k` is unrolled by two and the 3x3 kernel dimension is done sequentially. The kernel of loading input feature map, initializing and draining of the output feature map is ignored. We will focus on the data access pattern on the weight unified buffer.

This parallel computation engine design results in two streams of four channels of weight being loaded into the compute unit. This leads to 8 read ports in the weight unified buffer to achieve the desired throughput. The access pattern for those eight ports

$$0 : (k, x, y, kx, ky) \rightarrow weight(0, 2k, kx, ky) \tag{5.6}$$
$$1 : (k, x, y, kx, ky) \rightarrow weight(1, 2k, kx, ky) \tag{5.7}$$
$$2 : (k, x, y, kx, ky) \rightarrow weight(2, 2k, kx, ky) \tag{5.8}$$
$$3 : (k, x, y, kx, ky) \rightarrow weight(3, 2k, kx, ky) \tag{5.9}$$
$$4 : (k, x, y, kx, ky) \rightarrow weight(0, 2k + 1, kx, ky) \tag{5.10}$$
$$5 : (k, x, y, kx, ky) \rightarrow weight(1, 2k + 1, kx, ky) \tag{5.11}$$
$$6 : (k, x, y, kx, ky) \rightarrow weight(2, 2k + 1, kx, ky) \tag{5.12}$$
$$7 : (k, x, y, kx, ky) \rightarrow weight(3, 2k + 1, kx, ky) \tag{5.13}$$

and the scheduler will assign the same timestamp for all these eight port according to the schedule algorithm in Chapter 4. When we apply the heuristic-based approach to find the partition factors of the weight unified buffer we find the innermost (first-from-left) dimension across all ports has constant indexing. This allows us to fully partition this dimension choosing a $f_0 = 4$. As for the second dimension, the access expressions are $2k$ and $2k + 1$, where the greatest common divisor of strides from input iterators is 2. Consequently, we set the partition factor for this dimension to 2. Similarly, the partition factor for the remaining dimensions are set to 1, which indicates no partition. Finally, the resulting cyclic partition factor vector of the weight unified buffer is $[4, 2, 1, 1]$. Applying the bank id function $f(\vec{x})$ in Equation 5.4 and inner bank offset function $g(\vec{x})$ in Equation 5.5, we can get the following *bank ID* and *inner bank offset* **tuple maps** for each port.

$$0 : (k, x, y, kx, ky) \rightarrow \{(0, 0), (kx, ky)\} \tag{5.14}$$
$$1 : (k, x, y, kx, ky) \rightarrow \{(1, 0), (kx, ky)\} \tag{5.15}$$
$$2 : (k, x, y, kx, ky) \rightarrow \{(2, 0), (kx, ky)\} \tag{5.16}$$
$$3 : (k, x, y, kx, ky) \rightarrow \{(3, 0), (kx, ky)\} \tag{5.17}$$
$$4 : (k, x, y, kx, ky) \rightarrow \{(0, 1), (kx, ky)\} \tag{5.18}$$
$$5 : (k, x, y, kx, ky) \rightarrow \{(1, 1), (kx, ky)\} \tag{5.19}$$
$$6 : (k, x, y, kx, ky) \rightarrow \{(2, 1), (kx, ky)\} \tag{5.20}$$
$$7 : (k, x, y, kx, ky) \rightarrow \{(3, 1), (kx, ky)\} \tag{5.21}$$

Once the cyclic partition factor has been computed, the compiler proceeds to the next step, which involves verifying if there are any parallel memory accesses that share the same bank ID. In the event that such a partition conflict is detected, our heuristic cyclic banking approach fails, and the compiler falls back to exhaustively banking the memories by fully duplicating the entire buffer between each pair of input and output port that have overlapping data access. Our CGRA backend favors duplication of data over the solver-based method because the latter requires a switching network to send the correct data from the corresponding bank to the compute operand, leading to complex switching network logic.

5.2.2 Bank Implementation

The bank implementation data structure is a crucial element in the buffer mapping process that bridges the gap between the bandwidth-unlimited unified buffer and the bandwidth constrained physical implementation of unified buffers building blocks. The bank implementation data structure is essentially a set of constrained unified buffers that represent the behavior of each bank. The reason why this data structure is constrained is that each bank has a fixed number of input and output ports, and a fixed capacity limitation. Moreover, a partitioned data space is assigned to each bank to help dissect the corresponding access pattern and schedule from the unified buffer port to the port on the physical bank implementation.

Additionally, each bank in the bank implementation data structure has a connection relation with the unified buffer ports, which includes information on shift register delay and output port sharing, as discussed in Section 5.1. This connection relation will later be transformed into a interconnect logic during the code generation phase. These connections are flexible, with one input port being able to write data to multiple banks, and one bank able to receive data from multiple unified buffer input ports. Similarly, one read port can receive data from multiple banks, and one bank can also be read by multiple read ports.

5.2.3 Bank Merging

In order to achieve high bandwidth in parallel computation and improve energy efficiency, the physical implementation of the memory can have more than one input/output port. Specifically, our physical unified buffer, described in Section 6.1.2, has two input and two output ports. To build a general compiler that can target any number of I/O ports on the physical implementation, we implement a bank merging pass.

The bank merging pass optimizes the physical resource utilization by iteratively seeking opportunities for merging banks. Initially, a bank with single read port and single write port is created when creating the bank implementation in Section 5.2.2. Then, the pass analyzes the access patterns and tries to merge multiple banks if they have the same access data space partition. By merging banks, we can reduce the number of physical resources required for implementation.

The bank merging pass uses a collateral file that identifies the number of input and output port for the physical memory tiles that are available in the target hardware. If the physical memory is more capable, meaning that the physical memory implementation has more port than the abstract banks have, the compiler will perform bank merging iteratively until the requested number of ports is utilized.

5.3 Memory Chaining

To map unified buffers with higher capacity than any one physical buffer, buffer mapping chains several buffers into a single logical buffer. Each memory tile on the CGRA is assigned a unique ID. Our compiler statically analyzes the access map of the unified buffer and partitions the access map's range into pieces implemented by multiple chained physical buffers. The following equations transform a logical address a in the access map into a tile ID and a physical address in the memory tile, using the capacity C of the memory tile:

$$\text{ID}(a) = \text{floor}(a/C) \qquad \text{PhysicalAddress}(a) = a \bmod C \tag{5.22}$$

Subsequently, for each chained memory instance bearing a unique tile ID, the partitioning of the iteration domain can be deduced. Leveraging the schedule of the original unified buffer before the optimization of chaining, the compiler generates schedules for each memory partition accordingly.

5.4 Memory Hierarchy

It's common for our target hardware accelerators to have a memory hierarchy to optimize its energy and performance. Our unified buffer abstraction provides the flexibility to represent arbitrarily levels of memory hierarchy with any desired topological interconnect network connectivity. Within this abstraction, memory is conceptualized as an array of data accessed through compute or data move operations. To illustrate, creating a two-level memory hierarchy involves copying from a unified buffer with a large capacity to a smaller unified buffer. On our CGRA, we refer to the large, outer memory as the global buffer, while the smaller ones are called memory tiles. The global buffer uses ready-valid signaling to connect to the processor's memory system, and will stall the execution engine if a block of data has not been loaded before the time it needs to be pushed to the memory tiles.

To further improve energy efficiency, we have introduced an additional level of memory hierarchy, consisting of a register file near each processing element in our next generation CGRA. Our unified buffer supports this improvement by enabling the creation of extra copy operations into an separate data array, facilitating seamless integration of the register file into the memory hierarchy.

From an application lowering perspective, our unified buffer abstraction remains agnostic to the memory hierarchy level until the mapping stage. Utilizing the data accessing and schedule information, the capacity can be static analyzed after the banking and memory partition passes. The compiler can automatically select which hierarchical level the buffer will map to. If the user wants storage to be placed at a specific level in the memory hierarchy, our scheduling language allows them to specify this as well. This memory hierarchy level information is then carried downstream through the compiler lowering passes until the buffer mapping phase. Finally, the compiler will generate the correct configuration for each physical memory primitive based on the memory hierarchy information.

5.5 Vectorization

Single-port memory offers advantages in terms of high area density and low energy consumption per bit-fetch, making it a favorable choice for on-chip memory implementations. As illustrated in Figure 5.4, we can employ a single-port wide-fetch SRAM to simulate a multi-port memory. In this setup, a Serial-In-Parallel-Out (SIPO) buffer is used to aggregate individual data words into a wide word, while a Parallel-In-Serial-Out (PISO) buffer serializes the wide word into data that can be read by functional units. Since multiple data words are written/read during each SRAM access, these SRAM operations don't need to happen every cycle. If the fetch width is N, the wide port utilization decreases to 1/N. Through the interleaving of load and store operations on the single-port SRAM's interface, it becomes possible to emulate the functionality of a multi-port memory. In our target Coarse-Grained Reconfigurable Architecture (CGRA), we leverage this micro-architectural optimization to construct a memory tile with two inputs ports and two output ports using a 4-wide SRAM. In our design the SIPO buffer is referred to as the aggregator (AGG), and the PISO buffer is known as the transpose buffer (TB). Section 6.1.3 describes the design of this memory in more detail.

To support the use of physical buffers with wide-fetch SRAMs, we added a compiler pass, which is similar to vectorization, in the buffer mapping stage. This pass ensures that the access patterns of the buffers can be broken into sub-sequences with the same length as the SRAM fetch width as shown in Figure 5.5. At each input port of the buffer, this sub-sequence is assembled serially by the aggregator (AGG). Once the aggregator is full, the multi-word sub-sequence is written to the SRAM in parallel. When the transpose buffer (TB) at an output port is empty, it receives a wide sub-sequence from the SRAM, which it then can send data out serially on the output port.

We can think of the introduction of the AGG-to-SRAM, and SRAM-to-TB transaction as stripmining the innermost loops of the original program and adding wide fetch-width loads and stores as shown in Figure 5.5. Stripmining essentially split the innermost loop into two, creating the inner sub-loop body which will be parallelized for SRAM write and read. It worth mentioning that read operation may re-access the same data, so the compiler needs to determine the loop level (not always the innermost loop) to stripmine depend on the specific access pattern. Section 5.5.2 decribes this

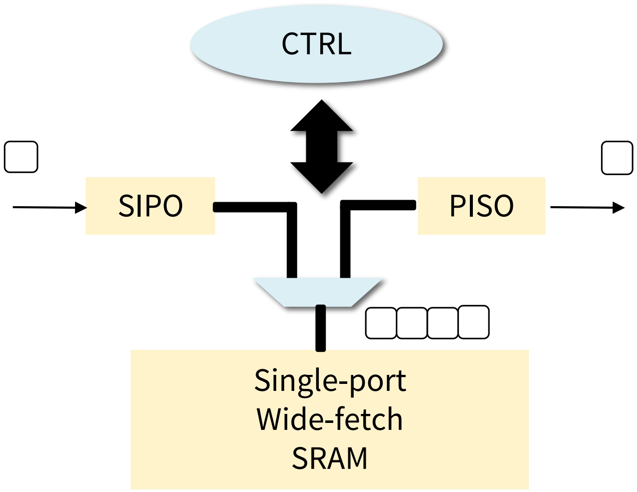


Figure 5.4: A simplified diagram of how to use single port wide fetch SRAM with serial-to-parallel(SIPO), parallel-to-serial(PISO) to imitate a multi-port memroy.

issue in more detail. The compiler generates the parallel memory access operations at the SRAM ports and records them in the abstract unified buffer. It also adjusts the schedules of aggregator-SRAM and SRAM-transpose buffer transactions to minimize the storage requirement in AGG and TB while respecting data dependencies and hardware resource limitations.

There are three transformations in the vectorization. Each transformation is associate with one of the unified buffer properties.

1. **Iteration Domain:** The consecutive memory access operation will be batched into single vectorized operation, which is as known as strip mining the iteration space.

2. **Address Range:** The range of access map need to be aligned with wide fetch word in the wide fetch SRAM. If it's not aligned, we need to pad extra access along the vectorized dimension to make sure all required data are loaded from or stored into the SRAM.

3. **Schedule:** A heuristic based schedule recipe will derive the SRAM access schedule from the schedule on the interface of the unified buffer, making sure that data in aggregator can be writtened into SRAM before it is overwritten and SRAM data can be written into TB before it's read from the output consumer.

5.5.1 Vectorization for Memory Write

Vectorization for a memory write operation is straightforward. Data is filled into the aggregator sequentially and a wide word will be written to the SRAM once it is ready. To created the vectorized operation, the original loop nest will be stripmined by the factor of fetch width and only one

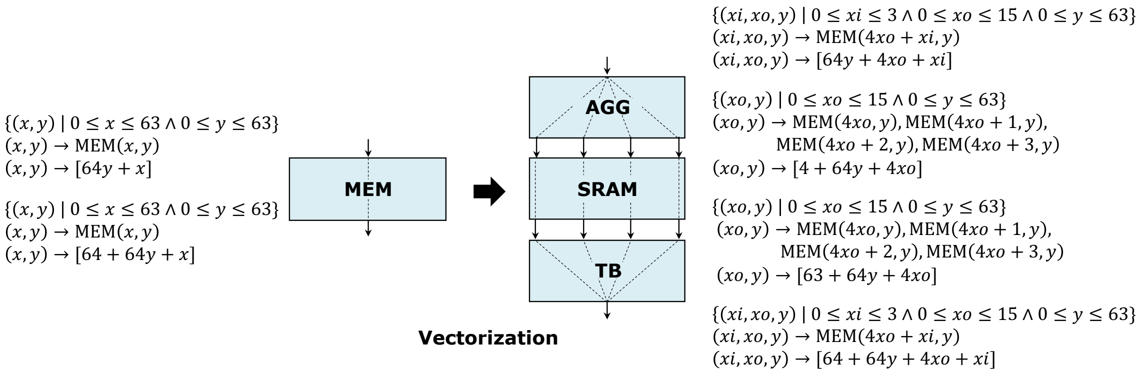

{(x, y) | 0 ≤ x ≤ 63 ∧ 0 ≤ y ≤ 63}
(x, y) → MEM(x, y)
(x, y) → [64y + x]
{(x, y) | 0 ≤ x ≤ 63 ∧ 0 ≤ y ≤ 63}
(x, y) → MEM(x, y)
(x, y) → [64 + 64y + x]
MEM
AGG
SRAM
TB
Vectorization
{(xi, xo, y) | 0 ≤ xi ≤ 3 ∧ 0 ≤ xo ≤ 15 ∧ 0 ≤ y ≤ 63}
(xi, xo, y) → MEM(4xo + xi, y)
(xi, xo, y) → [64y + 4xo + xi]
{(xo, y) | 0 ≤ xo ≤ 15 ∧ 0 ≤ y ≤ 63}
(xo, y) → MEM(4xo, y), MEM(4xo + 1, y), MEM(4xo + 2, y), MEM(4xo + 3, y)
(xo, y) → [4 + 64y + 4xo]
{(xo, y) | 0 ≤ xo ≤ 15 ∧ 0 ≤ y ≤ 63}
(xo, y) → MEM(4xo, y), MEM(4xo + 1, y), MEM(4xo + 2, y), MEM(4xo + 3, y)
(xo, y) → [63 + 64y + 4xo]
{(xi, xo, y) | 0 ≤ xi ≤ 3 ∧ 0 ≤ xo ≤ 15 ∧ 0 ≤ y ≤ 63}
(xi, xo, y) → MEM(4xo + xi, y)
(xi, xo, y) → [64 + 64y + 4xo + xi]

Figure 5.5: Unified buffer abstraction before and after vectorization. The innermost dimension x is strip-mined into xi, and xo. The unified buffer **MEM** is the memory in the brighten-then-blur application after introducing a shift register (Figure 5.1c).

operation will be preserved. The first step in the vectorization algorithm involves determining the dimension in the iteration domain which will be stripmined, based on the innermost storage dimension. Essentially, we identify the iteration domain dimension that is related to the innermost storage dimension. As an example, Figure 5.5 shows a unified buffer where `MEM` adds a 64 cycle delay between input and output streams. It has a two-dimensional iteration domain with index variables x and y and a two-dimensional address space. In this circumstance, iteration dimension x will be stripmined. The following transformation strip-mines x:

$$(x, y) \rightarrow (x\%\textsc{fw}, \lfloor x/\textsc{fw} \rfloor, y)$$

where $\textsc{fw}$ is the fetch width of the wide-fetch SRAM. This transformation creates a third iteration domain dimension for data aggregation in AGG and data serialization in TB.

