Title
Enabling Customized Computing in Datacenters: from Accelerator Design to System Integration

Author
Wei, Peng

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Enabling Customized Computing in Datacenters: from Accelerator Design to System Integration

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

by

Peng Wei

2018

ABSTRACT OF THE DISSERTATION

Enabling Customized Computing in Datacenters:
from Accelerator Design to System Integration

by

Peng Wei
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2018
Professor Jason Cong, Chair

CPU-FPGA heterogeneous architectures are attracting ever-increasing attention in an attempt to advance computational capabilities and energy efficiency in today's datacenters. Such architectures provide programmers with the ability to reprogram the FPGAs for flexible acceleration of many workloads. However, this advantage is often overshadowed by two critical issues: 1) the poor programmability of FPGAs and 2) the severe overhead of CPU-FPGA integration. For one thing, the conventional RTL-based FPGA design practice significantly slows down the application development cycle. Although recent advances in high-level synthesis (HLS) have improved the FPGA programmability to some extent, it still leaves programmers facing the challenge of manually identifying the optimal design configuration in a tremendous design space. This challenge thus demands intimate knowledge of hardware intricacies to address and a great deal of effort even for hardware experts. For another, even with a high-quality FPGA accelerator that achieves an orders-of-magnitude performance/watt gain for a computation kernel, such an impressive

gain can often be dramatically offset by the extra CPU-FPGA data communication overhead, resulting in a much reduced system-wide speedup, or even slowdown.

This thesis aims to address these two issues so as to facilitate the adoption of FPGAs in datacenters. To improve the FPGA programmability, we propose a methodology that automates the heavy code reconstruction from software programs towards behavioral descriptions of high-quality FPGA designs, through well-defined architecture templates. Specifically, we propose the composable, parallel and pipeline (CPP) microarchitecture as an accelerator design template. This well-defined architecture template derives high-quality accelerator designs for a broad class of computation kernels, and substantially reduce the overall design space. Also, it enables the introduction of the CPP analytical model that quantifies the performance-resource tradeoffs among different configurations of the CPP template. This in turn leads to fast design space exploration to identify the optimal CPP configuration in a reasonable time. On top of the architecture template and its analytical model, we develop the AutoAccel framework to automatically transform an input computation kernel program into the optimal CPP-based design for it. For general application developers, AutoAccel supplies a nearly push-button experience to produce an FPGA accelerator with good performance; for FPGA design experts, it greatly reduces the effort of manual design space exploration and code reconstruction; it thus substantially improves the FPGA programmability in both cases.

To come up with an efficient CPU-FPGA integration methodology, we first conduct a quantitative analysis on the microarchitectures of state-of-the-art CPU-FPGA platforms, with a key focus on the effective latency and bandwidth of the CPU-FPGA data communication. The analysis results reveal three important factors that affect the efficiency of CPU-FPGA integration: 1) payload size of every data transfer, 2)

the complicated, multi-stage CPU-FPGA data transfer routine, and 3) sharing of the FPGA resource among CPU threads. We then propose three techniques: batch processing, fully-pipelined data communication stack and FPGA-as-a-Service (FaaS) framework, for these three factors, respectively. Batch processing packs small inputs into a large payload; the fully-pipelined stack overlaps various data transfer stages and the compute stage; both improves the data processing throughput. The FaaS framework treats the CPU threads as clients, and the FPGA as the server, and shares the server among the clients via the canonical client-server paradigm. These three techniques form our proposed methodology for efficient CPU-FPGA integration, which is demonstrated through the JVM-FPGA integration process for the genome sequencing application.

The dissertation of Peng Wei is approved.

Glenn Reinman

Sriram Sankararaman

Milos Ercegovac

Jason Cong, Committee Chair

University of California, Los Angeles

2018

To my parents and wife.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

It would not have been possible to complete this dissertation without the support of many kind people around me. It is my pleasure and honor to acknowledge their contributions.

First, I want to express my sincere gratitude to my advisor, Chancellor's Professor Jason Cong, for his profound guidance on how to systematically conduct academic research, constant support to address various research challenges, and valuable suggestions on almost every aspect of my PhD life. I am one of the very few students in the group who sets off the PhD track without any previous research experience. It is Professor Cong that devotes a great deal of effort to helping me build up my individual research capability from ground level. It is my great honor to have Professor Cong as my advisor.

Also, I want to thank Professor Glenn Reinman, Professor Sriram Sankararaman and Professor Milos Ercegovac for their serving on my dissertation committee and constructive comments regarding my dissertation. I would like to express my special thank to Professor Glenn Reinman who provides me with many valuable suggestions on the computer architecture research, which greatly helps my further study on CPU-FPGA platform microarchitectures and the methodology of efficient CPU-FPGA integration.

Many of my colleagues in the Center for Domain-Specific Computing (CDSC) have contributed to the dissertation. It is a great pleasure to work with them throughout these years. I especially thank:

- Yu-Ting Chen, for leading our long-term collaboration in genome sequencing

acceleration project, and invaluable guidance regarding both academia and career.

- Cody Hao Yu and Peipei Zhou, for our collaborations in a variety of research studies, including genome sequencing acceleration, AutoAccel, and CPU-FPGA communication pipeline, many of which form the foundation for this dissertation.

- Zhenman Fang, for his great mentoring support to my early PhD career.

- Yuchen Hao, Sen Li, Tianhe Yu, Licheng Guo, Yuze Chi and Professor Jie Lei, for their great help in our research collaborations and my PhD life.

- Young-kyu Choi and Zhenyuan Ruan, for our collaborations, technical discussions, and being so caring roommates.

- Alexandra Luong, for all the help on various paper work activities, and outstanding service on CDSC reviews and other meeting events.

- Janice Martin-Wheeler, for the hand-editing of all paper manuscripts.

The research studies in this dissertation are partially supported by CDSC under NSF Innovation Transition (InTrans) and from the CDSC industrial partners including Baidu, Fujitsu, Google, Huawei, Intel, NEC and VMWare; the NSF/Intel Partnership on Computer Assisted Programming for Heterogeneous Architectures (CAPA); C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; CRISP, one of the six centers of the Joint University Microelectronics Program (JUMP). I would like to express my gratitude to these sponsors.

Finally, I want to give my thanks to my parents and wife, the most important persons in my life. I cannot go this far without them standing behind me.

VITA

2010	B.S. (Computer Science), Peking University
2013	M.S. (Computer Science), Peking University, Beijing, China
2013−2018	Research Assistant, CDSC, University of California, Los Angeles

PUBLICATIONS

Chen, Y.T., Cong, J., Lei, J. and Wei, P. A novel high-throughput acceleration engine for read alignment. FCCM, 2015.

Chen, Y.T., Cong, J., Li, S., Peto, M., Spellman, P., Wei, P. and Zhou, P. CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing. HITSEQ, 2015.

Choi, Y.K., Cong, J., Fang, Z., Hao, Y., Reinman, G. and Wei, P. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. DAC, 2016.

Chen, Y.T., Cong, J., Fang, Z., Lei, J. and Wei, P. When apache spark meets FPGAs: a case study for next-generation DNA sequencing acceleration. HotCloud 2016.

Cong, J., Wei, P., Yu, C.H. and Zhou, P. Bandwidth optimization through on-chip memory restructuring for HLS. DAC, 2017.

Cong, J., Wei, P., Yu, C.H. and Zhou, P. Latte: Locality Aware Transformation for High-Level Synthesis. FCCM, 2018.

Cong, J., Wei, P., Yu, C.H. and Zhang, P. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. DAC, 2018.

Yu, C.H., Wei, P., Grossman, M., Zhang, P., Sarker, V. and Cong, J. S2FA: an accelerator automation framework for heterogeneous computing in datacenters. DAC, 2018.

Cong, J., Wei, P. and Yu, C.H. From JVM to FPGA: Bridging Abstraction Hierarchy via Optimized Deep Pipelining. HotCloud, 2018.

Cong, J., Guo, L., Huang, P.T., Wei, P. and Yu, T. SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing. FPL, 2018.

Chi, Y., Cong, J., Wei, P. and Zhou, P. SODA: Stencil with Optimized Dataflow Architecture. ICCAD, 2018.

CHAPTER 1

Introduction

Due to power and energy constraints, conventional general-purpose processors are no longer able to sustain the performance and energy improvement in commercial datacenters. To overcome the inefficiency of homogeneous multicore systems, heterogeneous architectures that feature specialized hardware accelerators have been widely considered to be a promising paradigm [CSR11]. In particular, field programmable gate arrays (FPGAs), which offer the potential of orders-of-magnitude performance/watt gains for a broad class of applications while retaining reconfigurability, attract increasing attention as a mainstream acceleration technology. For example, both Microsoft and Baidu have incorporated FPGA-based accelerators in their datacenters to accelerate large-scale production workloads such as search engines [PCC14, CCP16] and neural networks [OLQ14, ORK15]. Amazon also introduced the F1 instance [amab], a compute instance equipped with one or more FPGA boards, in its Elastic Compute Cloud (EC2) [amaa]. Moreover, with the $16.7 billion acquisition of Altera, Intel recently announced the Xeon+FPGA Accelerator Platform [harb], which provides an FPGA and a Xeon processor in a single semiconductor package. Predictions have been made that as much as 30% of datacenter servers will have FPGAs by 2020 [hara]. This suggests that FPGAs could become a common component in future servers and play an important role as primary computing re-

sources [NSS16].

However, a major issue against the adoption of FPGAs in datacenters is the notoriously poor FPGA programmability. FPGA programming is generally recognized as an RTL (register-transfer level) design practice, which requires notable hardware expertise in designing accelerator microarchitectures such as controls, data paths, and finite state machines [Bri12]. This makes the effort of FPGA programming prohibitive to most datacenter application developers. It is even more challenging when the mainstream algorithm in an application domain is constantly evolving; i.e., an algorithm may have already been obsolete during the development process of its hardware accelerator.