Next, the compiler applies the following transformation on the ports of the wide-fetch SRAM to batch serial writes and reads into a vectorized wide word operation on the interface of the SRAM.

$$(x, y) \rightarrow (\lfloor x/\textsc{fw} \rfloor, y)$$

The above transformation is applied on the iteration domain, we also need to transform the address range of each operation, so that each vectorized fetch can be aligned with the wide-fetch SRAM word. The transformation is expressed in the following equation:

$$\text{MEM}(a, b) \rightarrow \text{SRAM}(\lfloor a/\textsc{fw} \rfloor, b). \tag{5.23}$$

However, it's possible that the address expression is quasi-affine. For instance, the address map is $(x, y) \rightarrow \text{MEM}(x + 1, y)$ where $x \in [0, 64)$ and the fetchwidth is 4, after applying the above

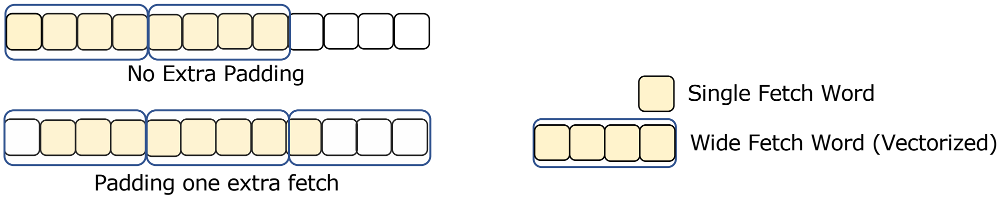


Figure 5.6: Illustration of the memory address transformation from single word address to wide word address. The upper case shows a trivial case that the boundary of the address is aligned with the wide fetch word, so that there is no extra padding. The bottom case shows the original address is skewed with wide fetch word and we need to pad one extra wide-word fetch at the end in order to cover all the needed data.

transformation, the SRAM access map becomes

$$\begin{aligned}(x, y) \rightarrow \{&SRAM(\lfloor(4x+1)/4\rfloor, y),\\ &SRAM(\lfloor(4x+2)/4\rfloor, y),\\ &SRAM(\lfloor(4x+3)/4\rfloor, y),\\ &SRAM(\lfloor(4x+4)/4\rfloor, y)\ where\ x \in [0, 16)\}\end{aligned}$$

Which can be further simplified into

$$(x, y) \rightarrow \{\mathrm{SRAM}(x, y)\ ,\ \mathrm{SRAM}(x+1, y)\}\ where\ x \in [0, 16)$$

This address map indicate that each vectorized wide SRAM fetch stretches across the boundary of wide SRAM word. In order to make it fit the affine controller we built in Physical unified buffer, we need align the address with the wide fetch SRAM word and pad one extra access on the end of the strip-mined iteration x. As is shown in Figure 5.6, this transformation makes sure that all required data is loaded from or stored into the SRAM. After this transformation applied, the SRAM access map for above example becomes

$$(x, y) \rightarrow \{\mathrm{SRAM}(x, y)\}\ where\ x \in [0, 17)$$

5.5.2 Vectorization for Memory Read

The process of vectorizing memory reads follows a similar approach to that used for vectorizing memory writes, but in a mirrored fashion. However, we want to minimize the read of SRAM, and reuse data from the more efficient transpose buffer as much as possible. In order to figure out what data will reside in the transpose buffer, we we introduce the concept of *data reuse distance*, which is defined as the size of newly written data between re-accesses to the same data. This parameter plays an important role in determining how the memory read is vectorized. If it's smaller than the capacity of the TB, the read operations will not propagate to the wide-fetch SRAM read since data can be reused within the TB. In general, we stripmining the innermost dimension and combine a number of read operation into a vectorized read from the wide fetch SRAM. However, after the reuse analysis, if the read within that loop iteration dimension can be handled by the TB, we will stripmining the next level of iteration, and the inner dimension will disappear in the iteration domain of SRAM read. On the contrary, if the reuse distance identifies that re-accessed data are too far apart to be reused, the newly written TB data will overwrite the old data, forcing data to be refetched from the SRAM to maintain the correctness of application.

Leveraging the power of polyhedral analysis and cycle accurate schedule proposed in Section 4.2, we can accurately measure the time difference between two iteration domain points that access the same address. If the size of the newly written data during that period of time is smaller than the *usable capacity* of the transpose buffer (TB):

$$\textsc{Usable\_Capacity} = \textsc{TB\_Capacity} - \textsc{Fetch\_Width}$$

the data will not be overwritten when we fetch it the next time. Considering the need to prefetch one word from SRAM-to-TB before it's read, we subtract the SRAM fetch width from the TB capacity. To filter out all data reaccess for SRAM read, the compiler needs to figure out the **innermost dimension** that exist in SRAM read iteration domain. The reuse analysis is done at the granularity of different iteration dimension levels. To represent the fact that data will be reused from TB rather than refetched from SRAM, all of the SRAM read iterations within that **dimension** will be dropped out in the access pattern transformation.

To demonstrate this optimization, let's assume we have a 1D sliding window access pattern where the row size is 32 and the window size is 5, we could access the window dimension in the inner most loop (example A in Figure 5.7) or reorder the loop to scan across the whole row and compute a partial sum (example B in Figure 5.7).

In example A, the access map is as shown in Equation 5.24 after the unified buffer is extracted.

$$Read(dx, x) \rightarrow \{\mathrm{MEM}(x + dx)\} \ where \ x \in [0, 32), dx \in [0, 5) \tag{5.24}$$

From the reuse analysis, we know iteration $Read(dx, x)$ and iteration $Read(dx - 1, x + 1)$ access the same address. Accordingly, the data are refetch with a reuse distance of 4. When stripmining the iteration domain for SRAM-to-TB data transfer, the innermost iterator `dx` is projected out. Consequently, the stripmined dimension for vectorization is the x dimension on line 4 from Figure 5.7, which is the second innermost dimension. The vectorized access map after stripmining is shown in Equation 5.25. However, as is shown in Equation 5.26 the innermost iteration dimension dx is retained on the TB read side due to data is reused in transpose buffer.

```
//Example A:
//Iteration Domain: {(dx, x) : 0<=x<32, 0<=dx<5}
//Access Map: { (dx, x)->Mem(dx+x) }
for x in [0:32]:  //dimension to be stripmined
  for dx in [0:5]:
      load(Mem(dx+x))



//Example B:
//Iteration Domain: {(x, dx) : 0<=x<32, 0<=dx<5}
//Access Map: { (x, dx)->Mem(dx+x) }
for dx in [0:5]:
  for x in [0:32]: //dimension to be stripmined
      load(Mem(dx+x))
```

Figure 5.7: Pseudo loop nest for a unified buffer that is read to conduct 1D convolution (sliding window)

Stripmining

$$\begin{aligned}
Read_{vec}(\lfloor\frac{x}{4}\rfloor) \rightarrow & \{\mathrm{MEM}(4\lfloor\frac{x}{4}\rfloor + x\%4)\} \qquad (5.25)\\
Read_{seq}(dx, x\%4, \lfloor\frac{x}{4}\rfloor) \rightarrow & \{\mathrm{MEM}(4\lfloor\frac{x}{4}\rfloor + x\%4 + dx)\} \qquad (5.26)
\end{aligned}$$

Subsequently, the address transformation presented in Equation 5.23 is applied on the right-hand-side of the access map. Illustrated in Equation 5.27 this create the address for SRAM with wide word width. Given the narrow word width of the transpose buffer, the write access map for the transpose buffer after vectorization shown in Equation 5.28 has wide address with a four-wide word configuration in the context of our CGRA single port memory, as detailed in Section 6.1.3. And these data words will be read from transpose buffer in single word granularity with sliding window patterns as is depicted in Equation 5.29.

Address transformation

$$\begin{aligned}
Read_{vec}(\lfloor\frac{x}{4}\rfloor) \rightarrow & \{\mathrm{SRAM}(\lfloor\frac{x}{4}\rfloor)\} \qquad (5.27)\\
Write_{vec}(\lfloor\frac{x}{4}\rfloor) \rightarrow & \{\mathrm{TB}(4\lfloor\frac{x}{4}\rfloor, \mathrm{TB}(4\lfloor\frac{x}{4}\rfloor + 1),\\
& \mathrm{TB}(4\lfloor\frac{x}{4}\rfloor + 2), \mathrm{TB}(4\lfloor\frac{x}{4}\rfloor + 3)\} \qquad (5.28)\\
Read_{seq}(dx, x\%4, \lfloor\frac{x}{4}\rfloor) \rightarrow & \{\mathrm{TB}(4\lfloor\frac{x}{4}\rfloor + x\%4 + dx)\} \qquad (5.29)
\end{aligned}$$

In order to better illustrate the access map after vectorization, we will use x_o to be $\lfloor\frac{x}{4}\rfloor$ and x_i for $x\%4$:

$$\textsc{Variable Substitution}$$

$$Read_{vec}(x_o) \rightarrow \{\mathrm{SRAM}(x_o))\} \tag{5.30}$$

$$\begin{aligned} Write_{vec}(x_o) \rightarrow &\{\mathrm{TB}(4x_o), \mathrm{TB}(4x_o+1), \\ &\mathrm{TB}(4x_o+2), \mathrm{TB}(4x_o+3))\} \end{aligned} \tag{5.31}$$

$$Read_{seq}(dx, x_i, x_o) \rightarrow \{\mathrm{TB}(4x_o + x_i + dx))\} \tag{5.32}$$

On the other hand, the read loop nest in example B in Figure 5.7 traverses through the entire row to generate a partial sum, leading to a larger reuse distance of 31. When creating a vectorized read operation from SRAM and stripmining the iteration space, it is apparent that the same data from different iterations within dimension dx cannot be reused from transpose buffer and thus are refetched from SRAM. So the dx is also stripmined into dx_o and dx_i by factor of 4. This situation is recorded in the transformation illustrated in Equation 5.34 and Equation 5.35, where the innermost dimension x is retained within the iteration domain.

$$Read(x, dx) \rightarrow \{\mathrm{MEM}(x + dx)\} \ where \ x \in [0, 32), dx \in [0, 5) \tag{5.33}$$

$$\textsc{Stripmining}$$

$$Read_{vec}(x_i, x_o, dx_i, dx_o) \rightarrow \{\mathrm{MEM}(4x_o + x_i + 4dx_o + dx_i)\} \tag{5.34}$$

$$Read_{seq}(x_i, x_o, dx_i, dx_o) \rightarrow \{\mathrm{MEM}(4x_o + x_i + 4dx_o + dx_i)\} \tag{5.35}$$

Furthermore, we need to create the transpose buffer write access map. We could have just copied the SRAM read access map. However, TB is operated as a circular buffer, where we push newly write data to the largest address. When we refetch the data from SRAM to TB, the TB address of the same SRAM data changes. The address is a function of the loop iteration. To generate unique address whiling writing data to the transpose buffer, we need to delinearize it from affine address expression to high dimensional expression. This high dimensional address expression will be converted back to 1D with different stride in the linearization step introduced in Section 5.6. For instance, if we have a sliding window expression $x + dx$, the following transformation is applied

$$\{(x + dx) \rightarrow (x, dx)\}$$

The outcome of this transformation is exemplified in Equation 5.37, which presents the result access map.

Address transformation

$$\begin{aligned} Read_{vec}(x_o, dx_i, dx_o) &\rightarrow \{\mathrm{SRAM}(x_o + dx_o)\} && (5.36) \\ Write_{vec}(x_o, dx_i, dx_o &\rightarrow \{\mathrm{TB}(4x_o, dx_o), \mathrm{TB}(4x_o + 1, dx_o), \\ &\qquad \mathrm{TB}(4x_o + 2, dx_o), \mathrm{TB}(4x_o + 3, dx_o)\} && (5.37) \\ Read_{seq}(x_i, x_o, dx_i, dx_o) &\rightarrow \{\mathrm{TB}(4x_o + x_i + dx_i, dx_o)\} && (5.38) \end{aligned}$$

The access pattern after variable substitution is shown in Equation 5.39, Equation 5.40 and Equation 5.41. It is important to note that apart from stripmining the innermost loop iteration x into x_o and x_i during the vectorization pass, we also stripmine the outer loop dx into dx_o and dx_i. Because dx has the address stride within a wide fetch word, the inner loop dx_i still remain preserved in the access map's inner most address expression without decoupling into a separate dimension, as is shown in Equation 5.41

Variable Substitution

$$\begin{aligned} Read_{vec}(x_o, dx_i, dx_o) &\rightarrow \{\mathrm{SRAM}(x_o + dx_o))\} && (5.39) \\ Write_{vec}(x_o, dx_i, dx_o) &\rightarrow \{\mathrm{TB}(4x_o, dx_o)), \mathrm{TB}(4x_o + 1, dx_o)), \\ &\qquad \mathrm{TB}(4x_o + 2, dx_o)), \mathrm{TB}(4x_o + 3, dx_o))\} && (5.40) \\ Read_{seq}(x_i, x_o, dx_i, dx_o) &\rightarrow \{\mathrm{TB}(4x_o + x_i + dx_i, dx_o))\} && (5.41) \end{aligned}$$

Notice that the expression in Equation 5.37 and Equation 5.29 does not specify the iteration domain bounds. As evident from Figure 5.8, due to the sliding window access pattern, only the iteration $dx = 0$ and $dx = 4$ loop has the data access block aligned with the wide fetch memory. To emcompass all the data that is required for computation, an additional wide fetch needs to be padded at the end of the row. Moreover, considering our hardware is built upon affine controllers, uniformity of the access pattern across all dx iterations is crucial. Thus, even though the fetch is redundant, an extra wide fetch must be appended to the inner loop iterations $dx = 0$ and $dx = 4$ in order to achieve this uniform access pattern.