Decades of research have focused on improving FPGA programmability. High-level synthesis (HLS) [CLN11] that allows hardware designs to be described in high-level programming languages like C/C++ is recognized as an encouraging approach. Such C/C++ code for hardware designs is generally called the hardware behavioral description. Apparently, this high-level description, compared to its RTL counterpart, is much like the general software program. In fact, many C/C++ programs, without any modification on themselves at all, are already valid hardware behavioral descriptions, and can be compiled by state-of-the-art HLS tools like Xilinx SDAccel [sda] into working FPGA circuits. However, a high-quality software program is generally far away from a high-quality hardware behavioral description due to the lack of proper consideration regarding the underlying FPGA architecture. Our experiments show that a software program, if naively treated as a hardware behavioral description, almost always leads to an FPGA accelerator that performs orders-of-magnitude worse than running the program on a modern CPU (see Section 3). This is because HLS still leaves programmers facing the challenge of manually identifying

the optimal design configuration among a tremendous number of choices, and heavily reconstructing the software code to realize the identified optimal configuration. Addressing this challenge demands intimate knowledge of hardware intricacies, and a great deal of effort even for hardware experts. Consequently, HLS still presents a significant gap between a software program and a high-quality hardware behavioral description, which emerges as a serious impediment against the acceptance of FPGAs by datacenter application developers.

Another critical issue against the adoption of FPGAs in datacenters is the severe overhead incurred by integrating FPGA accelerators into the conventional CPU system. Although standalone FPGA accelerators promise orders-of-magnitude performance/watt gains for a variety of computation kernels, the benefit is often considerably offset by the extra CPU-FPGA data communication overhead. This in turn results in greatly reduced system-wide speedup, or even slowdown [CCF16a, HWY16, PHA17]. For example, our study on the acceleration for the genome sequencing application reveals that although the FPGA accelerator achieves over 100× speedup for the Smith-Waterman computation kernel, a straightforward integration of the accelerator into the software program is going to cause an 1000× slowdown of the entire system [CCF16a]. As a consequence, an efficient CPU-FPGA integration methodology, which is able to truly fulfill the significant gains of FPGA accelerators on computation kernels, is eagerly needed.

This dissertation is devoted to addressing the aforementioned two issues and facilitating the adoption of FPGAs in datacenters. The first objective is to improve the FPGA programmability by paving the path from a software program to a high-quality hardware behavioral description that 1) is functionally equivalent to the software program, and 2) leads to a high-performance FPGA accelerator. To

meet this objective, we first conduct an analysis study to demystify the gap between software programs and hardware behavioral descriptions. Specifically, we start from a collection of benchmarks from MachSuite [RAS14] that consists of the software implementations of a broad class of computation kernels, feed these implementations without any code reconstruction into the Xilinx SDAccel tool to generate their corresponding hardware designs, and analyze the sources of microarchitectural inefficiency in these automatically generated designs. This study reveals three important insights: 1) software programs, which are designed to run on the CPU systems, poorly fit into the FPGA architectures; 2) the accelerators synthesized directly from software programs are trapped into a series of common sources of inefficiency; 3) these sources of inefficiency are able to be substantially resolved by following a best-effort code reconstruction practice of five refinement iterations. These five refinement iterations continuously improve the architecture of the generated accelerators, and finally achieve an 42$\sim$29,030$\times$ speedup.

Inspired by these insights, we propose our automated accelerator generation methodology to automatically transform software programs into high-quality hardware behavioral descriptions. Specifically, we derive from the best-effort code reconstruction practice an accelerator design template, called the composable, parallel, pipeline (CPP) microarchitecture, which provides a clear direction for the transformation. Such a clear direction significantly reduces the design space from "anything possible" to only the scope of CPP. Moreover, the well-defined CPP microarchitecture makes it possible to analytically quantify the performance-resource trade-offs among different design choices. We derive the CPP analytical model to achieve the analytical quantification, which in turn enables fast, analytical-based design space exploration to find the optimal CPP configuration in a reasonable time. Finally, we develop

the AutoAccel framework to automate the entire code transformation process. Our experiments show that AutoAccel-generated accelerators drastically outperform the naive implementations by 27,000×, indicating that our proposed methodology has strongly addressed the gap from software programs towards hardware behavioral descriptions. Meanwhile, the AutoAccel-generated accelerators also outperform the software implementations by 72×, indicating that our approach does lead to high-quality accelerator designs. For general datacenter application developers who may not want to be involved much in FPGA accelerator design, AutoAccel provides a nearly push-button experience to come up with a good design; for hardware experts who is willing to exert customized code reconstruction for better performance, AutoAccel also allows them to feed in the reconstructed code as input, and saves the manual effort in subsequent design space exploration and code transformation. In both cases, the FPGA programmability is substantially improved.

The dissertation then presents our research studies for addressing the second issue: the severe overhead incurred by the CPU-FPGA integration. In particular, we focus on the JVM-FPGA integration process since many state-of-the-art datacenter programming frameworks, e.g., Apache Hadoop [Whi12] and Spark [ZCD12], are based on the Java Virtual Machine (JVM) [LYB14]. We start from a case study in which we attempt to integrate an FPGA accelerator for the Smith-Waterman computation kernel [CCL15a] into CS-BWAMEM [CCL15b], a distributed genome sequencing software program that is based on the Apache Spark framework. The integration study reveals a rather surprising phenomenon that the system-wide performance is degraded by 1000× if we straightforwardly integrate the accelerator with the Spark program (see Section 5.1). Our quantitative analysis reveals that even though the accelerator outperforms its software counterpart by over 100×, it

demands an extra JVM-FPGA data transfer routine to send (receive) data to (from) the FPGA, which consumes 1000× more time than that of executing the kernel code on CPU. Therefore, the JVM-FPGA data communication overhead is the key factor to be blamed.

To acquire a better understanding of the CPU-FPGA data communication, we conduct a quantitative analysis on the microarchitectures of five state-of-the-art CPU-FPGA platforms, including the Alpha Data board [Xil17], Amazon EC2 F1 instance [amab], IBM CAPI [SBJ15], Intel Xeon+FPGA v1 and v2 [harb], with a key focus on their CPU-FPGA communication processes. The analysis results lead to a series of valuable insights for both datacenter application developers to choose the right platform and platform designers to evolve their platforms, and, moreover, suggest three key factors that affect the efficiency of the integration: 1) the payload size of each data communication transaction; 2) the complicated, multi-stage communication routine; 3) the sharing of the FPGA resource among multiple CPU threads. Inspired by these findings, we propose our integration methodology that contains three techniques for these three factors, respectively. First, we propose batch processing, which combines the input data of multiple transactions into one batch to send to the communication channel, to improve the payload size. Also, we propose a fully-pipelined JVM-FPGA data communication stack, which overlaps multiple data transfer stages and the compute stage, to improve the data processing throughput. Finally, we propose the FPGA-as-a-Service (FaaS) framework, which treats the CPU threads as clients and the FPGA accelerator as the server, and share the FPGA among the CPU threads via the canonical client-server framework, to achieve efficient resource sharing. By applying the proposed methodology back to the genome sequencing application, we not only eliminate the 1000× slowdown, but

achieve a 3.5× system-wide performance improvement. This demonstrates the effect of our proposed CPU-FPGA integration methodology in alleviating the data communication overhead.

The remainder of this dissertation is organized as follows. Chapter 2 presents the background and related work, including recent advances of the HLS technology, state-of-the-art CPU-FPGA platforms, the MachSuite benchmark suite and the genome sequencing workload that are used for experimentation, and summaries of previous studies on the improvement for FPGA programmability and CPU-FPGA integration. Chapter 3 presents our best-effort code reconstruction practice to address the gap between software programs and hardware behavioral descriptions. Chapter 4 presents our CPP microarchitecture that is derived from the best-effort practice, its analytical model, and the AutoAccel framework. Chapter 5 presents our methodology for efficient CPU-FPGA integration, including our case study for the genome sequencing acceleration, the quantitative microarchitectural analysis regarding state-of-the-art CPU-FPGA platforms, and the three techniques that we propose to alleviate the data communication overhead. Chapter 6 concludes the dissertation and presents future research directions.

CHAPTER 2

Background and Related Work

This chapter presents the background information and related work for this dissertation. We first introduce recent advances of the high-level synthesis (HLS) technology, with a key focus on to what extent HLS improves the FPGA programmability and what problems HLS still leaves that prevent the programmability from being further enhanced (Section 2.1). Next, we categorize state-of-the-art CPU-FPGA platforms and summarize the microarchitectural features of each category (Section 2.2). We then describe our experimental workloads: the MachSuite benchmark suite and the genome sequencing workload (Section 2.3). Finally, we summarize the related work to this dissertation, including the previous studies on HLS enhancement, genome sequencing acceleration via distributed computing and hardware accelerators, and data communication optimization (Section 2.4).

2.1 The High-Level Synthesis Technology

A field-programmable gate array (FPGA) [BFR12] is an integrated circuit that contains an array of reprogrammable logic and memory blocks: lookup tables (LUTs), flip-flops (FFs), digital signal processing slices (DSPs) and block RAMs (BRAMs). Connected through a hierarchy of reconfigurable interconnects, these blocks can be

customized into different circuits to solve various computation problems. Such hardware customizability allows FPGA circuits to avoid the significant overhead of the general-purpose microprocessors, resulting in orders-of-magnitude performance/watt gains for a broad class of computation kernels.

The notoriously poor programmability of FPGAs, however, emerges as a serious impediment against their acceptance by datacenter application developers. Conventionally, FPGA designers use register-transfer level (RTL) description languages, e.g., Verilog HDL (hardware description language) [TM08], to perform accelerator development for target computation kernels. Such a RTL programming practice has already made FPGA design unwelcome to most datacenter application developers who were mainly trained as software engineers. Besides, compared to simply writing a software program to implement a computation kernel, the RTL design demands a much longer development time even for hardware experts, and thus inevitably extends the time to market. Since many datacenter applications are constantly evolving, an old computation kernel may have already been obsolete before its accelerator is ready to use.

To address this issue, FPGA researchers and vendors start to resort to the HLS technology [CLN11] which uses software programming languages like C/C++ to describe hardware designs and relies on the HLS tools to transform the software-like design descriptions into the RTL descriptions. In fact, commercial HLS tools such as Xilinx SDAccel [sda] and Intel FPGA SDK for OpenCL [int] have been widely used to fast prototype user-defined functionalities expressed in high-level languages on FPGAs without involving RTL descriptions. The example design flow used by common commercial HLS tools is shown in Fig. 2.1. First, a user input program is compiled to the LLVM Intermediate Representation (IR) [llv07], along with the construction

of its control data flow graph (CDFG). Then, the IR-to-HDL code transformation is performed to map the IR to an RTL design with scheduling optimization. This completes the HLS process that maps the behavioral description of a design to its RTL description. Subsequently, the conventional FPGA design automation flow is launched to generate the design's bitstream file that contains the configuration data for FPGA's logic and RAM blocks.

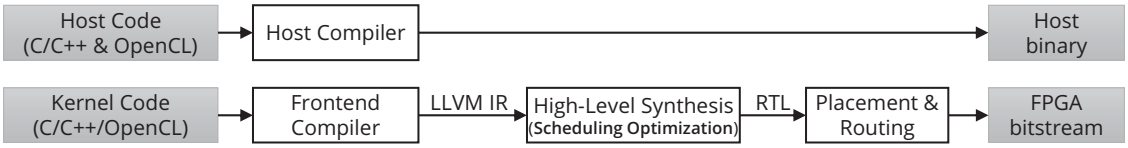


Figure 2.1: Commercial HLS Tool Design Flow

Commercial HLS tools usually have for users a set of language extensions, such as C pragmas, that provide the guidances of memory organization and task scheduling to complement the missing information of static analysis while optimizing the design. The language extensions are specified by the user at the source code level, but the core HLS code transformation and optimization happens at the intermediate representation (IR) level, indicating that the effectiveness of user guidances highly depends on its IR structure and front-end compiler. It implies that two programs with the same functionality but different coding styles (leading to different IR structures) might result in a significant performance difference. In fact, this difference can be up to several orders of magnitude based on our experiences (see Chapter 3). As a consequence, programmers have to pay attention to every detail that may affect the generated IR structure, which often demands a profound understanding of the FPGA architecture and circuit design. This easily leads to an impression that software programs and hardware behavioral descriptions, though looking quite alike,

are completely different, and the gap between them is still prohibitively large.

There has been existing effort that attempts to shorten this gap. The Merlin compiler [mer, CHP16a, CHP16b] is one representative example. The Merlin compiler is a source-to-source code transformation tool that brings FPGA programming to an even higher level than HLS. Specifically, it provides a transformation library that contains a series of code transformation primitives, each of which is associated with a predefined C pragma that abstracts a commonly used code reconstruction strategy, as listed in Table 2.1. As long as the developer inserts a pragma to the right place, the Merlin compiler will apply the corresponding transformation to reconstruct the computation kernel code automatically. As a result, HLS designers who harness the power of Merlin pragmas can save a great deal of labor effort in manually reconstructing their code to implement these primitives everywhere needed.

Table 2.1: Merlin Compiler Code Transformations

Transformation	Target	Parameters	Description
Data tiling	Loop	tilesize=S	Tile the loop and create on-chip buffers to cache the data with size S.
	Example: #pragma Accel data_tiling tilesize=16		
Memory Coalescing	Buffer	bitwidth=b	Pack DRAM buffer to b bits.
	Example: #pragma Accel bitwidth variable=buf factor=512		
Pipeline	Loop	N/A	Create a coarse- or fine-grained pipeline (dataflow).
	Example: #pragma Accel pipeline		
Parallelism	Loop	factor=N	Tile the loop and create N processing elements (PEs).
	Example: #pragma Accel parallel factor=4		

Based on the transformation library, Fig. 2.2 presents the Merlin compiler execution flow. It leverages the ROSE compiler infrastructure [ros00] and polyhedral framework [ZLC13] for abstract syntax tree (AST) analysis and transformation. The front-end stage analyzes the user program and separates host and computation kernel. The kernel code transformation stage then applies multiple code transformations according to user-specified pragmas. Note that the Merlin compiler will perform all necessary code reconstructions to make a transformation effective. For example, when performing loop unrolling, the Merlin compiler not only unrolls a loop but also conducts memory partitioning for the sake of avoiding bank conflict [CJL11]. Finally, the back-end stage takes the transformed kernel and uses the HLS tool to generate the FPGA bitstream.

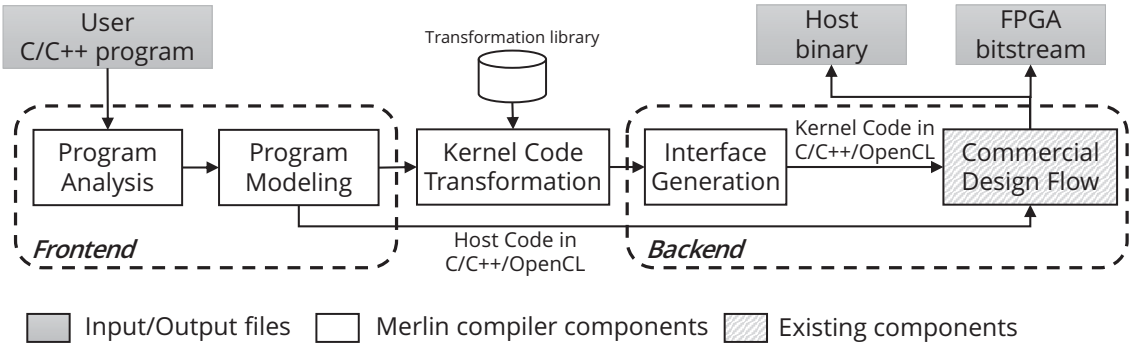


Figure 2.2: The Merlin Compiler Execution Flow

Compared to the standard HLS solution, the Merlin compiler further improves the FPGA programmability by making design optimization "semiautomatic": instead of manually reconstructing the code to make one optimization operation effective, programmers now can simply place a pragma and let the Merlin compiler do the necessary changes. However, it still relies on programmers to determine the optimal combination and parameters of the transformation operations, and thus does not

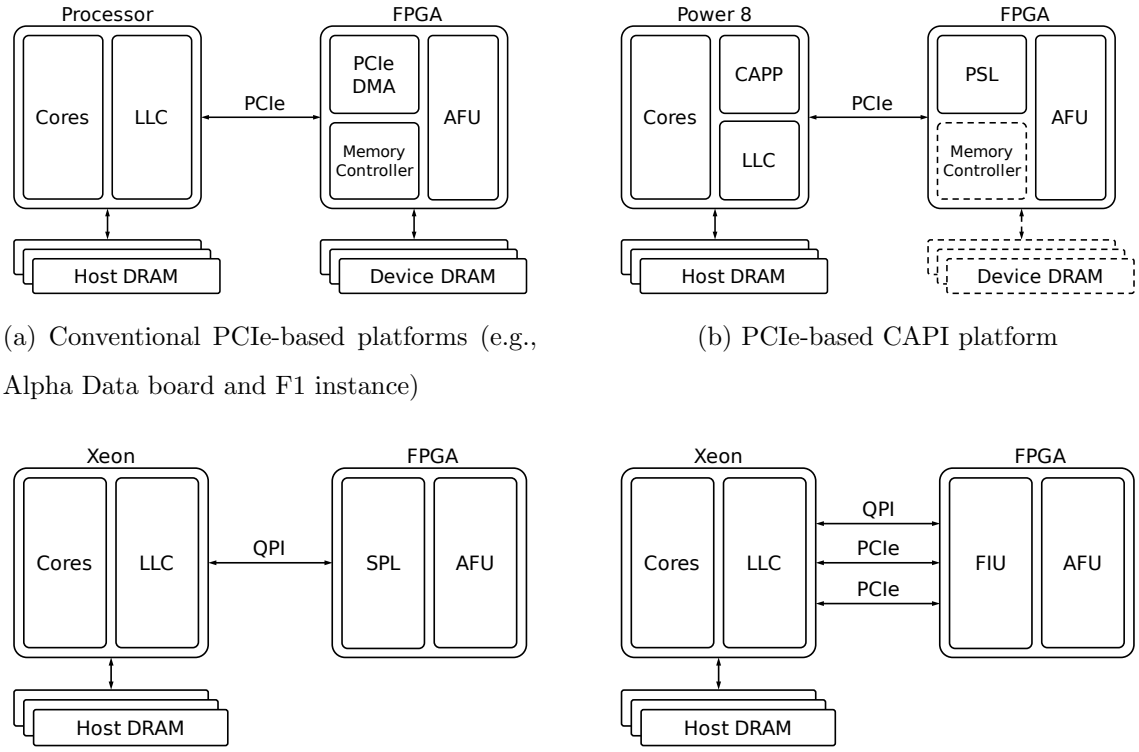
substantially relieve the burden. In this dissertation, we leverage its transformation library to agilely implement our proposed AutoAccel framework (see Chapter 4).

2.2 CPU-FPGA Platforms for Datacenters

A high-performance interconnect between host processor and FPGA is crucial to the overall performance of CPU-FPGA platforms. In this section, we first summarize existing CPU-FPGA architectures with typical PCIe and QPI interconnect. Then we present the private and shared memory models of different platforms.

Typical PCIe-based CPU-FPGA platforms feature Direct Memory Access (DMA) and private device DRAM (Fig. 2.3(a)). To interface with the device DRAM as well as the host-side CPU-attached memory, a memory controller IP and a PCIe end-point with a DMA IP are required to be implemented on the FPGA, in addition to user-defined accelerator function units (AFUs). Fortunately, vendors have provided hard IP solutions to enable efficient data copy and faster development cycles. For example, Xilinx releases device support for the Alpha Data card [Xil17] in the SDAccel development environment [sda]. As a consequence, users can focus on designing application-related AFUs and easily swap them into the device support to build customized CPU-FPGA acceleration platforms.