5.5.3 Scheduling for SRAM Fetch

Before vectorization, the unified buffer only specifies the schedules on its exterior ports. Given the introduction of two vectorized operations inside the physical implementation's micro-architecture, the last step of vectorization pass schedules the aggregator-to-SRAM and SRAM-to-transpose buffer transactions.

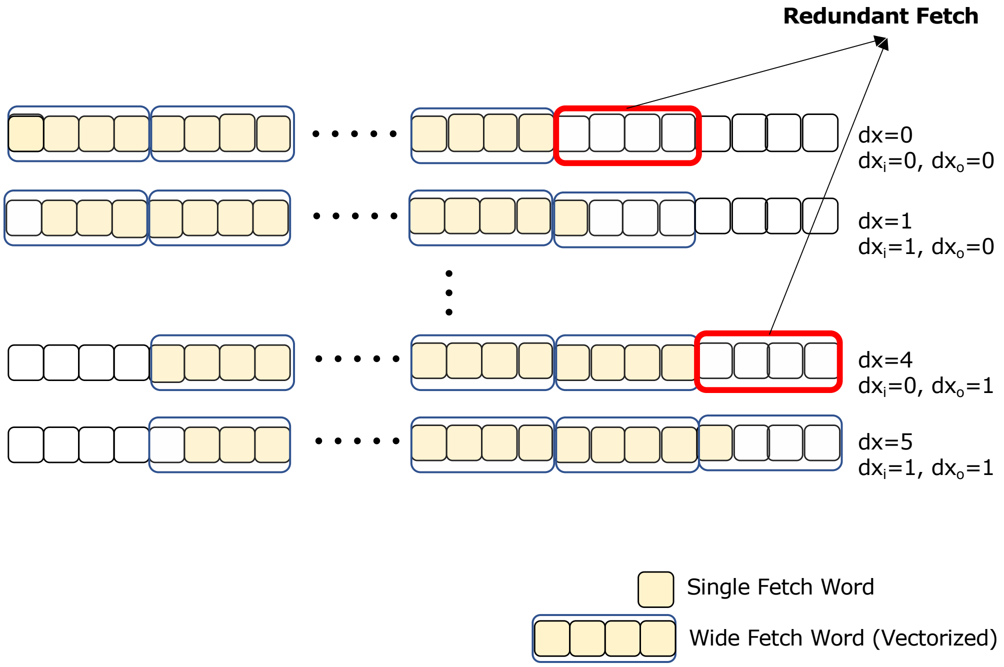


Figure 5.8: The data access pattern from the example B loop nest in Figure 5.7. Horizontally, it shows the access in the innermost x loops. Vertically, it illustrates the iterations among different dx loops. Because of the sliding window access pattern is not aligned with the wide fetch word, we need to pad one extra wide fetch to cover all the data that is going to be read.

The address map vectorization process in previous sections derives the iteration domain transformation. Those transformations are then applied to the domain of the schedule to create the correct vectorized access interval(cycle stride). Due to the fact that multiple memory operations will share the single wide SRAM port, we need to schedule the newly created SRAM operations to avoid resource conflict. The final step is figuring the offset of the schedule, which is the absolute start time. To be specific, we prioritize the SRAM write operation, which writes the sub-sequence to SRAM once the aggregator is full. Then it adjusts the schedule offset of SRAM read to minimize the storage requirement in transpose buffer. The schedule of SRAM read and write must respect SRAM port constraints and all data dependencies.

The wide fetch memory also introduces additional latency cycles. To compensate for the schedule changes, we introduce cycle padding in the innermost dimension of the schedule interval, aligning it with the fetch width multiplier. This padding was done in the II calculation during the scheduling stage described in Section 4.3. Additionally, for a memory that cannot be implemented as wire, we set the minimum latency between data write and read on the interface to twice of the fetch-width. This rationale originates from the presumption of sequential data propagation through the aggregator and transpose buffer, with round trip latency encompassing the period during which two wide words are written to SRAM. These transforms, and the physical implementation constraints are essential to create a correct global schedule in the buffer extraction stage in Chapter 4 and are done during that step.

Finally, in the compiler we explicitly break a wide fetch unified buffer into its three components: aggregator (AGG), SRAM, and transpose buffer (TB). Each component is a also a unified buffer and is represented by its iteration domain, access patterns and schedules, for the of each physical buffer. This expansion requires the compiler to generate the access map and schedule for the agg-to-SRAM and SRAM-to-TB transactions as shown in Figure 5.5. This information will be used in the compiler's code generation phase to configure the address generation and scheduling hardware as depicted in Figure 6.4b.

5.6 Address Linearization

The access pattern in the unified buffer abstraction supports an arbitrary number of data dimensions, but a physical buffer uses a linear SRAM and requires the N-dimensional addresses to be converted to a single dimension. To achieve this, an inner product is applied between each N-dimensional address $\vec{a}$ and an offset vector $\vec{o}$ that encodes the memory layout. Each element, o_i, within the offset vector represent the extent of separation in linear space when dimension a_i increments by 1. The following transformation effectively convert multidimensional address space to a single dimension:

$$\textsc{Mem}(a_0, a_1, ..., a_{N-1}) \rightarrow \textsc{Mem}((\Sigma^i a_i \cdot o_i) \bmod c)$$

The factor c is the live capacity of the buffer. It must be less than or equal to the memory capacity. This modulo factor indicate the buffer will operated in circular fashion and guarantees the live data will not be overwritten.

Consider the memory in Figure 5.5 as an example. The data size is 64×64. Polyhedral analysis identifies that there are a maximum of 64 live pixels, since it's a single row line buffer and the data will never be reused after the next row is written. Address linearization infers that a circular buffer can be implemented, so the compiler calculates the inner product of (x, y) and the offset vector $(1, 64) \bmod 64 = (1, 0)$, which results in the linear address $x \times 1 + y \times 0 = x$.

An additional instance is the transpose buffer access pattern after vectorization, as illustrated in Equation 5.38. The reuse analysis has already figured out the reuse distance of 31 surpasses the transpose buffer size of 8, necessitating a data refetch from SRAM. Consequently, the live capacity of this unified buffer is set to 8. It's important to mention that the offset of second dimension is 36 rather than 32 because of the presence an extra wide fetch from SRAM. As a result, the compiler calculates the linearization of address $(4x_o+x_i+dx_i, dx_o)$ with the offset vector $(1, 36) \bmod 8 = (1, 4)$, which results in the linear address $(x_i, x_o, dx_i, dx_o) \rightarrow \{4x_o + 4dx_o + x_i + dx_i\}$.

5.7 Unified Buffer Implementation

Upon completing all buffer mapping optimizations described in this chapter, the next step is taking all the data created in the optimization pass and converting the unified buffer into a data structure known as the **unified buffer implementation**. This data structure consists of a set of sub-blocks that closely resemble the characteristics of physical memory and can be directly mapped to the physical hardware in an one-to-one manner. Notably, each individual block is also embedded as a unified buffer or an ensemble of unified buffers for wide fetch memory, combined with supplementary meta-data tailored to code generation requirements.

In addition to the set of sub-unified-buffer-blocks, the unified buffer implementation incorporates a collection of logic ports, maintaining the same interface as the original unified buffer during the lowering process. The relationship between these logic ports and the ports of the banks is responsible for capturing interconnect information between the memory units and compute units on the reconfigurable accelerators. This connection information also contains details about port sharing, which can be further optimized using shift registers. These shift registers are derived from the port reduction optimization discussed in Section 5.1.

The resulting unified buffer implementation, serving as a comprehensive data structure, will then be utilized as input for backend code generation. It contains all the necessary scheduling and address information which guided the code generation process in the backend for configuring diverse hardware implementations.

Chapter 6

Backend Portablity

In order to achieve the goal of backend-specific code generation for various controllers, it is essential to utilize the unified buffer abstraction that captures enough information to optimize schedule at behavioral level, which can then be tailored to specific backends by providing hardware characteristic information. In previous optimization passes, the buffer mapping process required limited information about the physical SRAM macro which was passed up to the compiler. The abstract unified buffer was then broken down into multiple sub-components, namely the unified buffer implementations, which could be mapped one-to-one onto physical implementations. The next step in this process is to generate configurations for each memory given a specific memory tile implementations.

In order to enable the hardware designer to explore a vast design space to find an efficient implementations, the compiler should allow complete flexibility in the memory hardware design, which makes the compiler mapping job much more difficult. It is essential that the compiler targets various backends without compromising the functionality of the supported applications. Therefore, this chapter will explain how the backend of the compiler enables portability of the unified buffer. The design space we explore includes:

- **SRAM Macro**: On-chip memory predominantly utilizes SRAM, where the SRAM Macro exhibits diverse physical characteristics. Notably, these characteristics encompass variations in the number of ports, fetch width, and depth (referred to as capacity). In order to support different physical memory implementation, our compiler incorporates the SRAM Macro characteristic as physical constraint and passed this information to the buffer mapping stage, described in the Chapter 5. Subsequently, the abstract unified buffer was then broken down into multiple sub-component, included in the unified buffer implementation, each of which can be seamlessly correlated with individual physical implementations and a code generation pass will be executed to create the runtime configurations.

- **Controller Specialization** : Involves deciding whether and how to create specialized controllers on the target architecture. If there is no dedicated controller available on the hardware, the compiler must synthesize the hardware logic using existing reconfigurable resources such as processing elements (PEs) on CGRAs or look-up tables (LUTs) on FPGAs. If the block contains a dedicated controller, the compiler must incorporate a specialized code generation process to create programs or configurations that run efficiently on that specific hardware. This enables the optimization of resource density for the particular domain of access pattern, resulting in improved energy and area efficiency.

- **Timing Constraints** : At present, domain-specific accelerators can be divided into two categories based on their timing constraints. The first type of system uses static timing, where all memories are synchronized by a global clock that counts the cycle after starting the application. The second type of system employs a dynamic micro-architecture, which leverages a ready-valid handshake mechanism between different on-chip buffers and compute kernels. In Section 6.2, we will introduce a memory implementation, Buffet, which uses ready-valid timing protocol. Generating code for Buffet demonstrates that while our compiler proposes a cycle-accurate schedule, it extracts sufficient information to support both static timing and ready-valid timing protocol on the reconfigurable accelerators.

6.1 Physical Unified Buffer (PUB)

To demonstrate the backend portability of unified buffer abstraction, we leverage a flexible physical buffer hardware generator to create a number of different hardware implementations of unified buffers, each with increasing efficiency. In this section we explore buffers which are statically scheduled.

6.1.1 Dual-Port SRAM

The simplest hardware implementation of a unified buffer wraps a dual-port SRAM with logic that computes the addresses and sequences of read/write enables for the iteration domain at each port (Figure 6.1). Since all implementations have a finite size, the design also contains logic for chaining multiple physical buffers into a larger buffer.

To implement a naïve physical buffer, we place three modules at the input and output ports of the memory. These modules are IterationDomain (ID), AddressGenerator (AG), and ScheduleGenerator (SG). They provide implementations of the corresponding components on the ports of the unified buffer abstraction. The IterationDomain module implements counters corresponding to `for` loops, while the AddressGenerator and ScheduleGenerator modules implement mapping logic from an IterationDomain module to an address and a read/write enable for the associated memory port.

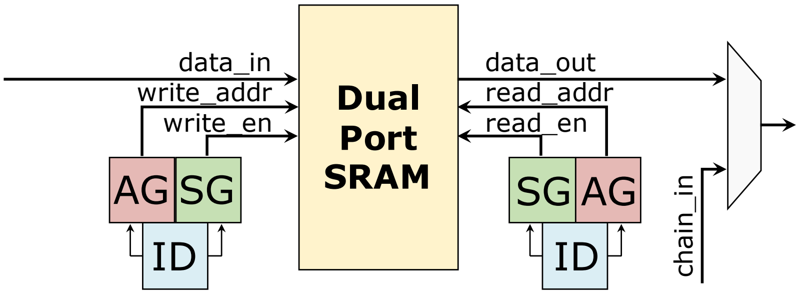


Figure 6.1: A common physical buffer implementation with a dual-port SRAM. Two IterationDomain (ID) modules each drive an AddressGenerator (AG) and a ScheduleGenerator (SG) to orchestrate writes to and reads from the memory. The output has a multiplexer for memory chaining.

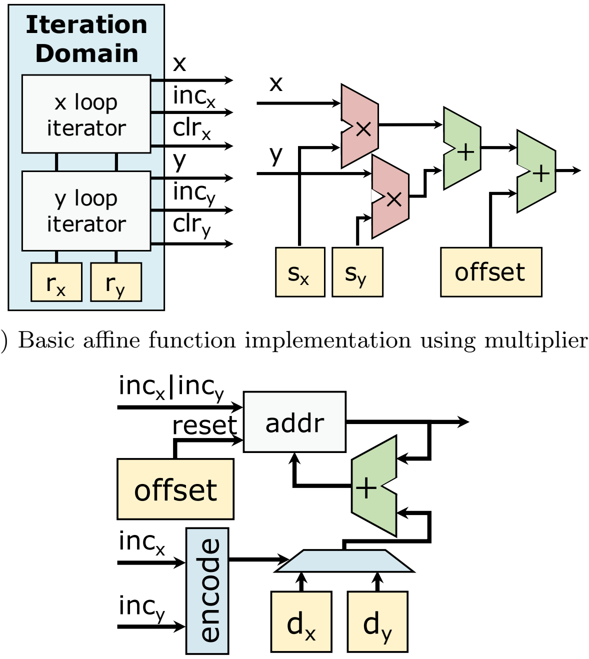
6.1.2 Controller Specialization

The AG and SG modules can be described as affine functions of the iteration domain. For instance a two-dimensional affine function ($s_x * x + s_y * y + \text{offset}$). A straightforward hardware implementation of this two-dimensional affine function would use the design with two multipliers and adders combined with two counters shown in Figure 6.2a. For the experiment that using a CGRA without dedicated controller, we logic synthesize this implementation using processing element on our CGRA.