IBM integrates Coherent Accelerator Processor Interface (CAPI) [SBJ15] into its Power8 and future systems, which provides virtual addressing, cache coherence and virtualization for PCIe-based accelerators (Fig. 2.3(b)). A coherent accelerator processor proxy (CAPP) unit is introduced to the processor to maintain coherence for the off-chip accelerator. Specifically, it maintains the directory of all cache blocks of the accelerator, and is responsible for snooping the CPU bus for cache block



(a) Conventional PCIe-based platforms (e.g., Alpha Data board and F1 instance)

(b) PCIe-based CAPI platform

(c) QPI-based Xeon+FPGA v1

(d) Xeon+FPGA v2 with 1 QPI and 2 PCIe channels

Figure 2.3: A tale of five CPU-FPGA platforms

status and data on behalf of the accelerator. On the FPGA side, IBM also supplies a power service layer (PSL) unit alongside the user AFU. The PSL handles address translation and coherency functions while sending and receiving traffic as native PCIe-DMA packets. With the ability of accessing coherent shared memory of the host core, device DRAM and memory controller become optional for users.

Intel Xeon+FPGA v1 [harb] brings the FPGA one step closer to the processor via QPI where an accelerator hardware module (AHM) occupies the other processor socket in a 2-socket motherboard. By using QPI interconnect, data coherency is

maintained between the last-level cache (LLC) in the processor and the FPGA cache. As shown in Fig. 2.3(c), an Intel QPI IP that contains a 64KB cache is required to handle coherent communication with the processor, and a system protocol layer (SPL) is introduced to provide address translation and request reordering to the user AFU. Specifically, a page table of 1024 entries, each associated with a 2MB page (2GB in total), is implemented in SPL, which will be loaded by the device driver during runtime. Though current addressable memory is limited to 2GB and private high-density memory for FPGA is not supported, this low-latency coherent interconnect has distinct implications for programming models and overall processing models of CPU-FPGA platforms.

Xeon+FPGA v2 co-packages the CPU and FPGA to deliver even higher bandwidth and lower latency than discrete forms. As shown in Fig. 2.3(d), the communication between CPU and FPGA is supported by two PCIe Gen3 x8 and one QPI (UPI in Skylake and later) physical links, which are presented as virtual channels on the user interface. The FPGA logic is divided into two parts: the Intel-provided FPGA interface unit (FIU) and the user AFU. The FIU provides platform capabilities such as unified address space, coherent FPGA cache and partial reconfiguration of user AFU, in addition to implementing interface protocols for the three physical links. Moreover, a memory properties factory for higher level memory services and semantics is supplied to provide a push-button development experience for end-users.

Accelerators with physical addressing effectively adopt a separate address space paradigm (Fig. 2.4). Data shared between the host and device must be allocated in both the host-side CPU-attached memory and the private device DRAM, and explicitly copied between them by the host program. Although copying array-based data structures is straightforward, moving pointer-based data structures such as linked-

lists and trees presents complications. Also, separate address spaces cause data replication, resulting in extra latency and overhead. To mitigate this performance penalty, users usually consolidate data movement into one upfront bulk transfer from the host memory to the device memory. The Alpha Data board and Amazon EC2 F1 instance fall into this category.

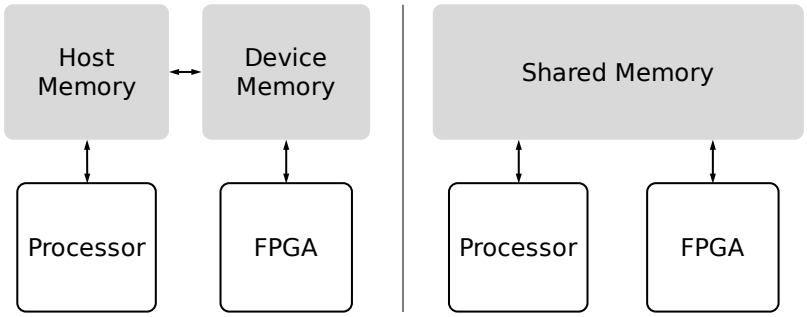


Figure 2.4: Developer view of separate and shared memory spaces

With tighter *logical* CPU-FPGA integration, the ideal case would be to have a unified shared address space between the CPU and FPGA. In this case (Fig. 2.4), instead of allocating two copies in both host and device memories, only a single allocation is necessary. This has a variety of benefits, including the elimination of explicit data copies, pointer semantics and increased performance of fine-grained memory accesses. CAPI enables unified address space through additional hardware module and operating system support. Cacheline-aligned memory spaces allocated using `posix_memalign` are allowed in the host program. Xeon+FPGA v1 provides the convenience of a unified shared address space using pinned host memory, which allows the device to directly access data on that memory location. However, users must rely on special APIs, rather than normal C or C++ allocation (e.g., `malloc/new`), to allocate pinned memory space.

Xeon+FPGA v2 supports both memory models by configuring the supplied mem-

ory properties factory, so that users can decide whether the benefit of having unified address space outweighs the address translation overhead based on their use case.

2.3 Workloads

2.3.1 The MachSuite Benchmark Suite

MachSuite [RAS14] is a benchmark suite that contains a broad class of computation kernels programmed as C functions for accelerator study. For each kernel, MachSuite provides at least one implementation that is based on a commonly used algorithm in software programming, e.g., the queue-based algorithm for the BFS (breadth-first search) kernel. This feature makes MachSuite a natural fit for demonstrating our proposed design automation methodology that aims to facilitate datacenter application developers in transforming software programs into high-quality FPGA accelerator designs in a swift stroke. Table 2.2 lists its computation kernels, each with a brief description about its functionality and input settings.

In this dissertation, the MachSuite benchmark suite is mainly used in Chapter 3 and 4 for the presentation of our automated accelerator generation methodology. Specifically, we demonstrate our best-effort code reconstruction practice via five iterations of refinement for the MachSuite computation kernel accelerators. We then use the benchmarks to evaluate the effectiveness of our proposed CPP microarchitecture and its analytical model, and the AutoAccel framework. MachSuite is also used for the evaluation of our proposed CPU-FPGA data communication pipeline in Section 5.4.

Table 2.2: Benchmark Description

Kernel	Functionality and Input Settings
AES	Advanced encryption standard. Input: 256-bit key; 64MB data.
BFS	Breadth-first search. Input: 4K nodes, 64K edges.
GEMM	General matrix multiplication ($O(N^3)$). Input: two 1024×1024 64-bit FP matrices
KMP	Knuth-Morris-Pratt string matching. Input: 128MB string; 16B substring.
MD	Molecular dynamics. Input: 128×128×128 space, each with 16 molecules on average.
NW	Needleman-Wunsch sequence alignment. Input: 64K pairs of 128-nucleotide seq.
SORT	Merge sort. Input: 64MB integer array.
SPMV	Sparse matrix-vector multiplication. Input: 4096×512 ELLPACK data and index.
VITERBI	Viterbi algorithm. Input: 1M 128-element chains.
FFT	Fast Fourier transform. Input: 65536 strides each with 1KB size.
STENCIL	Stencil computation. Input: a 4096×4096 image

2.3.2 Next-Generation Genome Sequencing

Next-generation sequencing (NGS) [SJ08] results from a combination of chemical engineering and computer science innovations. To sequence a human's entire genome, a number of copies of the individual's genome are fragmented into small pieces, called *reads*. These reads are fed into the chemical sequencer to determine the order of nucleotides for each of them. The sequenced reads are stored as ASCII strings, and aligned to specific locations of a golden reference genome to be assembled into an entire genome sequence. Generally, a sequencing instance processes hundreds of millions of reads, and each read is independently sequenced and aligned.

A typical genome sequencing pipeline contains the following two stages [LH10]. First, a read uses its substrings of various lengths to find candidate alignment positions on the reference genome. This stage is called *seeding*, and the substrings are called *seeds*. Second, each seed is extended leftward and/or rightward to both ends of the read. This stage is called *extending*, and a two-dimension dynamic program-

ming algorithm, the Smith-Waterman algorithm [SW81], is applied to extend the seeds. Each read generates up to hundreds of extending tasks, i.e., up to hundreds of Smith-Waterman kernel invocations.

Various software tools have been proposed for one or more pipeline stages. The Burrows-Wheeler Aligner (BWA) [LD09, LD10, Li13] and Bowtie [LS12] are the most widely used software packages for the seeding and extending stages. The former is mainly used for DNA sequencing applications, like variant discovery, while the latter is mainly used for RNA sequencing applications, such as differential gene expression. The Genome Analysis Toolkit (GATK) [MHB10] contains a set of tools to handle the analysis after alignment. BWA, Bowtie and GATK leverage commodity multicore CPUs to explore the inherent task parallelism of next-generation genome sequencing. However, all of them are single-machine applications, which restricts the degree of parallelism to the number of CPU threads.

Scale-out computing has attracted more and more attention from both academia and industry. Big-data computing frameworks, like MapReduce [DG08, Whi12] and its successor Apache Spark with in-memory computing enhancement [ZCD12], have achieved great success in enabling easy development and deployment of big-data applications in datacenters. Next-generation sequencing has massive inherent parallelism, which makes it a good candidate for cluster scale acceleration. Recently, several cluster scale sequencing tools are being proposed to serve as alternatives of traditional tools like BWA, Bowtie and GATK. In [CCL15b], Chen et al. proposed CS-BWAMEM, a Spark-based MapReduce implementation for the seeding and extending stages. CS-BWAMEM implemented the same algorithm used in the latest

BWA package, and improved the overall performance by 7x with a 25-node cluster.[1] In [MNH13], Massie et al. proposed ADAM, another Spark-based implementation, which provides a set of formats, APIs and tools for genome sequencing and analysis. By redefining the format of genome data to be more cluster-friendly, ADAM reports a 50x speedup on a 100-node cluster over GATK. In this dissertation we focus on the extending stage implemented in CS-BWAMEM, and use it as an application showcase to demonstrate how our proposed JVM-FPGA integration methodology reverses a 1000× system-wide slowdown back to a 3.5× speedup.

The two-dimension dynamic programming algorithm Smith-Waterman [SW81] used in the extending stage of the genome sequencing pipeline is a fundamental operation in computational biology. FPGA acceleration for the Smith-Waterman algorithm has attracted great attention from both academia and industry. Zhang et al. [ZTG07] implemented a systolic array based FPGA accelerator with up to 250x speedup over a 2.2GHz AMD Opteron processor. Kim et al. [Kim11] implemented a dynamic programming string matcher accelerator with 15x speedup over a 16-core CPU server. Olson et al. [OKC12] proposed a scalable FPGA-based solution that is faster than the Bowtie read aligner by 31x. Arram et al. [ATL13] implemented an approximate string matcher on a Maxeler FPGA board, outperforming the CPU-only and GPU-based implementation by 293x and 134x, respectively. Chen et al. [CCL15a] designed a throughput-oriented FPGA accelerator that is dedicated to accelerating the customized Smith-Waterman kernel implemented in BWA-MEM, the latest version of the BWA package, reporting a 26x speedup over a 12-core server. Ahmed et al. [ASH15] also designed an accelerator for the Smith-Waterman

[1]The original BWA package is implemented in C++, re-implementing it in Java leads to a slowdown of around 2x. Therefore, only a 7x speedup is reported for a 25-node cluster.

kernel in BWA-MEM and claimed a 5.7x speedup over a 8-core server.

These proposed accelerators show a great potential for accelerating the Smith-Waterman computational kernel in genome sequencing applications, but lack integration into the full pipeline for large-scale whole genome sequencing. In this dissertation we focus on the integration of one recently-proposed Smith-Waterman accelerator [CCL15a] into the Spark implementation CS-BWAMEM.

2.4 Related Work

This section presents the related work to this dissertation. We first summarize the previous studies on the enhancement of the HLS technology in Section 2.4.1. Next, we present in Section 2.4.2 the studies on the CPU-FPGA integration.

2.4.1 Enhancements to High-Level Synthesis

High-Level Synthesis Automation. Many automated code transformations for the code reconstruction strategies covered in our best-effort practice (Chapter 3) have been proposed using commercial HLS tools like Xilinx SDAccel [sda] and Intel FPGA SDK for OpenCL [Int16], or open-source tools such as LegUp [CCA11] and CHiMPS [PBD08] as a back-end.

For *on-chip data caching*, existing automation strategies mainly focus on analyzing data access patterns, identifying data reuse between loop instances, and then generating on-chip buffers with proper partitions [CZZ12, PZS13, PSK15]. However, most solutions only consider arrays with affine accesses. Automated data caching for an array with arbitrary (non-affine, random, or even both) access patterns is still

an open research problem. For *PE duplication*, the difficulty is that if we duplicate a large computation module, many hardware resources are required and imply less number of PEs. As a result, some researchers deal with this problem by developing algorithms to realize the duplication of a suitable PE granularity under resource constraints [HWB09, CHZ14], but leverage code modularization to users. Therefore, the restrictions on transforming user programs are still necessary. For *pipelining*, the impediments to achieving fine-grained fully pipelining mainly include 1) data/loop-carried dependency, 2) uncertain loop bounds [LBC15, LWC16], and 3) non-affine memory access [VHS15]. Although many researchers have figured out some solutions to each problem, a complete solution is still missing. For *double buffering*, the most widely used application is to form a coarse-grained (nested loop) pipeline [PKB16, LGJ14, TLZ15]. They extract necessary information from the problem using static analysis or user directives and apply predefined templates to form a coarse-grained pipeline using double buffers. Again, those solutions are not yet applicable to arbitrary user programs. On the other hand, there have some advance techniques that highly rely on hardware expertise so we do not cover in Chapter 3. For example, automated unified cache generation on FPGAs is implemented by [PKB16, WH13]. Advanced on-chip memory partition optimizations to avoid data access conflicts for improving pipelining and parallelism are also well-studied [WLZ13, WLC14, CG15, SYZ16].

In summary, although many existing tools and frameworks are able to deal with certain HLS code optimization, simply applying all of them may not achieve high performance. The reason is that many kinds of optimization are related to hardware characteristics and may be affected by coding style a lot, so the order and type of applied optimization form a huge design space which is hard to be explored manu-

ally. However, our best-effort code reconstruction practice provides a direction for automation tool developers to figure out an effective solution of integrating existing HLS optimization to be a comprehensive framework. This is where our AutoAccel automated accelerator generation framework is inspired from.

Domain-Specific Languages. While generating accelerators from generic programming languages presents challenges in discovering parallelism, pipeline structures and memory access, researchers have explored domain-specific languages [SBC] to describe certain patterns and structures using domain knowledge. Lime [ABC10] is a Java-based domain-specific language that provides several parallel patterns to improve the programmability. Bluespec [Arv03] is a functional hardware description language based on Haskell with atomic actions. Chisel [BVR12] embeds hardware construction primitives with Scala and supports high-level abstractions and generators. DHDL [PKB16, KDP16] is an intermediate hardware representation that can be generated from parallel patterns such as map, reduce, zip and filter. This dataflow representation can be used to generate low-level HDL and aid design space exploration. TABLA [MPA16] provides an FPGA accelerator generator for machine learning algorithms that produces synthesizable Verilog code from user model specifications using a set of predesigned templates. Using a similar approach, DNNWeaver [SPA16] targets deep neural models.

Although these tools can provide higher productivity and generate more efficient hardware when applications have certain amenable characteristics, they are often limited to small domains and do not work well for applications outside those domains.

Analytical Modeling. Fast performance estimation on FPGAs has become popular in recent years. In general, performance analysis is mainly performed at either

IR level [WHZ16, ZPW17, PKB16, KDP16, GWC16] or source code level [ZMS16]. The performance analysis at IR level has to compile the source code to the IR and perform analysis by traversing the control flow graph with dynamic profiling; while the source code level analysis is performed by statically looking at high-level constructs such as loops. Since most of the existing work performs analysis without explicitly considering back-end design flow [WHZ16, ZPW17, GWC16, PKB16, KDP16], their analysis cannot reflect the optimization done by the commercial tool. On the other hand, similar to our CPP analytical model, [ZMS16] builds the performance model with the help of the commercial tool, but [ZMS16] provides neither the resource model nor automated code transformation, so users still need to manually change the kernel code while considering the FPGA resource limitation. Although other studies also have the model for different kind of resources [PKB16, KDP16, ZPW17, ZPL16, Zha17, DZZ18], their LUT models are either based on machine learning [PKB16, KDP16, ZPW17, DZZ18] or even missing [ZPL16, Zha17].

2.4.2 CPU-FPGA Platforms and Integration

CPU-FPGA Platform Analysis and Optimization. In addition to the commodity CPU-FPGA integrated platforms summarized in Section 5.2, there is also a large body of academic work that focuses on how to efficiently integrate hardware accelerators into general-purpose processors. Yesil et al. [YOK15] surveyed existing custom accelerators and integration techniques for accelerator-rich systems in the context of data centers, but without a quantitative study as we did. Chandramoorthy et al. [CTI15] examined the performance of different design points including

tightly coupled accelerators (TCAs) and loosely coupled accelerators (LCAs) customized for computer vision applications. Cotat et al. [CMD15] specifically analyzed the integration and interaction of TCAs and LCAs at different levels in the memory hierarchy. CAMEL [CGG13] featured reconfigurable fabric to improve the utilization and longevity of on-chip accelerators. All these studies were done using simulated environments instead of commodity CPU-FPGA platforms.

A number of approaches have been proposed to make accelerators more programmable by supporting shared virtual memory. NVIDIA introduced "unified virtual addressing" beginning with the Fermi architecture [Nvi09]. The Heterogeneous System Architecture Foundation announced heterogeneous Uniform Memory Accesses (hUMA) that will implement the shared address paradigm in future heterogeneous processors [Rog13]. Cong et al. [CFH17] propose supporting address translation using two-level TLBs and host page walk for accelerator-centric architectures. Shared virtual memory support for CPU-FPGA platforms has been explored in CAPI and the Xeon+FPGA family [SBJ15, harb]. This dissertation covers both the separate memory model (Alpha Data and F1 instance) and shared memory model (CAPI, Xeon+FPGA v1 and v2).

There is also numerous work that evaluates modern CPU and GPU microarchitectures. For example, Fang et al. [FMY15] evaluated the memory system microarchitectures on commodity multicore and many-core CPUs. Wong et al. [WPS10] evaluated the microarchitectures on modern GPUs. This work is the first to evaluate the microarchitectures of modern CPU-FPGA platforms with an in-depth analysis.

Integrating Accelerators into Datacenters. There is an increasing trend to integrate FPGA accelerators into modern datacenters. For example, Microsoft has

developed a customized FPGA board, Catapult, and placed it into each server to accelerate the ranking stage of the Bing search engine in a 1632-node cluster [PCC14]. With a key focus on discussing the robust design of the large-scale system architecture, this publication did not reveal many details of the programming framework. Moreover, IBM has proposed the Coherent Accelerator Processor Interface (CAPI) to connect a PCIe-based FPGA board to a POWER8 processor, and integrated such FPGAs into its in-memory data structure store Redis to accelerate its Data Engine for NoSQL [BRH15]. In addition, there are some young start-ups, like Falcon [Sol] and Ryft [Ryf], that are working on the integration of FPGAs with Spark to accelerate big-data analytics. In Chapter 5 we aim to provide a more generalized methodology and insight for efficient integration of FPGA accelerators into state-of-the-art big-data computing frameworks like Spark, and therefore stimulate more innovations in this very hot area.

There are also several academic studies that deploy the Hadoop MapReduce framework or Message Passing Interface (MPI) [GGL99] in an FPGA-based cluster. In [LC13], Lin et al. deployed Hadoop in a cluster of low-end Xilinx Zynq FPGA SoC boards to accelerate a standard FIR filter. Similarly, in [NMG15], Neshatpour et al. deployed Hadoop in a Zynq-based cluster to accelerate machine learning kernels. In [MK15], Moorthy et al. deployed both OpenMPI and MPICH in a Zynq-based cluster to accelerate graph processing applications. In [TL10], Tsoi et al. proposed a heterogeneous cluster with GPUs and FPGAs, and deployed OpenMPI to accelerate N-body simulation. Our application showcase focuses on integrating FPGA accelerators into the Spark MapReduce framework due to its popularity and ease of programming and deployment for big-data applications.