To improve the area and energy efficiency, we can replace the multipliers with adders by realizing that the incoming x and y values come from counters, so each multiplier output can be generated by a simple recurrence relation: $\text{out}(i{+}1) = \text{out}(i) + d$, where delta d is the amount the multiplier output increases with each update. Furthermore, we can reduce the required hardware to a single adder by realizing that at any update, only one loop variable is incrementing (and many may be reset). This means we can precompute how much the affine function should change when each loop increments, and we can express the affine function $\text{A}(x, y)$ as a state transition from iteration i to iteration $i{+}1$. This turns into the recurrence relation $\text{A}(x, y)_{i+1} = \text{A}(x, y)_i + (\texttt{inc}_y ?\ \texttt{d}_y : \texttt{inc}_x ?\ \texttt{d}_x : 0)$, where $\text{A}(x, y)_0 = $ offset, $\texttt{inc}_y$, $\texttt{inc}_x$ are booleans that indicate whether to increment, and $\texttt{d}_x$, $\texttt{d}_y$ are increment deltas. With this transformation, the whole function can be implemented with one adder as shown in Figure 6.2b. Figure 6.3 shows an example of the relation between the strides, ranges, and deltas for a simple downsample-by-2 traversal of an 8×8 image. Since we only need the delta for one loop variable at a time, we only require a single adder and a register along with a multiplexer to increment the running address by the delta of the outermost loop variable that is incremented.

6.1.3 Wide-Fetch, Single-Port SRAM

While dual-port SRAMs are often used for an FPGA or an ASIC, they are not the most efficient push memories for two reasons: first, dual-port SRAMs can be more than two times larger than their single-port counterparts for the same storage capacity while consuming 40% more energy per



(a) Basic affine function implementation using multipliers.

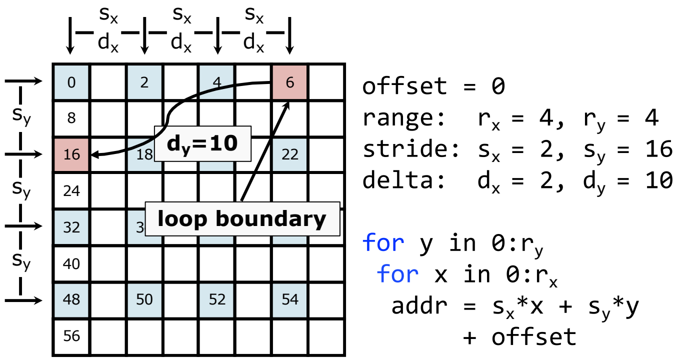
(b) Affine function as a recurrence relation.

Figure 6.2: Area optimizations in the affine function hardware for address and schedule generation with a two-dimensional iteration domain. **(a)** An implementation that uses the value of the counters in the iteration domain. **(b)** An implementation that embeds the address delta between loop levels.

access [61]. Second, energy per accessed byte is often lower if more data is fetched from an SRAM on each cycle [80]. Thus, in custom-designed memories, wide-fetch single-port memories are typically used to emulate multiple ports to improve energy per access.

To emulate simultaneous reads and writes with a single-port SRAM, we create a physical buffer that consists of three buffers, as shown in Figure 6.4. One of the buffers is a large SRAM, while two small buffers are placed on either side of the SRAM. The smaller buffers are implemented using registers/register files, and contain eight to sixteen words (2-4 fetch blocks) when a four-word fetch SRAM is used. The small buffer between the input port and the SRAM (aggregator: AGG) serves as a serial-to-parallel converter and the buffer between the SRAM and the output port (transpose buffer: TB) serves as a transpose buffer for small blocks (4×4 for our design) and a parallel-to-serial converter. To maximize the utility of this buffer, we gave it two input and two output ports, the maximum a four wide SRAM could support, and implemented logic to support port sharing. We instantiated an ID and AG at the select line of a multiplexer that chooses which port accesses the SRAM at any given time. Figure 6.4a shows a block diagram of the physical implementation of a push memory with two input ports and two output ports.

The performance of this memory depends on how much of the data in the wide fetch can be used.



```
offset = 0
range:  r_x = 4, r_y = 4
stride: s_x = 2, s_y = 16
delta:  d_x = 2, d_y = 10

for y in 0:r_y
 for x in 0:r_x
  addr = s_x*x + s_y*y
       + offset
```

Figure 6.3: An example of a simple downsample-by-2 iteration pattern over an 8×8 image. The relationship between the strides and deltas for a two-level loop nest are shown.

The design can maintain maximum performance in the common case, when the inner stride is one. The TB also allows this structure to transpose a matrix at full rate, if it is fetched in 4×4 blocks.[1] To make the best use of this wide access memory requires the access patterns to be vectorized to automatically decouple the access pattern into sub-sequences and map them onto the controllers shown in Figure 6.4. This is performed in the vectorization pass described in the buffer mapping stage (Section 5.5). Of course, the bandwidth per port decreases when the access stride increases. Since either the input or the output can always be streamed sequentially with inner stride = 1,[2] in the worst case, the memory supports a throughput of 4 words every 5 cycles. For example, there is 1 write and 4 reads needed for 4 words.

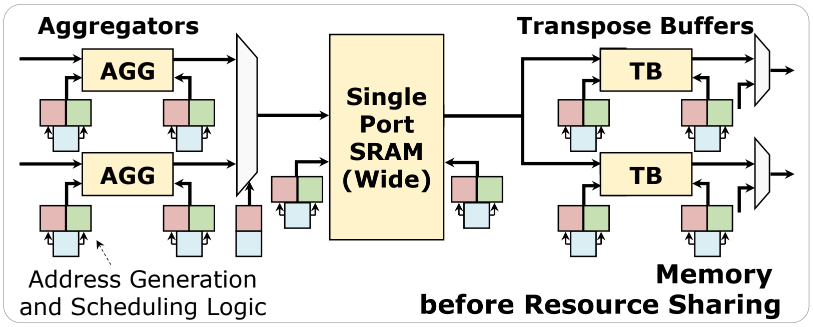
6.2 Buffet

Buffet is an on-chip memory paradigm that employs a buffer implementation idiom with explicit decoupled data orchestration (EDDO) to enable efficient on-chip memory access, particularly in deep learning applications. It's important to underscore that Buffet is not a compiler abstraction, but rather a hardware primitive that can be utilized as a substitution for other on-chip memory, such as cache, scratch pad or FIFO. Therefore, Buffet can direct substitute the physical unified buffer as described in Section 6.1, and is an example of a buffer which uses ready/valid and not static timing.

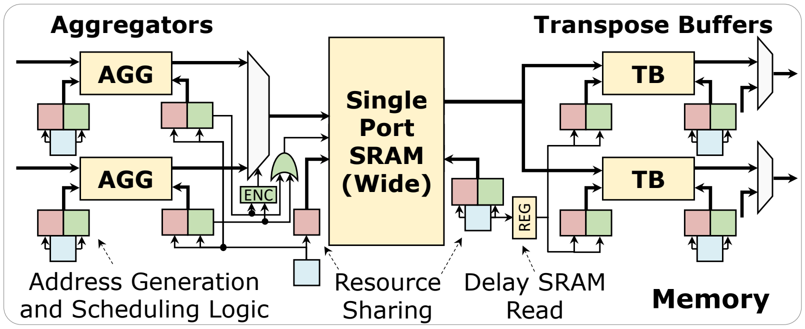
The Buffet architecture comprises a buffet controller and a collection of physical memory banks, both of which are hardware primitives that can be statically allocated and used as a local scratchpad

[1] For a transpose, the SRAM is fetched in column order, where each fetch returns a short row of 4 elements. This access pattern fetches 4 stacked rows from the memory. The output port then reads the first element of each row, outputting the first column, and then the 3 other columns, before the next set of rows are fetched. When this inner loop completes, you have written the first four columns into the destination memory, and the outer SRAM loop moves to the next set of four columns.

[2] If the write address is not consecutive, we can always apply a linear transformation on the data layout to create a pattern that access the data contiguously.


Aggregators
Transpose Buffers
AGG
AGG
Single Port SRAM (Wide)
TB
TB
Address Generation and Scheduling Logic
Memory before Resource Sharing

(a) Before resource sharing.



(b) After resource sharing.

Figure 6.4: Diagram of a PUB with a wide-fetch single-port SRAM, aggregator (AGG), and transpose buffer (TB). Sets of ID/AG/SG controllers control the input and output of each sub-component.

by multiple compute units. The buffet controller provides data staging and synchronization mechanism, facilitating decoupled write, read and update. These primitives contribute to efficient on-chip data reuse.

Figure 6.5 shows the operations supported by buffets. Producer hardware units can `fill` data into the SRAM macro encapsulated in a buffet. `Fill` operation essentially performs sequential write, appending data to the existing live data block. Consumer hardware can `read` or `update` any live data in the buffet. Furthermore, a `shrink` operation can be invoked to advance the live window, thereby releasing space for future `fill` operations. `read`, `update` and `shrink` operations are associate with a address to identify the location of the data, as is illustrated in bottom right of the Figure 6.5. It worth noting that buffet rely on an external hardware to generate address.

One notable feature of the Buffet architecture is its utilization of dependency checking mechanism, which synchronize the producer and consumer operations. The `fill` operation operates within a distinct control thread and remains decoupled from the other three operations. This design allows the controller to stall any `read` or `update` operation that violates a read-after-write dependency, as well as any `fill` operation when the memory is full. To support dynamism in the application it adopts a ready-valid interface for both the address and data path as is depicted in Figure 6.5. Specifically, the same address generator which designed for our physical unified buffer (described in the Section 6.1) is employed in the Buffet backend for a fair comparison in the evaluation.

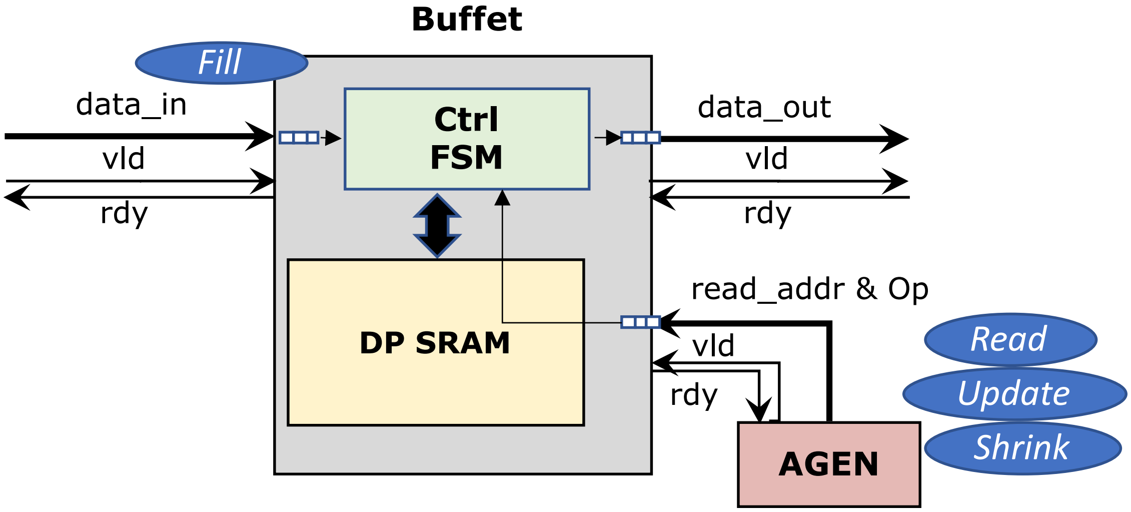


Figure 6.5: Buffet storage idiom, utilizing ready-valid timing protocol

Table 6.1: The characteristics of our PUB memory primitive and alternative memory implementations. Our compiler, using the unified buffer abstraction, supports more memory implementations as compared to FPGA compilers and other accelerator compilers.

Memory Backend	**PUB**	DP-SRAM +AG	DP-SRAMs + PEs	Ready-valid (Buffet)	BRAM + LUTs
SRAM Macro	**SP**	DP	DP	DP	DP
Build-in AG	**Yes**	Yes	No	No	No
Control Protocol	**Static**	Static	Static	Ready-valid	Static
Accerlerator Architecture	**CGRA**	CGRA	CGRA	ASIC	FPGA

6.3 Code Generation for Various Hardware Backend

The unified buffer abstraction provides our compiler with the ability to map to a wide range of physical memory implementations, as demonstrated in Table 6.1. Importantly, this is possible because the unified buffer abstraction captures sufficient information regarding the data movement behavior, and because the frontend optimizations are either generic or could be configured using only a small amount of characteristic information for the specific backend. By maintaining all of the data movement and scheduling information together, our backend can efficiently select the essential data that it needs to configure or generate the target hardware.

Figure 6.6 sketches the compilation process, denoting the classes of hardware our system can target. Our compiler first transforms the arrays in the loopnest describe in Halide IR into unified buffers with schedule optimizations described in Section 4.2. These steps schedule all operations on the buffers' interface, creating the addressing and scheduling information. To ensure the scheduler optimally exploits the resource for memory access, it's necessary to convey the count of logical port to the scheduler. This indicates the number of parallel access can that can be executed. Moreover, due to the increasing complexity of the single port wide fetch memory, the latency and fetch width information also need to pass up to scheduler to guarantee the validity of the schedule.

Subsequently, the compiler employs mapping optimization including port reduction optimization, chaining and banking described in Chapter 6 to get the optimized unified buffer implementation. As for buffer mapping stage, the SRAM macro specification, including capacity, word width and number of port is provided. This information is utilized within this stage to guide the compiler break the abstract unified buffer into implementations that satisfied the physical constraints. Specifically, if it is specified to use the PUB with wide fetch SRAM Macro, the vectorization pass will be invoked in this process.