Meanwhile, there are also some efforts that integrate GPU accelerators into

Hadoop and Spark. For example, in [GBS15], Grossman et al. proposed an automated flow to generate OpenCL kernels for Hadoop programs in a GPU-equipped cluster. In [LL15], Li et al. integrated GPU accelerators with Spark for deep learning algorithms. While these approaches usually target the integration of coarse-grained accelerators, we mainly focus on the integration of fine-grained FPGA accelerators, which introduces more challenges, like efficient communication and sharing.

A recent study in [SCN15] tried to automatically generate the OpenCL accelerator code from original Java code in Spark, which is an interesting and challenging direction for future work. Unfortunately it did not report any performance data. Finally, some prior efforts [YTT08, SWY10] also tried to propose a new MapReduce framework for easy accelerator development in a single-node FPGA-based platform.

CHAPTER 3

Best-Effort Code Reconstruction Guidelines for Microarchitecture Optimization

This chapter presents our analysis study that aims to demystify the gap between software programs and hardware behavior descriptions[1]. We find from this study that a best-effort code reconstruction practice with five refinement strategies is able to effectively fill this gap and lead to high-quality accelerator designs. We then are inspired to derive this best-effort practice into the composable, parallel, pipeline (CPP) microarchitecture and propose our AutoAccel framework on top of it, which is presented in Chapter 4.

3.1 Overview

Decades of research has been focusing on improving FPGA programmability. High-Level Synthesis (HLS) [CLN11] can derive high-quality accelerator designs directly from high-level behavioral descriptions, saving programmers from extensive hand-coding in RTL and manual tuning. State-of-the-art HLS tools such as the Xilinx SDAccel [sda] and Intel FPGA SDK for OpenCL [int] allow computational kernels

[1]This study is presented in [CFH18, CWY17]. I would like to convey my appreciation to all coauthors for their contributions to this study.

to be described in C/C++ and OpenCL, which can then be compiled and synthesized into FPGA accelerators. Using these tools, programmers can easily write synthesizable code or convert existing software implementations into FPGA accelerators. On top of the HLS compilation flows, FPGA vendors further provide system-level development environments that integrate a collection of intellectual properties (IPs), drivers, libraries and APIs. This facilitates faster integration of FPGA accelerators into CPU programs, frees developers from system-level integration such as external memory interfacing and CPU-FPGA data communication. As a result, a software programmer can extract targeted computational kernels from C programs (hosts), connect kernels to hosts via IDE-provided APIs, and feed both to the IDEs to infer FPGA accelerators from the kernels and integrate them into the hosts automatically. This demonstrates a quantum leap on FPGA programmability towards software programming, compared to the register-transfer level IC design.

However, a simple push-button process is far from producing high-performance FPGA accelerators. To demonstrate this, we compare the performance of a single-thread CPU with FPGA accelerators that are directly generated from the same software code of benchmarks from MachSuite. Unfortunately, these naive FPGA accelerators are more than 200x slower than the original software implementations running on a Xeon CPU core, which defeats our purpose of accelerating these computational kernels in the first place (see Section 3.3). To improve the quality of HLS-generated accelerators, many prior studies [HWB09, CZZ12, WH13, WLZ13, PZS13, CHZ14, LGJ14, LBC15, PSK15, VHS15, CG15, TLZ15, PKB16, LWC16, SYZ16] focus on proposing enhancements to HLS languages to express certain hardware structures. Nonetheless, most studies require the understanding of hardware intricacies in order for programmers to direct HLS tools to generate the right hardware structure. Also,

choosing the right combination of optimization strategies among an exponential set of candidates is non-trivial even for experienced accelerator designers [WHZ16]. As a result, programmers often get the impression that software programs and hardware behavioral descriptions, though looking alike, are complete different.

In this study we aim to address the following questions: 1) what are the impediments that prevent software programs from being high-quality hardware behavioral descriptions? (2) Is there a common code reconstruction practice that is able to transform a broad class of computation kernel programs into high-quality behavioral descriptions? To answer these questions, we attempt to apply various HLS optimizations to the ported computational kernels from MachSuite. We are encouraged by the fact that the naively generated accelerators fall into a set of common sources of inefficiency, and a best-effort code reconstruction practice can produce quite compelling results: improving the accelerator performance by 42∼29,030× over the naive baseline, and outperforming a Xeon CPU core by 34.4× on average. Specifically, we use data-driven refinement to iteratively optimize the accelerator design: in each refinement iteration, we pinpoint the performance bottleneck and apply a set of HLS optimizations listed in Table 3.1. These HLS optimizations include explicit data caching through batch processing and data tiling, customized pipelining, processing element (PE) replication, double buffering and scratchpad reorganization. They take effect by constantly improving the underlying microarchitecture of the accelerators, which inspires us to derive the best-effort practice into the CPP microarchitecture and propose the entire automated accelerator generation methodology.

Table 3.1: Summary of optimization strategies.

HLS Optimization	Counterpart in Soft. Programming	Speedup	Example
Explicit Data Caching	Data Tiling	5.6∼32.1x	Fig. 3.4(a)
Customized Pipelining	Directive-Based Programming	1.3∼10.3x	Fig. 3.4(b)
PE Replication	Multithreading	1.0∼53.6x	Fig. 3.4(b)
Double Buffering	Comp./Comm. Overlapping	1.0∼2.1x	Fig. 3.4(c)
Scratchpad Reorganization	Bit Packing	1.1∼19.1x	Fig. 3.4(d)

3.2 Experimental Setup

In this study we focus on the currently more accessible PCIe-based CPU-FPGA platform and HLS design flow to demonstrate the code reconstruction practice. Table 3.2 lists the detailed hardware and software configuration. A Xeon CPU is connected with an Xilinx Virtex-7 FPGA board through the PCIe interface. For a fair comparison, both the CPU and the FPGA fabric were launched in 2012. On top of the platform hardware, we use Xilinx SDAccel to provide a hardware-software co-design environment.

This study presents the code reconstruction practice through a complete accelerator design demonstration on a collection of benchmarks in MachSuite [RAS14]. Starting from the accelerators synthesized directly from the MachSuite kernel functions, Section 3.3, 3.4 and 3.5 present the five refinement strategies in the entire accelerator refinement process. The acceleration for the SORT (merge sort) kernel

Table 3.2: Configuration of hardware and software.

Host CPU Model	Intel Xeon E5-2420 @ 1.9GHz (released in 2012)
Host Memory	64GB DDR3-1600
FPGA Fabric	Xilinx Virtex-7 @ 200MHz (released in 2012)
Device Memory	16GB DDR3-1600 (Max Band.: 12.8GB/s)
CPU-FPGA Interface	PCIe Gen3 x8 (Max Band.: 8GB/s)
Synthesis Environment	SDAccel 2017.1

is relatively different from that of the others. Merge sort has a tree-reduce characteristic, which means that the degree of parallelism will decrease by 2× after each merge layer. The last few layers have very limited parallelism and are hard to be accelerated by FPGAs which heavily rely on parallelism to outperform CPUs. A common practice to resolve this issue is to let the FPGA accelerator focus on the first few layers and let the CPU do the remainder. We adopt this approach and set the goal of the SORT kernel to making every 1MB data chunk sorted.

3.3 Strategy #1: Explicit Data Caching

We start from the naive FPGA accelerators directly synthesized from the original MachSuite kernel functions, which are slower than the Xeon CPU by 70∼765×. Nonetheless, programmers are able to improve the performance of the accelerators by 5.6∼32.1× through the code reconstruction that realizes explicit data caching. Section 3.3.1 pinpoints the performance bottleneck in the naive baseline and analyzes the underlying reason. Section 3.3.2 presents the use of explicit data caching to resolve this.

```
void aes(...) { ... }
void kernel(char* data, int size) {
#pragma HLS interface m_axi port=data
  for (int i=0; i<size; i++) {
    aes(data+i*16);
}}
```

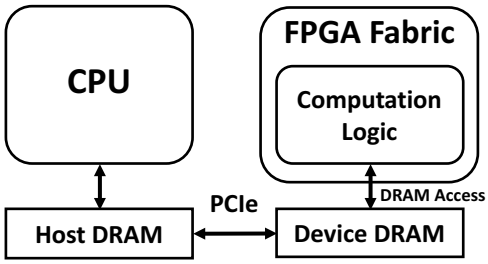


Figure 3.1: Example AES kernel and naively generated architecture.

To better demonstrate the refinement process, we use the AES (advanced encryption standard) kernel as an example to deliver the code implementation of each step. Fig. 3.1 shows the baseline AES kernel code (we ignore the key in this example to simplify the description). The *kernel* function accepts a certain **size** of **data**, and iteratively calls *aes* function to encrypt the **data**. Each *aes* function call encrypts a 128-bit data block, so the **data** pointer shifts by 16 bytes after each iteration. The *interface* pragmas in the *kernel* function specify the interface between the host program and the accelerator kernel. The **data** are transferred from the host to the device through PCIe, and stored in the device DRAM.