The final stage involves the buffer mapping code generation. This step is bifurcated into two different backends: one for generating configurations for CGRAs and one for targeting FPGAs. For FPGA based implementations, leveraging the code generator introduced in [38], the unified buffer implementation is converted into C-loops which are then fed into an FPGA HLS(High Level

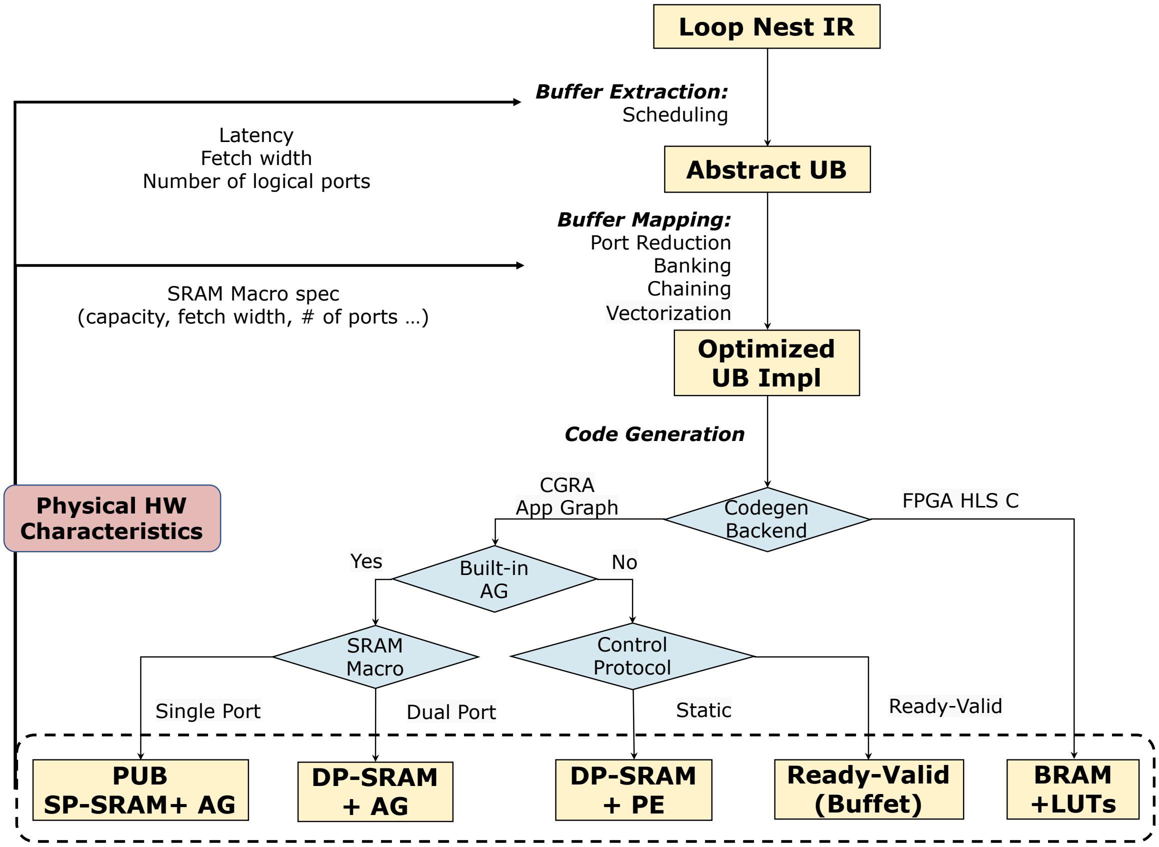


Figure 6.6: Compilation process targeting different memory backends.This diagram shows the compiler process for different hardware targets. All applications start with Halide which is extracted and mapped using the Unified Buffer Abstraction. From here, the usage of FPGAs, address generators, SRAM Macro, and control protocol determines which backend mapping to use. These options include BRAMs+LUTs, Ready-Valid Buffets, DP-SRAMs + PE, DP-SRAMs + AG, and our physical unified buffer(PUB) implemented by single port SRAM with address generators.

Synthesis) tool to logic synthesize the design using resource on FPGA fabric including block RAMs and lookup table (**BRAMs + LUTs**). For CGRA targets, a different branch of action is taken. The memories are connected to the hardware compute kernels through the port connection relation extracted in the unified buffer implementation data structure described in Section 5.7, forming an application graph which is then mapped onto the CGRA.

Our compiler encounters a divergence point in the CGRA backend based on whether controllers are pre-built into physical buffers. If the buffers do not contain controllers, the compiler will logic synthesize the controllers, followed the logic in Figure 6.2a. For a n-dimensional affine expression, this controller design will utilize n counter, n multiplier and $n-1$ adder, where the number of dimension is tailored for the specific stream pattern each buffer requires. These controllers are then added to the non-memory part of the application graph that is mapped into PEs on a CGRA. This part of the compilation covers machines that have simple dual-ported memories as physical buffers, **DP-SRAM+PE**, as well as more complex memories like **Buffets** shown at the bottom right of Figure 6.6.

A Buffet [67] is a more sophisticated buffer implementation that monitors data dependencies between the read and write ports, and supports a ready-valid interface. The schedule established during the buffer extraction phase is static and cycle accurate, which means all memories are synchronized by a global clock that count the number of unstalled cycles after starting the application. When mapping to ready-valid timing protocol, our compiler eliminates the schedule generator and drops the cycle-level scheduling information between write and read operations. Instead, it uses this information to ensure that write to read, and read (to free) to write dependencies are maintained. Thus, the global schedule is turned into a local schedule, which controls the delay and rate of read operation relative to the write operation, instead of using a global cycle count. Given the inherent dependency checking between write and read operations[3] in Buffet, the schedule could be further simplified by separate read and write into decoupled threads. Only the relative sequencing within each thread (between `read` and `update`) is preserved, as the address generator will issue those operations in order. To generate the free command, the mapper leverages the cycle accurate schedule, to determine the iteration domain point (loop level) which indicates that the last read operation in the coarse grained tile has completed. And that iteration domain event will trigger the send of the free signal to the Buffet.

Since Buffets use a ready/valid timing protocol, in our evaluation section we needed to create a ready/valid CGRA. This required that all the PEs and interconnect in the CGRA support ready-valid ports. Specifically, supporting broadcast logic and shift register within a ready-valid system involves data orchestration and propagation for multiple destinations. Adding this support to our CGRA were the major changes needed to support buffets.

If the controllers are built into the physical buffers, the compiler configures the embedded controllers to implement the stream access patterns on all of its ports. It will extract a `range` and a `stride` parameter per dimension of the affine expression. Furthermore, a `offset` will be extracted to indicate either the starting address or the earliest clock cycle that the operation is executed. This path is taken for the physical buffers described in Section 6.1, **DP-SRAM+AG** and **PUB**, which are the left bottom branches of Figure 6.6.

[3] write corresponding to `fill` operation in Buffet, while read encompasses both `read without update` and `read with update`,

Chapter 7

Evaluations

This chapter evaluates some of the benefits of using the unified buffer abstraction throughout the compilation process. The next section describes the evaluation methodology used to generate the results presented in this chapter. Using this method, Section 7.2 compares different unified buffer implementations, and demonstrates the performance gains possible by having a clean interface between the compiler and hardware. This clean interface also makes it possible to improve application performance through better compiler code scheduling/optimization, as shown in Section 7.3. Finally, Section 7.4 demonstrates the advantage of CGRAs over FPGA solutions, which can only be achieved with if the compiler uses a higher level model for streaming memories.

7.1 Evaluation Methodology

To evaluate our compiler, we use it to compile the applications listed in Table 7.1 to CGRAs, and compare the resulting performance to a Zynq UltraScale+ 7EV FPGA. The applications span stencil operations in image processing and tensor operations in deep neural networks as found in previous Halide scheduling papers [2, 60]. To generate an FPGA bitstream, our compiler transforms the buffers in each Halide application into unified buffers and applies all optimizations. We build upon the work by [38] to generate synthesizable C code that we feed into Vitis HLS. Our system generates identical synthesizable C as Huff et al., which compares favorably to other competitive DSL-FPGA systems [38]. The HLS output is fed into Xilinx's Vivado system that synthesizes, places, and routes the resulting design at 200 MHz. We use Vivado [87] to report resource consumption, energy use, and performance.

Our CGRA, shown in Figure 7.2, resembles an island-style FPGA, with LUTs replaced by processing element (PE) tiles with 16 bit integer/floating-point ALUs, and BRAMs replaced by memory (MEM) tiles with different unified buffer implementations, including our optimized PUBs. The CGRA is embedded in a full system-on-chip (SoC). As is illustrated in Figure 7.1, the CGRA directly

Table 7.1: Halide applications used in the evaluation section. All stencil applications utilize a 64x64 input image size. The ResNet conv-layer operates on a 28x28 image size with the same input/output channel size IC=OC=16. The MobileNet layer processes a 28x28 image with 4 depthwise channels and 3 pointwise channels. The GEMM kernel multiplies two 64x64 matrices. The SR-CNN begins with a 30x30 input image and performs a 1x1 convolution with IC=64 and OC=8. Subsequently, a 3x3 convolution follows with IC=OC=8. The final layer consists of a 1x1 convolution with IC=8 and OC=64.

Application	Type	Description
gaussian	stencil	3×3 convolutional blur
harris	stencil	Corner detector using gradient kernels and non-maximal suppression
upsample	stencil	Up sampling by repeating pixels
unsharp	stencil	Mask to sharpen the image
camera	stencil	Camera pipeline with demosaicking, image correction, and tone scaling
gpyr	stencil	Gaussian pyramid with four levels of down sampling
laplacian	stencil	Laplacian pyramid with three levels
resnet	DNN	ResNet layer using multi-channel convolution
mobilenet	DNN	MobileNet layer using separable, multi-channel convolution
gemm	DNN	General matrix multiplication
SR-CNN	DNN	A three-layer convolutional neural network specialized for super resolution

connects to a large multi-banked, double-buffered memory called the global buffer. The global buffer has 16 banks; each bank is 256 kB and connects to a different section of the top edge of the CGRA. The data tiles required by the CGRA are first brought into the global buffer and then streamed into the CGRA. This allows computation on the current tile in the CGRA to be overlapped with the movement of the next tile into the global buffer. The global buffer provides deterministic access latency to the CGRA and hides the non-deterministic latency of the main memory.

Applications are written in Halide, and scheduling primitives are used to define tiling and buffer allocation. Additionally, a physical hardware constraints file is provided to the compiler as collateral, facilitating backend identification. Following the creation of a Halide application, all further steps happen automatically without manual annotation or interventions. When targeting the CGRA, our compiler outputs a logical description of the design that is fed into custom mapping, placement, and routing tools designed for this CGRA. To generate power and area numbers, we created a complete Verilog design of the CGRA and used Cadence Genus and Innovus tools to synthesize, place, and route the MEMs and PEs of the CGRA in a 16nm technology at 900 MHz. Power numbers are extracted from gate-level simulations.

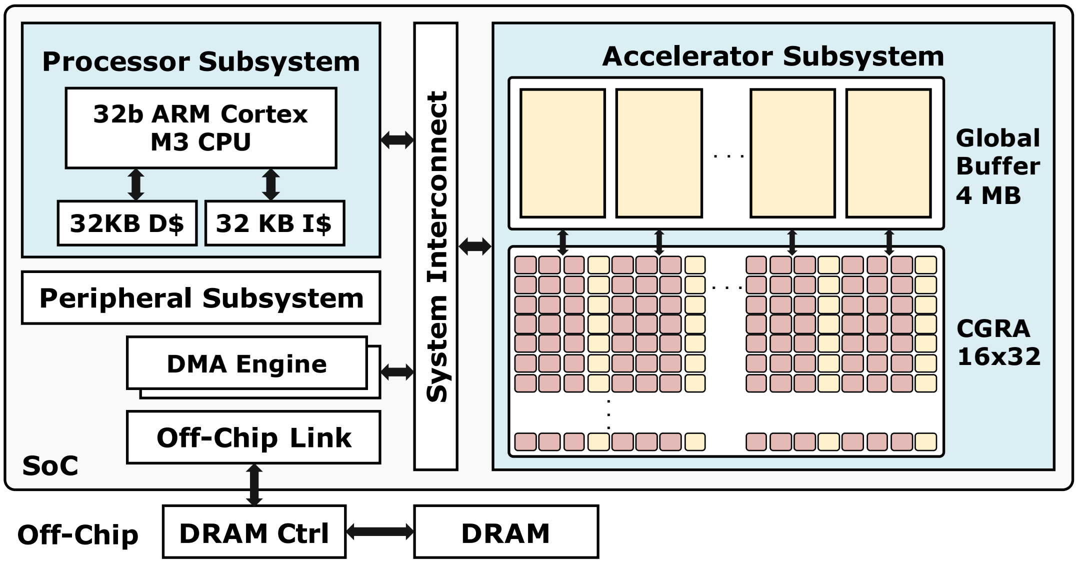


Figure 7.1: SoC consisting of the CGRA and the global buffer.

7.2 Hardware Optimization Case Study

We leverage our flexible backend to compare different approaches for physical buffer design on a CGRA. There are two major ways to build on-chip storage [68]: you can either use reconfigurable compute units to do both computation and address generation [3][57] (DP-SRAM + PEs and buffet) or you can include address generators in the memory components [44][69] (DP-SRAM + AG and PUB). Comparing the second row with the third row of Table 7.2 shows that even for a simple application, it is more efficient to use embedded address generators than to use the processing elements (PEs) on the target platform. Adding this logic to a dual-port 2048 × 16 bit SRAM (Figure 6.1) reduces the total unified buffer area by 46% and energy by 25% compared to implementing the addressing and control on PEs. We achieve further improvements by replacing the dual-port (DP) SRAMs with single-port (SP) SRAMs. The area of the dual-port 2048 × 16-bit SRAM is around 2.5× larger than a single-port 512 × 64-bit SRAM with the same capacity. Thus, as the fourth row of Table 7.2 shows, even with the extra aggregation and transpose logic, using a wider single-port SRAM results in a buffer that is 18% smaller and consumes 31% lower energy than the best dual-ported version. The advantages of adopting a single-port SRAM are further demonstrated by the power outcomes illustrated in Figure 7.3. Across all benchmark applications, the memory with a single port wide-fetch width (PUB) exhibits better power performance compared to the dual-port counterpart with a single-fetch width. From the breakdown of power, the reduced energy per access associated with the single-port SRAM configuration lead to energy efficiency for the whole memory tile.

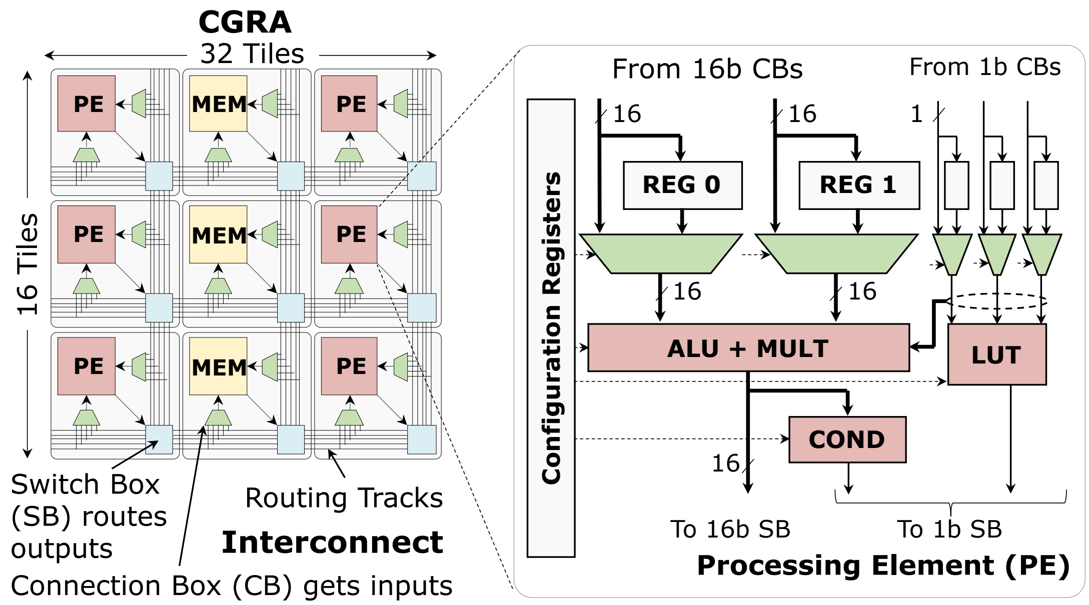


Figure 7.2: Our CGRA is a 16 × 32 array of processing element (PE) and memory (MEM) tiles. One-fourth of the tiles are MEMs and the rest are PEs. The memory tile contains the optimized PUB described in Section 6.1.3 depicted in Figure 6.4b.