3.3.1 Cache: Not a Free Lunch Any More

Fig. 3.2 presents the execution time breakdown of the FPGA accelerators for the MachSuite kernels before any refinement. It suggests that the DRAM access is dominating the overall execution for every kernel. This is due to the fact that the cache hierarchy in CPUs, which provides a memory subsystem with low access latency while retaining programmer transparency, does not exist on FPGAs. In contrast, FPGAs' on-chip BRAMs (block RAMs) that serve as the counterpart to caches are conceptually scratchpads, and have to be explicitly manipulated by software

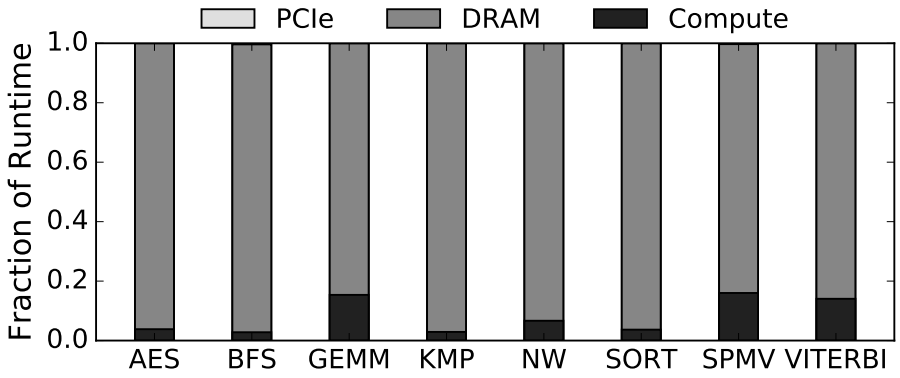


Figure 3.2: Execution time breakdown before any refinement.

programs to realize data caching. With such manipulation missed in the kernel function, the generated accelerator will connect the computation logic directly with DRAM with no high-speed data caching component in-between, as illustrated in Fig. 3.1. Every data access has to physically go off chip, which costs a per-access initialization overhead of approximately 100 FPGA cycles (i.e., 500ns).

In summary, the programmer-transparent cache memory system is not a free lunch any more for HLS-based FPGA accelerator programming. In order to continue harnessing data caching to alleviate the DRAM access overhead, programmers must reconstruct the computation kernel code to explicitly cache data into the FPGA on-chip memory.

3.3.2 Batch Processing and Data Tiling

We present two techniques to implement explicit data caching: batch processing and data tiling. Batch processing, as its name hints, batches a number of jobs together and processes them in one action. This approach is used for computational kernels whose working set sizes are far less than the total size of on-chip BRAM. We use the

AES kernel as an example to explain this approach. As introduced in the beginning of Section 3.3, an AES job, i.e., an *aes* function call, encrypts only a 128-bit data block. Since the working set size of an AES job is much smaller than the size of on-chip BRAM (a few MBs), programmers can maximize data reuse, i.e., temporal locality, by caching and processing one 128-bit data block at a time. However, fetching 128-bit data at a time still leads to a serious DRAM access overhead because of the 100-cycle per-access initialization overhead. One alternative is to process multiple contiguous 128-bit data encryption jobs in one batch. With batch processing, multiple DRAM data fetches are combined into one memory burst operation, which spends 100 cycles in initialization and approximately 1 cycle (5ns) in fetching each piece of data. The DRAM access overhead is then amortized.

We now present data tiling which first divides a job into a set of subjobs and then processes one or a few subjobs at a time. This approach is used for computational kernels with relatively large working set sizes that are close to or far larger than the total size of on-chip BRAM. We use the GEMM (general matrix multiplication) kernel as an example to explain the approach. The GEMM kernel calculates the product of two matrices. While the matrices may be too large to be fully cached in BRAM, the matrix multiplication can be divided into additions and multiplications of submatrices, each of which can be as small as 1x1. Programmers can then process one or more subjobs at a time to explore the temporal locality in the subjob level.

An important design choice is the caching size. The experimental platform used in this study supplies approximately 4MB BRAM for FPGA accelerators (and the other few MBs for system-level IPs). While a larger caching size that enables larger memory burst length is always beneficial in amortizing the 100-cycle initialization overhead, the effect of this amortization diminishes as the burst length increases.

Theoretically, if the payload size of a memory burst reaches 64KB, the burst length will be over 1000, and the impact of the initialization overhead will be reduced to less than 10%. This indicates that explicit data caching can be realized with a very small BRAM consumption.

Fig. 3.4(a) illustrates the implementation of explicit data caching through a code reconstruction of the AES baseline. We can see that the overall execution of the AES kernel is decoupled into a series of *load-compute-store* iterations. Programmers only need to declare local arrays that represent on-chip BRAM buffers, and iteratively *load* input the data of a batch of jobs (batch processing) or one or few subjobs (data tiling) to the arrays to *compute*, and *store* output data back to DRAM. The "memcpy" operations for loading/storing data will be inferred into memory read/write bursts to reduce the average DRAM access latency. This code reconstruction leads to the addition of an intermediate BRAM layer between the computation logic and DRAM, as illustrated in Fig. 3.5(a). Instead of letting the computation logic retrieve data directly from DRAM, this on-chip BRAM layer caches all necessary data for each iteration of computation.

Fig. 3.3 shows the normalized speedups of the accelerators compared to the Xeon CPU core after applying explicit data caching. It also compares the performances of the accelerators with various caching sizes (the SORT kernel targets the sorting of each 1MB data chunk, so the caching size is set at 1MB only). Each "infinite" bar delivers a speedup estimation where the caching size is infinite, so that no initial overhead of memory bursts is counted. Two insights are revealed from the data. First, explicit data caching results in a significant performance improvement over the naive baseline. Second, the caching size has a negligible performance impact. On one hand, the performances between the 64KB, 1MB and "infinite" groups are

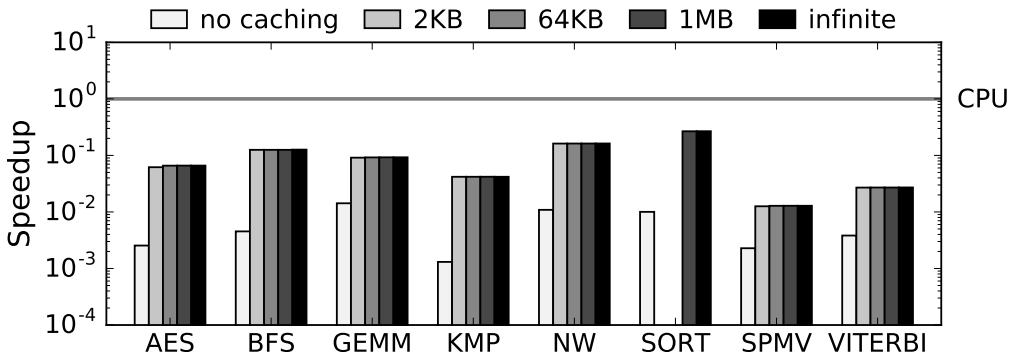


Figure 3.3: Normalized speedups in different caching sizes.

almost identical, which is consistent with our previous analysis on the design choice of caching size. On the other hand, although there might be some performance differences between the 64KB and 2KB (close to the size of one BRAM block) groups, we observe that after explicit data caching is applied, the performance is dominated by computation (see Section 3.4). Thus, the performances between the 2KB and 64KB groups are also very similar. This suggests that programmers can always consider shrinking down the caching size from the maximum (∼4MB) to 1MB or 64KB to spare the BRAM resources for other optimization strategies.

3.4 Go Parallel

We now start from the accelerators that have been applied explicit data caching to. Fig. 3.6 presents the execution time breakdown of the accelerators. As the data hints, computation is dominating the overall execution for every kernel. The major reason is that the accelerators are doing computation sequentially with the culprit: a 200MHz clock frequency that is 9.5× lower than that of the Xeon CPU. Consequently, FPGA accelerators heavily rely on exploring parallelism to dissolve the frequency

```
void aes(...) { ... }
void load(char *buf, char *in) {
  memcpy(buf, in, BATCH_SIZE); }
void store(char *out, char *buf) {
  memcpy(out, buf, BATCH_SIZE); }
void compute(char *buf_data) {
  for (int i=0; i<BATCH_SIZE; i+=16) {
    aes(buf_data+i*16); }}
void kernel(char *data, int size) {
  char buf_data[BATCH_SIZE];
  int batch_num = size/BATCH_SIZE;
  for (int i=0; i<batch_num; i++) {
    load(buf_data, data+i*BATCH_SIZE);
    compute(buf_data);
    store(data+i*BATCH_SIZE, buf_data);
}}
```

(a) Applying explicit data caching

```
int PE_BATCH = BATCH_SIZE / PE_NUM;
void aes(char *data) {
  for (...i...) {
#pragma HLS pipeline
}}
void load(...) { ... }
void store(...) { ... }
void compute(char *buf_data) {
  for (int j=0; j<PE_NUM; j++) {
#pragma HLS unroll
    for (int i=0; i<PE_BATCH ; i+=16)
      aes(buf_data[j]+i*16); }}
void kernel(char *data, int size) {
  char buf_data[PE_NUM][PE_BATCH];
#pragma HLS array_partition var=buf_data dim=1
  ... }
```

(b) Applying pipelining and PE replication

```
void aes(...) { ... }
void load(...) { ... }
void store(...) { ... }
void compute(...) { ... }
void kernel(char *data, int size) {
  char buf_data[3][PE_NUM][PE_BATCH];
#pragma HLS array_partition var=buf_data dim=1
#pragma HLS array_partition var=buf_data dim=2
  for (int i=0; i < size/BATCH_SIZE; i++) {
    switch (i % 3) {
      case 0:
        load(buf_data[0], data+i*BATCH_SIZE);
        compute(buf_data[1]);
        store(data+i*BATCH_SIZE, buf_data[2]);
        break;
      case 1: ...; break;
      case 2: ...; break; }
}}
```