7.2.1 Comparison with Buffet

We synthesize a buffet implementation with the same dual-port 2048 × 16-bit SRAM macro. The SRAM area proportion is lower than what is reported by [67] due to the added interconnect needed for CGRA reconfiguration. Although the buffet controller takes a smaller proportion of area, its function is only equivalent to the schedule generator in our PUB controllers, and does not contain the address generation capability. Even if we ignore the missing address generator, our PUB memory is smaller and more area efficient. Using more complex controllers and single ported memories seems to be the best strategy for building physical buffers.

The advantage of the PUB implementation is even larger than Table 7.2 indicates. Our PUB's four-word-wide fetch allows it to support two input ports and two output ports, which double the peak read/write bandwidth compared to the dual-port memories. As shown in Table 7.3's left two columns, PUB requires fewer physical buffers for all applications besides resnet.

As for the energy per access, our non-optimized dual-port memory consumes less energy per-fetch compared with a Buffet. Although the Buffet energy we report does not include the address generation logic, the more optimized PUB is 35% more energy efficient. This energy saving results from the wide fetch SRAM bundling multiple reads or writes into one access, which amortizes the memory and controller energy over multiple memory access.

[1] Since a buffet does not have an address generator in its design, the area and energy shown in the table do not include address generation.

Table 7.2: Total memory area and energy for a 3 × 3 convolution using different implementations of the physical on-chip storage. Both area and energy decrease as we specialize the physical buffer. Total area and energy include control logic and address generation except for buffet.

	References	MEM Area (μm^2)	SRAM Area (%)	MEM Energy (pJ / access)	SRAM Energy (%)
Buffet[1]	[67]	14.3k	86	3.9	84
DP SRAM + PEs	[3][57]	31.1k	40	4.8	70
DP SRAM + AG	[44][69]	16.7k	74	3.6	92
4 Wide SP SRAM +AGG +TB +AGs	Ours	13.7k	42	2.5	61

While using wide-fetch memories has many benefits, it also has some costs. It requires the generated schedules to be padded to align with the fetch width. Table 7.4 shows the RTL simulation latency data using the same halide schedule. Comparing our PUB latency with other single fetch width variants illustrates that this padding usually does not affect the latency, but can have a modest effect if the size of some of the data blocks is small, as it is in resnet. In this case, the convolution neural networks have complicated memory access pattern where we need to pad on the edge to make it aligned with our memory fetch width.

Table 7.3: Memory usage comparison between different physical memory implementations(smaller is better).

# of memories	PUB (ours)	DP+AG	Buffet
upsample	1	2	2
gaussian	1	2	6
harris	5	11	23
resnet	80	80	80

Table 7.4: RTL simulation latency comparison between different physical memory implementations(smaller is better)

Latency (cycles)	PUB (ours)	DP+AG	Buffet
upsample	16399	16383	16401
gaussian	4095	4095	4100
harris	4095	4095	4139
resnet	9807	8739	8751

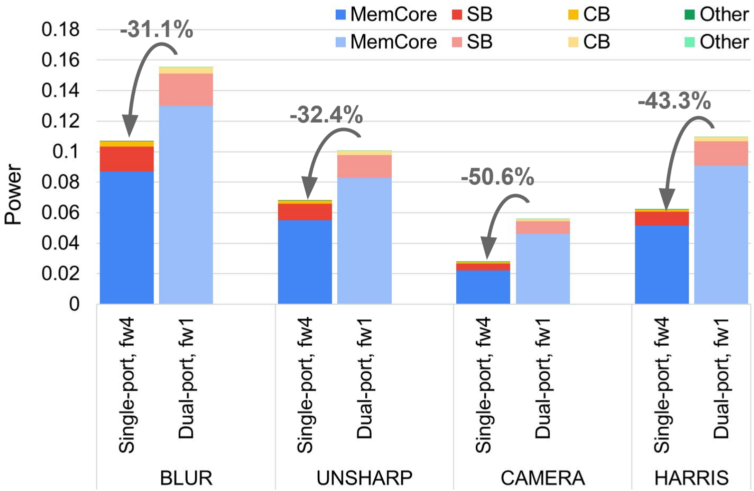


Figure 7.3: Power consumption compared between DP-SRAM +AG and PUB among different applications.

7.2.2 Full Application Area Evaluation

Finally, we map nine applications to three CGRA architectures with different physical buffer implementations and evaluate their total area. To map to the memory backend without address controllers, we generate the controllers using PEs based on the number of dimensions in the unified buffer schedule. In our unoptimized version (1), we make no modifications to the PEs. In the optimized version (2), an operator for a counter is added to the PE; this change saves 67% area. As shown in Figure 7.4, using dedicated address generation (AG) logic in the dual-port memory tile lets PEs be used exclusively for computation. Breakdown of the PE count is shown in Table 7.5. Note that these area savings occur while the throughput stays the same. Furthermore, using a wide, single-port SRAM with more external ports saves silicon area, while expanding functionality. Both of these properties lead to an average 2.2× less total area needed to implement the same application as compared to DP + optimized PEs. From the breakdown in Figure 7.4, SRAM macro area reduces 3.3 times, and memory controller area reduces 4.5 times, while PE area remains the same. The area savings are even greater in deep learning applications, which are memory intensive.

This case study shows the importance of building physical buffers with efficient and customized controllers that can extract the most performance out of each memory macro. Performing these optimizations yielded a physical memory implementation that is half the area and energy of the original design. Of course these optimizations are only possible when one has a flexible compiler with backend portability. This combination empowers hardware designers to navigate the design space of memory physical implementations, providing the freedom to explore various configurations and strategies. This approach ensures that memory designs are optimized to meet the specific

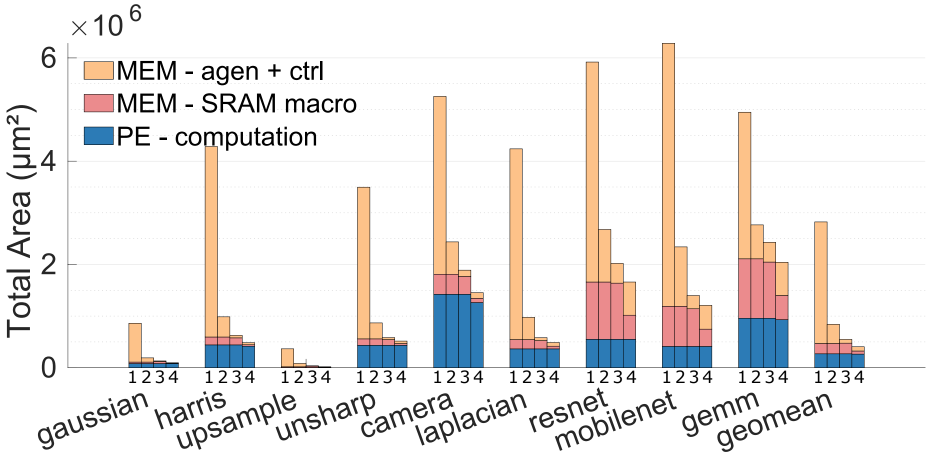


Figure 7.4: Comparison of area for different memory implementations with stacked bars divided into MEM addressing and control area, MEM SRAM macro area, and PE area used for computation. Four different implementations of a physical buffer are evaluated: (1) a dual-port (DP) SRAM with unmodified PEs for address generation, (2) a DP SRAM with PEs optimized for address generation, (3) a DP SRAM with optimized address generator (AG), and (4) our final PUB with a single-port SRAM with fetch width of 4, aggregator (AGG), and transpose buffer (TB) each with their AGs.

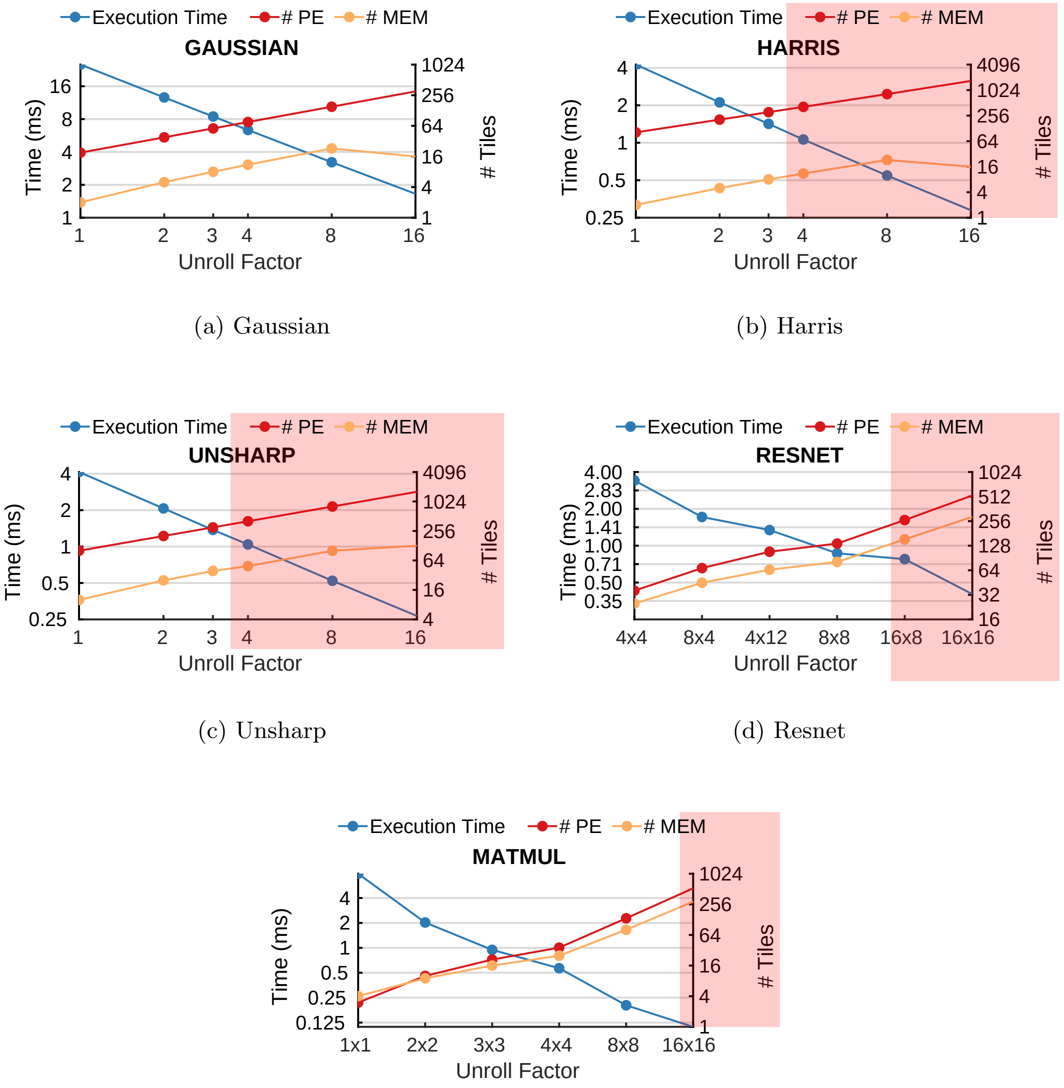
requirements of diverse applications, thus enhancing overall system efficiency.

7.3 Evaluation of Compiler Optimizations

The goal for the unified buffer abstraction is to optimize the application mapping to a reconfigurable architecture, which explores the maximum throughput that fully utilizes all resources. Meanwhile, several automatic optimization passes are implemented in the flow to increase the computation occupancy and reduce the memory consumption.

7.3.1 Computation Resource Aware Scheduling

During the buffer extraction stage, a resource aware scheduling algorithm is applied to help the user to fully utilize the massive pool of hardware on an reconfigurable accelerator. As users, we could leverage high level scheduling language, such as Halide, to generate the loop nest description language introduced in Section 4.1. This allow us explore the trade-off between latency and resource utilization. In our approach, we reinterpreted the loop nest intermediate representation (IR), where loop unrolling takes on the role of executing different loop iterations in parallel, each assigned to dedicated hardware resources. Figure 7.5 shows how system performance scales when we employ loop unrolling and allocate dedicated compute hardware in parallel. Notably, the execution time exhibits a linear decrease on the logarithmic scale, indicating that our designs are highly scalable in



(a) Gaussian

(b) Harris

(c) Unsharp

(d) Resnet

(e) Matmul

Figure 7.5: Execution time versus resource utilization tradeoff by using Halide's scheduling. At high unrolling factors, designs do not fit on the CGRA (384 PEs, 128 MEMs); this is indicated on the charts by the red shaded regions.

Table 7.5: The table compares the utilization of Processing Elements (PEs) across various memory implementations. The first column indicates the additional PEs incorporated for address generation, corresponding to implementation (1) in Figure 7.4. The second column denotes the count of extra PEs when optimized specifically for address generation, corresponding to implementation (2) in Figure 7.4. The third column shows the baseline number of PE used for computation contributing to the blue bar in Figure 7.4

# of PE	Extra PE for AGEN	Extra PE for AGEN (opt)	PE for computation
gaussian	156	19	19
harris	768	91	103
upsample	66	15	0
unsharp	612	72	101
camera	656	146	331
Laplacian	660	100	89
resnet	755	237	128
mobilenet	918	268	96
GEMM	508	353	223

terms of execution time when additional resources are utilized. Furthermore, optimizations in buffer mapping stage, including shift register creation and banking, help create a buffer implementation that satisfies the bandwidth require of the parallel computation unit. Combined with the resource aware scheduling, the compiler flow ensures that both hardware and software elements work harmoniously to achieve target system performance.

The number of MEMs used decreases at very high unroll factors (16 for gaussian Figure 7.5a and harris Figure 7.5b) because our compiler replaces line buffers having fewer than 20 words with register chains. Each design eventually fails to map to our target CGRA when it exceeds 384 PEs or 128 MEMs. Specifically, designs that exceed available resources are shaded in red in Figure 7.5. Based on this investigation, an application designer would use an unroll factor of 16 for gaussian, 3 for harris, 3 for unsharp, 8×8 for resnet, and 8×8 for gemm to fully utilize our target CGRA.

Table 7.6: PE usage comparison before and after compute sharing.

# of PE	No-share	Share
gpyr	16	6
SR-CNN	656	182

Aside from creating parallel hardware, our compiler is capable of sharing the same compute hardware between different operations. This optimization reduces the resource utilization for the certain computation pipeline, either with unbalance workload per stage or too large to fit onto a single CGRA chip.

For instance, in Gaussian Pyramid's processing, images are progressively downsampled as they traverse deeper into the pipeline. Consequently, the compute intensity diminishes by a factor of

Table 7.7: Mem usage comparison before and after compute sharing. The number in parenthesis indicated the extra memory tile used as bank selection controller.

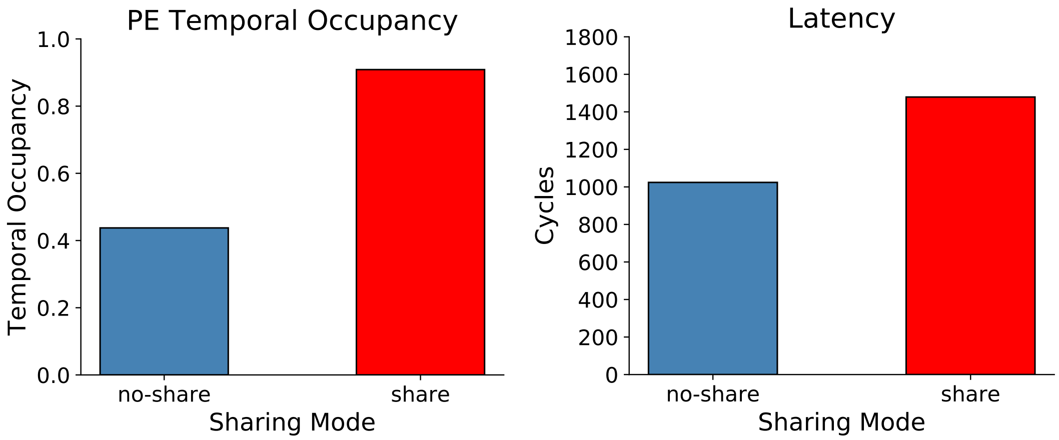
# of Memory	No-share	Share
gpyr	3	12 + (8)
SR-CNN	48	48 + (24)

four with each successive layer. A straightforward approach of assigning a dedicated compute unit for each Gaussian kernel results in a significant reduction in temporal compute occupancy. As depicted in Table 7.6, sharing the same compute hardware across different Gaussian kernels reduces the consumption of processing element (PE) resources. Moreover, sharing the same compute hardware between unbalanced pipeline stages also improve the utilization of the hardware, as shown in Figure 7.6a. Here it nearly doubles the compute occupancy, which is the proportion of time that the PEs are utilized. However, it's important to note that this approach introduces an increase in latency, approximately by 40%, shown in Figure 7.6b, as the computations must be executed sequentially. As for memory utilization comparison, shown in Table 7.7, we saw a slight increase memory usage in the gaussian pyramid application. Because sharing the compute unit changes the schedule and the intermediate storage unit cannot be implemented as line buffer. As a result, these buffers were replaced by four memory banks feeding the 2x2 downsample kernel. Meanwhile there are several memory tile added, which is used to select which memory tile the processing element should access.

SR-CNN[18] is a multi-layer convolutional neural network proposed for image super resolution task. It contains three convolutional layers followed by non-linear ReLu layers. However, a straightforward mapping of each convolution layer onto dedicated hardware would necessitate over 600 processing elements (PEs), as is depicted in the second row of Table 7.6, surpassing our CGRA's resource capacity, which is limited to 384 PEs. To effectively accommodate this DNN on our CGRA, we leverage the compute sharing primitive and share the same convolutional layer compute hardware across three different layers, executing them sequentially. This strategy substantially reduces the PE consumption to under 200 while simultaneously improving the PE's temporal occupancy by 40%. While this approach does lead to a twofold increase in latency and an increase in the number of memories, we have successfully managed to address this concern by sequentially fusing all three layers, thereby converting a previously seemingly impossible task into a feasible implementation on our CGRA.

7.3.2 Memory Resource Aware Scheduling

Machine learning applications demand memory bandwidth to effectively feed the compute units and maintain continuous computational throughput. The challenge lies in orchestrating the on-chip data



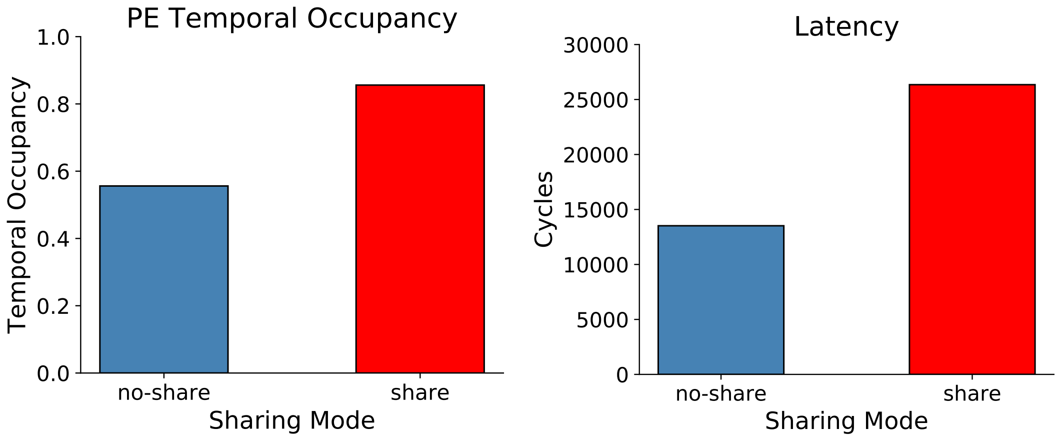
(a) PE Temporal Occupancy

(b) Latency

Figure 7.6: Gaussian Pyramid PE temporal occupancy and latency change after enable compute sharing.

movement to extract optimal performance. This section delves into our successful mapping of all eight ResNet convolutional layers onto the CGRA. These layers exhibit variations in terms of input channels, output channels, image dimensions, and strides. Each layer is properly blocked with 8 input channels and 8 output channels computed in parallel. Our compiler then schedules the execution of each layer using the schedule optimization describe in Section 4.2. As shown in Figure 7.8, we compare three versions of the resource-aware schedules based on compute occupancy, which is the proportion of time that the PEs are utilized. Compared to the sequential scheduling baseline, adding the memory resource aware scheduling which enables double buffering, significantly increases compute occupancy by overlapping data transfers with computation. Applying loop flattening and loop perfection increases the compute occupancy an additional 10%. Notice that this optimization is more effective for the early layers in the DNN where the feature maps have larger spatial sizes. Tiling the width and height of the input feature maps creates an overhead from extra nested tiling loops. The loop optimizations remove this overhead. While the 4-wide memories help with energy-efficiency, we also see that they hold occupancy back by approximately 10% as compared to using a quad-port RAM with single word fetch width. This is the memory implementation that can provide optimal performance regardless of energy efficiency. As mentioned in the prior section, the wide fetch PUB needs to wait for extra cycles while fetching useless data when the data does not align properly in the wide-fetch memories.

In addition to maximizing compute performance, our memory resource-aware scheduler offers the advantage of enabling hardware designers to comprehensively assess the impact of altering the physical memory implementations. This capability proves especially valuable when dealing with common computational patterns, such as matrix multiplication, frequently encountered in machine learning



(a) PE Temporal Occupancy

(b) Latency

Figure 7.7: SR-CNN PE temporal occupancy and latency change after enable compute sharing.

applications. Matrix multiplication involves a multiply and accumulation (MAC) operation, wherein information from multiple input channels is combined into a single output channel, necessitating the execution of four memory operations: initialization, update read, update write, and drain. The most effective implementation will be to assign a dedicated port for each of these operations, leading to a memory design with two read and two write ports. The compiler's awareness of memory capabilities empowers users to explore different hardware designs, including the possibility of mapping partial sum accumulation onto a dual-port memory (with one read and one write port). However, this approach introduces serialization of updates and data transfers, causing computational stalls. As we can see from the Figure 7.9, altering from dual port memory to our PUB with two input and two output port fully exploit the computation potential. It enhances the compute occupancy for the ResNet layers, with an increase of approximate 50%, except for the first layer. The relative minor change observed in the first layer is due to the 7×7 convolution which is applied in this layer. This convolution layer is more compute intensive than the subsequent layers. In this specific computation pattern, the time spent on data loading is negligible compared to the time spent on computation.

7.3.3 Shift Register Optimization

Another optimization we perform is the shift register optimization to reduce memory port requirement, and further reduce memory resources usage. As described in Section 5.1, this optimization involves replacing memories with a small dependency distance between read and write operations or between two different read operations with small delay buffers or registers and wires. Table 7.8 presents a comparison of the number of memories replaced by this optimization in contrast to the naive implementation, which relies on exhaustive banking to ensure sufficient bandwidth. In stencil

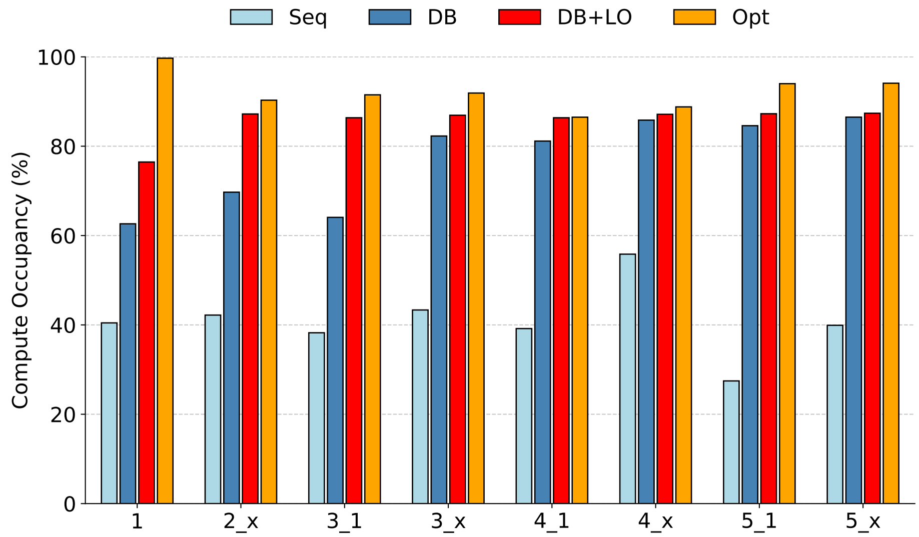


Figure 7.8: Ablation study of the effectiveness of coarse-grained loop optimizations. For ResNet layers, compute occupancy increases as double buffering (DB) and loop optimizations (LO) are added. A quad-port memory leads to even higher compute occupancy with less energy efficiency.

Table 7.8: Shift register optimization replaces memory tiles with registers or wires.

	Original MEMs	MEMs after optimization	Savings (%)	Registers added
gaussian	9	1	89%	6
harris	67	5	93%	30
unsharp	66	6	91%	40
camera	158	25	84%	26
resnet	136	81	40%	0

applications, registers are used to reuse adjacent pixels within stencil windows. In ResNet, data from the same input channel is reused by compute for producing different output channels in parallel. Instead of duplicating input memories, this optimization instantiates a single memory and broadcasts values, enhancing memory efficiency.

7.4 System Level Evaluation

The unified buffer abstraction lets us successfully compile a wide range of applications in Table 7.1 onto a CGRA. Having the same compiler generate code for both CGRA and FPGA enables us to fairly measure the energy efficiency benefit of our CGRA architecture. Although we use the FPGA code generated by our own compiler as the baseline, our FPGA backend is based on Huff's work

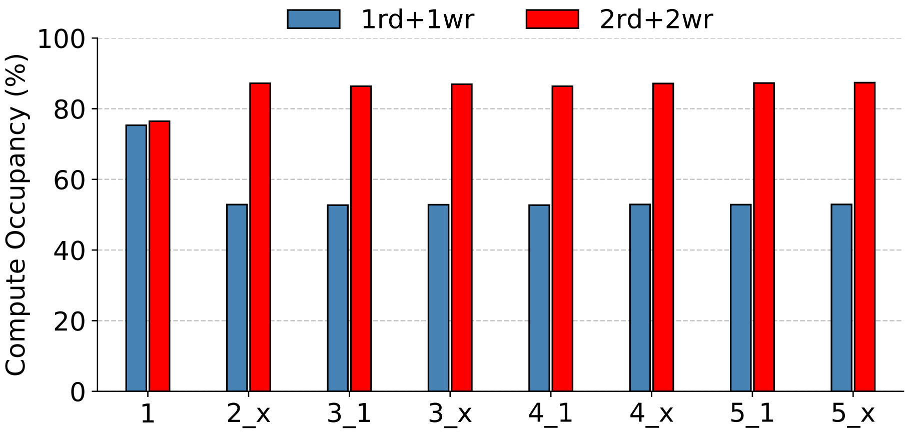


Figure 7.9: Compute occupancy for different memory backends. Blue bars indicate the schedule with dual port memory(1 read 1 write) and red bars indicate the schedule with quad port memory(2 reads and 2 writes).

[38] which demonstrated state-of-the-art performance against the leading FPGA compilers [13].

Figure 7.10 shows the resulting energy/operation consumed. The more efficient physical unified buffer implementation and optimized 16 bit logic mean that the CGRA is 3.5× more efficient than the FPGA. Figure 7.11 shows the applications' time per pixel on the CGRA, FPGA, and a CPU. Time per pixel is the runtime divided by the total number of output values. Our CPU comparison is an Intel Xeon 4214 with 16.5 MB cache with a 2.2 GHz base frequency. We use the same Halide application code for each backend, then validate the output images against each other. The CGRA is able to outperform the CPU, and dominates the FPGA with 4.7× faster runtimes due to its higher clock frequency. With these comparisons, we see that the compiler optimizations coupled with an efficient memory design lead to a competitive accelerator design.

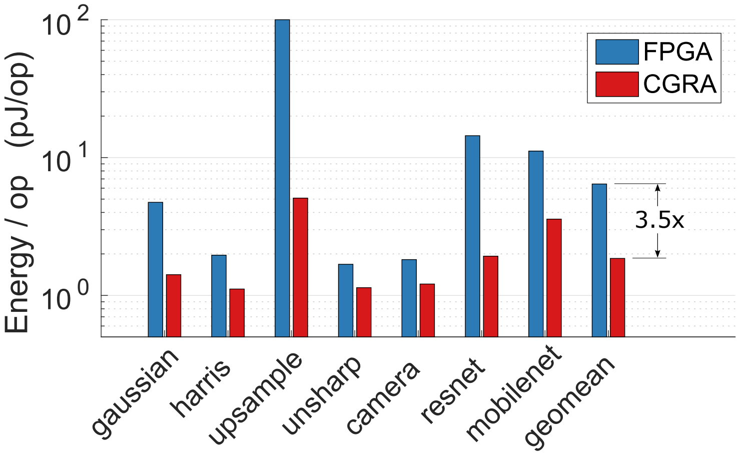


Figure 7.10: Comparison of energy per operation for running kernels on a CGRA and FPGA.

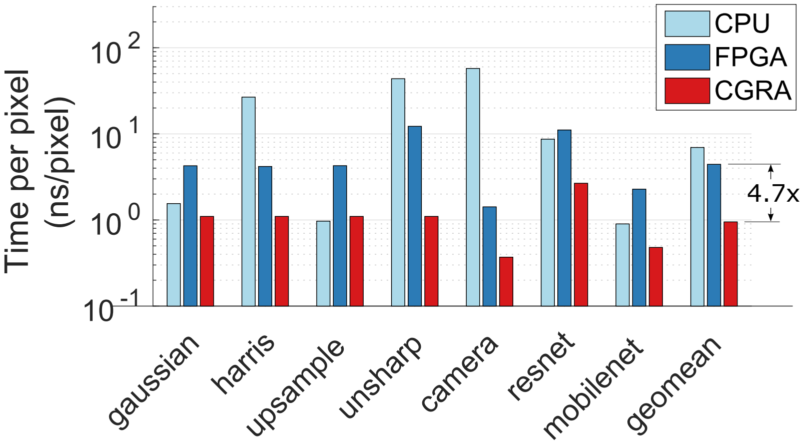


Figure 7.11: Time per pixel on CGRA, FPGA, and CPU.

Chapter 8

Conclusion

The evolving landscape of computing architectures has underscored the increasing significance of accelerators, especially those rely on push memories for optimizing efficiency and performance. Compilers tailored for these hardware architectures have emerged as crucial tools in the pursuit of efficient hardware realization. However these accelerators use push memories which explicitly control all data movement, shifting the responsibility for efficiently managing data movement from the hardware to both the compiler and programmers. This challenge grows when considering the need to effectively utilize massive amount of hardware resources that reconfigurable architectures contain. Previous efforts have often focused on a limited application domain or strongly associated with a specific hardware backend, sometimes lacking the capability to generate code for practical systems in a comprehensive manner. Inspired by the famous quote attributed to Butler Lampson, *all problems in computer science can be solved by another level of indirection*, in this thesis, we have introduced a novel abstraction for push memory, namely, the **unified buffer**, designed to support the application compilation process to reconfigurable accelerators as well as facilitating physical hardware design for reconfigurable architecture.

Through the evaluation of the unified buffer abstraction, this abstraction has proven to be versatile, capable of capturing the access pattern in a broad spectrum of algorithms including both image processing and machine learning. By introducing an intermediate representation, we successfully decouple the software scheduling problem from the intricacies of hardware mapping. This approach automates push memory scheduling optimization through a collection of compiler techniques, including polyhedral analysis and software pipelining, effectively shielding users from the low-level hardware details.

In terms of hardware mapping, our framework empowers the exploration of local reuse opportunities and resource consumption reduction, thereby enabling the creation of highly efficient hardware implementations on reconfigurable architecture. Moreover, the schedule created during the buffer extraction stage exhibits a level of generality, ensuring backend portability, offering support for various hardware backends such as CGRA, FPGA, and the accelerator push memory paradigm, Buffet. Furthermore, this separation of software mapping flow from hardware backend extends support for hardware implementation design space exploration, facilitating the creation of optimized Physical Unified Buffers tailored to CGRA architectures.

With this baseline compiler infrastructure there are several research areas that would further advance the field of compiler technology for reconfigurable accelerators. We currently leverage Halide scheduling primitives to specify the tiling and storage binding for our schedule. All of this information is determined by the experienced user of our system. Further research is needed to automate these tiling and storage level decisions. For instance Interstellar[88] and Timeloop[65] are already making strides in this direction, though they used a brute force search approach in auto-scheduling the mapping of DNN application. There's potential to extend this work to support guided search or machine learning based algorithm for tiling and storage level optimizations as well as supporting fusion would greatly improve the overall system.

While the cycle accurate static schedule adopted in our unified buffer abstraction provides enough information, they can sometimes be overly constrained, particularly in scenarios where the system exhibits greater dynamism, such as push memory systems with ready-valid interfaces. In such systems, the absence of an absolute cycle count means that synchronization occurs in a coarser-grained manner, with only the relative order between operations being significant. Future work can focus on raising the schedule abstraction level. This could involve developing more generic schedule abstraction or mechanisms to annotate dependencies on a higher-level representation, such as a tree structure. By interleaving the granularity of dependencies, compilers can have more flexibility in lowering schedules down to diverse hardware backends.

Currently, we only support compute resource sharing. Memory sharing between multiple UBs poses challenges, particularly due to the PUB architectures on our CGRA using affine controller. To address this, future work can explore ways to extend the concept of memory sharing using piecewise affine controller designs. This extension would enable efficient utilization of shared memory resources among multiple UBs, improving overall resource efficiency and performance on CGRA-based accelerators.

As we look ahead to the post-Moore's Law era[35], it is evident that domain-specific architectures will dominate the computing landscape. To avoid the need for reinventing compilers for each new domain specific architecture, a delicate balance in compiler development must be struck between portability and efficiency. We introduced the unified buffer abstraction to address this compilation challenge and validated its potential by developing a compiler capable of mapping image processing and machine learning applications onto various accelerators. We hope this compiler abstraction not only could be directly applied by other domain-specific architecture compilers but also offers insights into scheduling and mapping applications onto push memory accelerators, paving the way for future advancements in the field.

Bibliography

[1] Amd xlinx, ai engines white paper.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize Halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4), July 2019.

[3] Oguzhan Atak and Abdullah Atalar. Bilrc: An execution triggered coarse grained reconfigurable architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(7):1285–1298, 2013.

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

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 101–113, New York, NY, USA, 2008. Association for Computing Machinery.

[6] Jingwei Cai, Yuchen Wei, Zuotong Wu, Sen Peng, and Kaisheng Ma. Inter-layer scheduling space definition and exploration for tiled accelerators. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[7] 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. In *Proceedings of the ACM/SIGDA International Symposium on*

Field-Programmable Gate Arrays (FPGA), pages 33–36, New York, NY, USA, 2011. Association for Computing Machinery.

[8] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. Amber: A 367 gops, 538 gops/w 16nm soc with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, pages 70–71, 2022.

[9] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: A data-centric approach for mapping deep learning operators on spatial accelerators. *ACM Trans. Archit. Code Optim.*, 19(1), dec 2021.

[10] D.C. Chen and J.M. Rabaey. A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, 1992.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

[12] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(1):127–138, 2016.

[13] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, New York, NY, USA, 2018. IEEE.

[14] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A DSL compiler for accelerating image processing pipelines on FPGAs. In *Proceedings of the International Conference on Parallel Architectures and Compilation*, pages 327–338, New York, NY, USA, 2016. Association for Computing Machinery.

[15] Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236, 2010.

[16] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[17] Alain Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[18] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Image super-resolution using deep convolutional networks. *IEEE transactions on pattern analysis and machine intelligence*, 38(2):295–307, 2015.

[19] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[20] 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. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–422, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Juan Escobedo and Mingjie Lin. Graph-theoretically optimal memory banking for stencil-based computing kernels. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 199–208, New York, NY, USA, 2018. Association for Computing Machinery.

[22] Juan Escobedo and Mingjie Lin. Graph-theoretically optimal memory banking for stencil-based computing kernels. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 199–208, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, pages 365–376, 2011.

[24] Xitian Fan, Di Wu, Wei Cao, Wayne Luk, and Lingli Wang. Stream processing dual-track cgra for object inference. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(6):1098–1111, 2018.

[25] Paul Feautrier. Some efficient solutions to the affine scheduling problem. I. One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, 1992.

[26] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. Tangram: Optimized coarse-grained dataflow for scalable nn accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 807–820, New York, NY, USA, 2019. Association for Computing Machinery.

[27] R. Govindarajan, Erik Altman, and Guang Gao. Theory of modulo-scheduled pipelines. 04 1997.

[28] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A data-movement-aware scheduling language for gpus. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 71–82, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. Graphene: An ir for optimized tensor computations on gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 243–254, New York, NY, USA, 2016. Association for Computing Machinery.

[31] R.W. Hartenstein, A.G. Hirschbiel, M. Riedmuller, K. Schmidt, and M. Weber. A novel asic design approach based on a new machine paradigm. *IEEE Journal of Solid-State Circuits*, 26(7):975–989, 1991.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[33] 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 Transactions on Graphics (TOG)*, 33(4):144:1–144:11, 2014.

[34] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)*, 35(4):85:1–85:11, 2016.

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

[36] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[37] Q. Huang, C. Hong, J. Wawrzynek, M. Subedar, and Y. Shao. Learning a continuous and reconstructible latent space for hardware accelerator design. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 277–287, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[38] Dillon Huff, Steve Dai, and Pat Hanrahan. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 186–194, New York, NY, USA, 2021. IEEE.

[39] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, page 703–718, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Intel Inc. Altera OpenCL. `https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html`, 2022.

[41] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678, New York, NY, USA, 2014. Association for Computing Machinery.

[42] Norman 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. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[43] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 296–311, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Philipp Käsgen, Mohamed Messelka, and Markus Weinhardt. Hiprep: High-performance reconfigurable processor - architecture and compiler. In *International Conference on Field-Programmable Logic and Applications (FPL)*, pages 380–381, Los Alamitos, CA, USA, sep 2021. IEEE Computer Society.

[45] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 242–251, New York, NY, USA, 2019. Association for Computing Machinery.

[46] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7):318–328, jun 1988.

[47] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[48] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 51–57, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Yujun Lin, Mengtian Yang, and Song Han. Naas: Neural accelerator architecture search. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1051–1056, 2021.

[50] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications. *ACM Comput. Surv.*, 52(6), oct 2019.

[51] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators. *ACM Trans. Archit. Code Optim.*, 20(2), mar 2023.

[52] Maxeler Inc. MaxCompiler. `https://www.maxeler.com/products/software/maxcompiler`, 2022.

[53] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

[54] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*, pages 61–70, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[55] Mentor. *Catapult Synthesis User and Reference Manual.* Mentor, Wilsonville, OR, USA, 2019.

[56] Mentor Graphics Inc. Catapult high level synthesis, 2022.

[57] Mirsky and DeHon. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *1996 Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, pages 157–166, New York, NY, USA, 1996. IEEE.

[58] Thierry Moreau, Tianqi Chen, and Luis Ceze. Leveraging the VTA-TVM hardware-software stack for FPGA acceleration of 8-bit ResNet-18 inference. In *Proceedings of the Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning (ReQuEST)*, New York, NY, USA, 2018. Association for Computing Machinery.

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

[60] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Transactions on Graphics*, 35(4), July 2016.

[61] Vivek Nautiyal, Gaurav Singla, Lalit Gupta, Sagar Dwivedi, and Martin Kinkade. An ultra high density pseudo dual-port sram in 16nm finfet process for graphics processors. In *IEEE International System-on-Chip Conference (SOCC)*, pages 12–17, New York, NY, USA, 2017. IEEE.

[62] Chris J. Nicol. A coarse grain reconfigurable array (cgra) for statically scheduled data flow computing. 2017.

[63] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. Stream-dataflow acceleration. *SIGARCH Comput. Archit. News*, 45(2):416–429, June 2017.

[64] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch, and Oliver Sinnen. Exact and practical modulo scheduling for high-level synthesis. *ACM Trans. Reconfigurable Technol. Syst.*, 12(2), may 2019.

[65] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel

Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315, 2019.

[66] Sang Yoon Park and Pramod Kumar Meher. Efficient fpga and asic realizations of a da-based reconfigurable fir digital filter. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61(7):511–515, 2014.

[67] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 137–151, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

[69] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel paterns. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 389–402, New York, NY, USA, 2017. Association for Computing Machinery.

[70] 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 Transactions on Architecture and Code Optimization (TACO)*, 14(3):1–25, 2017.

[71] 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 Notices*, 48(6):519–530, 2013.

[72] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, 1994.

[73] Oliver Reiche, Moritz Schmid, Frank Hannig, Richard Membarth, and Jürgen Teich. Code generation from a domain-specific language for c-based hls of hardware accelerators. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, New York, NY, USA, 2014. IEEE.

[74] Amit Sabne. Xla: Compiling machine learning for peak performance. 2020.

[75] Vivek Sarkar, William Harrod, and Allan E Snavely. Software challenges in extreme scale systems. In *Journal of Physics: Conference Series*, volume 180, page 012045. IOP Publishing, 2009.

[76] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[77] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, New York, NY, USA, 2016. IEEE.

[78] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. AutoDSE: Enabling software programmers to design efficient FPGA accelerators. *ACM Transactions on Design Automation of Electronic Systems*, 27(4), Feb 2022.

[79] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. Ultra-elastic cgras for irregular loop specialization. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 412–425, New York, NY, USA, 2021. IEEE.

[80] Artem Vasilyev. *Evaluating Spatially Programmable Architecture for Imaging and Vision Applications*. PhD thesis, Stanford University, 2019.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[82] Sven Verdoolaege. ISL: An integer set library for the polyhedral model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[83] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA. In *The ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 93–104, New York, NY, USA, 2021. Association for Computing Machinery.

[84] Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium*

on Field-Programmable Gate Arrays, FPGA '14, page 199–208, New York, NY, USA, 2014. Association for Computing Machinery.

[85] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. Memory partitioning for multidimensional arrays in high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[86] Xilinx. *Vivado Design Suite User Guide High-Level Synthesis.* Xilinx, San Jose, CA, USA, 2019.

[87] Xilinx Inc. Vivado high level synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, 2022.

[88] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–383, New York, NY, USA, 2020. Association for Computing Machinery.

[89] Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. Enhanced loop flattening for software pipelining of arbitrary loop nests. Technical report, University of Washington, Seattle, Washington, 2010.

[90] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 161–170, New York, NY, USA, 2015. Association for Computing Machinery.

[91] 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. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 56:1–56:8, New York, NY, USA, 2018. Association for Computing Machinery.

[92] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. SARA: Scaling a reconfigurable dataflow accelerator. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 1041–1054, New York, NY, USA, 2021. IEEE.

[93] Zhiru Zhang and Bin Liu. SDC-based modulo scheduling for pipeline synthesis. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 211–218, New York, NY, USA, 2013. IEEE.

[94] Wei Zuo, Yun Liang, Peng Li, Kyle Rupnow, Deming Chen, and Jason Cong. Improving high level synthesis optimization opportunity through polyhedral transformations. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 9–18, New York, NY, USA, 2013. Association for Computing Machinery.