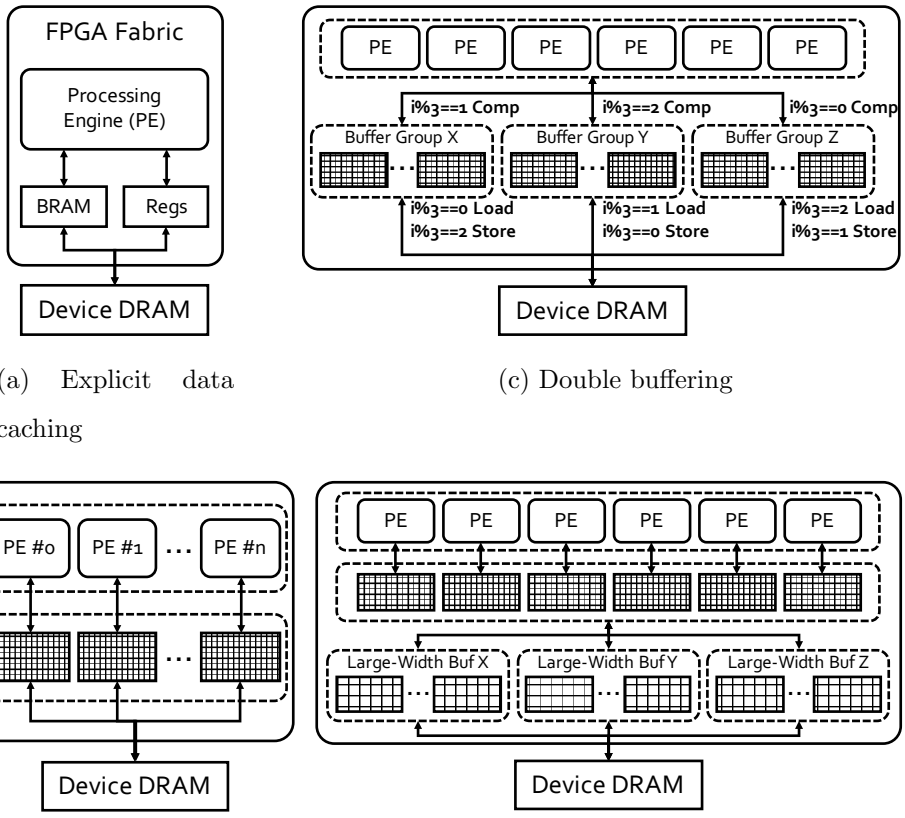
(c) Applying double buffering

```
void aes(...) { ... }
void load(...) { ... }
void store(...) { ... }
void compute(ap_uint<W> large_buf[][PE_BATCH])
{ char normal_buf[BATCH_SIZE];
#pragma HLS array_partition var=normal_buf
  for (int j=0; j<PE_NUM; j++) {
#pragma HLS unroll
    memcpy(...); // copy in
    ... // parallel compute
    memcpy(...); }} // copy out
void kernel(ap_uint<W> *data, int size) {
  ap_uint<W> buf_data[3][PE_NUM][PE_BATCH];
  ... }
```

(d) Applying scratchpad reorganization

Figure 3.4: Step-by-step example of applying the five optimization strategies to the AES benchmark. The complete code after applying all strategies is shown in List 3.2.

disadvantage. Section 3.4.1 and 3.4.2 present the use of two principal parallelism exploration strategies to reduce the computation time: customized pipelining and processing element (PE) replication. These code reconstruction strategies further improve the performance of the MachSuite accelerators by up to 417.8×.



(a) Explicit data caching

(c) Double buffering

(b) PE Replication

(d) Scratchpad Reorganization

Figure 3.5: High-level architecture diagram, corresponding to Figure 3.4.

3.4.1 Strategy #2: Customized Pipelining

Pipelining is a fundamental concept in computer science and is used by both CPUs and accelerators. Given no pipeline stalls, a pipeline with N stages can boost the

throughput by N times. Although various events, including branch misprediction and cache/TLB misses, impede the depth of a CPU pipeline to increase perpetually, FPGA accelerator designers can customize very deep pipelines for pure computational units with hundreds or even thousands of stages to greatly improve the accelerator performance. Fine-grained pipelining to meet the maximum achievable data transfer speed almost becomes a de facto standard for FPGA accelerator designers.

Customizing a full pipeline with an initiation interval (II) equal to 1 (i.e., the pipeline can process one iteration of data every cycle) is difficult in many cases even for hardware experts. However, a best-effort code reconstruction, i.e., simply adding a "pipeline" pragma to a loop block, can often lead to impressive performance improvement. We evaluate all 40 loop blocks in all benchmarks, and find that 27 loop blocks can be immediately pipelined, and 6 loop blocks can be pipelined if being reconstructed into perfect loops. Moreover, a significant speedup on the computation is observed for many kernels, as listed in Table 3.3. Some kernels such as SPMV, NW, GEMM and KMP reach a 7.0∼10.9× speedup, because the main bodies of these kernels are nested loop blocks. SPMV and GEMM do linear algebra in a two-level and three-level nested loop, respectively; NW does a two-dimension dynamic programming in a two-level nested loop; KMP matches a substring in a string in a two-level nested loop. These loop bodies are well pipelined. Other kernels such as AES, BFS and SORT have relatively complicated kernel function bodies, and reach relatively moderate speedup—40%∼80% performance improvement. VITERBI is a delicate case. Although it also does a dynamic programming in a nested loop body, it requires each pipeline stage to complete a few *floating-point* additions, multiplications and comparisons (subtractions), which results in a pipeline with a relatively large II. In contrast, the NW kernel, with a similar computation pattern, requires each

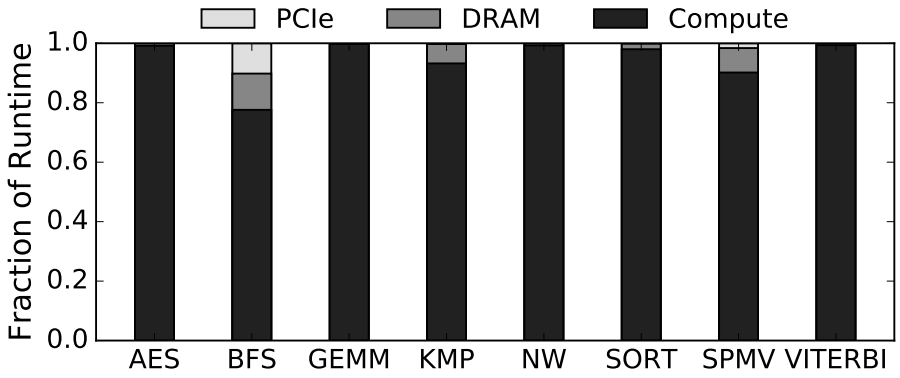


Figure 3.6: Execution time breakdown before exploring parallelism.

Table 3.3: Performance speedup of pipelining on computation.

Kernel	Speedup	Kernel	Speedup	Kernel	Speedup
AES	1.4x	BFS	1.4x	GEMM	10.5x
KMP	7.0x	NW	8.8x	SORT	1.8x
SPMV	10.9x	VITERBI	3.2x		

pipeline stage to complete merely a few *low-width integer* additions and *bit-level* comparisons, which can be finished in one cycle, i.e., achieving an II=1 pipeline. Therefore, the speedup for VITERBI (3.2×) is fairly less than that for NW (8.8×), but still considerable.

3.4.2 Strategy #3: Processing Element Replication

Processing element (PE) replication explores the task-level parallelism in kernels. If a large number of independent jobs can be found in a kernel, then programmers can create multiple PEs to process them in parallel. This concept is not new to software programmers. With multicore becoming ubiquitous in modern processors,

programmers have been accustomed to mapping independent jobs onto multiple cores and do them in parallel. Here, FPGA PEs and CPU cores are counterparts. As a consequence, the implementation of PE replication can be considered as a special "multithreading programming".

Fig. 3.4(b) illustrates the implementation of customized pipelining and PE replication in one code example since the two strategies work on mutually exclusive code regions. This code example is updated from the one in Fig. 3.4(a), which implements explicit data caching. We intentionally omit the implementation details that remain unchanged so as to highlight the newly added code regions. This update features three major changes. First, the "pipeline" pragma is added into each loop block of the *aes* function to perform customized pipelining. Second, the "unroll" pragma is adopted to generate multiple PE replicas for parallel computation. In addition, the "memory partition" pragma is used to partition the local arrays used for explicit data caching into multiple segments, the number of which is equal to the PE replication factor. The first change pipelines the loop blocks, and the latter two changes together realize PE replication. Figure 3.5(b) illustrates the refined architecture after applying pipelining and PE replication. Compared to Figure 3.5(a), the refined architecture partitions the intermediate BRAM layer into multiple groups, each of which communicates only with one PE replica.

One thing worthwhile mentioning here is memory partitioning, whose objective is to realize parallel data supply for all PE replicas. FPGA is a kind of reconfigurable logic that contains distributed computation building blocks: LUTs (lookup tables) and DSPs (digital signal processors), and BRAM blocks (distributed on-chip memory building blocks). The computation building blocks enable the creation of multiple PE replicas, and the BRAM blocks supply a many-port on-chip memory system.

Specifically, the Xilinx Virtex-7 FPGA fabric has approximately 3000 BRAM blocks, i.e., a virtually 3000-port on-chip memory system which has the potential to feed up to 3000 PEs concurrently. To fulfill this potential, however, a programmer needs to partition the local array, i.e., the on-chip BRAM buffer for data caching, into multiple segments, each made up of a set of BRAM blocks. When input data are loaded into the BRAM buffer, they will be scattered into different segments and processed by different PEs simultaneously.

Fig. 3.7 compares the performance improvement on computation with various PE replication factors.[2] For each kernel, we normalized all the performances to that of the accelerator with one PE to ease the observation. Most kernels achieve a linear performance improvement. Such kernels as AES, NW and VITERBI can be divided into fully parallel jobs and thus reach close-to-ideal speedup. The performance of the SORT kernel does not scale linearly due to its tree-reduce characteristic, i.e., the degree of parallelism is reduced by 2x after each merge layer. Therefore, the last few merge layers will have less degrees of parallelism than the number of PEs available, and it cannot be accelerated in fully parallel. For the BFS kernel whose jobs are chain-dependent because of the sequentially accessed queue structure, PE replication is not applicable and thus the kernel is not shown in Fig. 3.7.

Fig. 3.8 presents the overall speedup of each accelerator over the Xeon CPU core after implementing pipelining and PE duplication. The number under each kernel's name represents the best PE duplication factor. The horizontal line (1) represents the CPU baseline with the bars above the line representing speedup and below representing slowdown. Most accelerator designs have two-orders-of-magnitude speedups

[2]Some kernels may not generate a 128-PE design due to FPGA resource constraints. The corresponding bar is thus left blank.

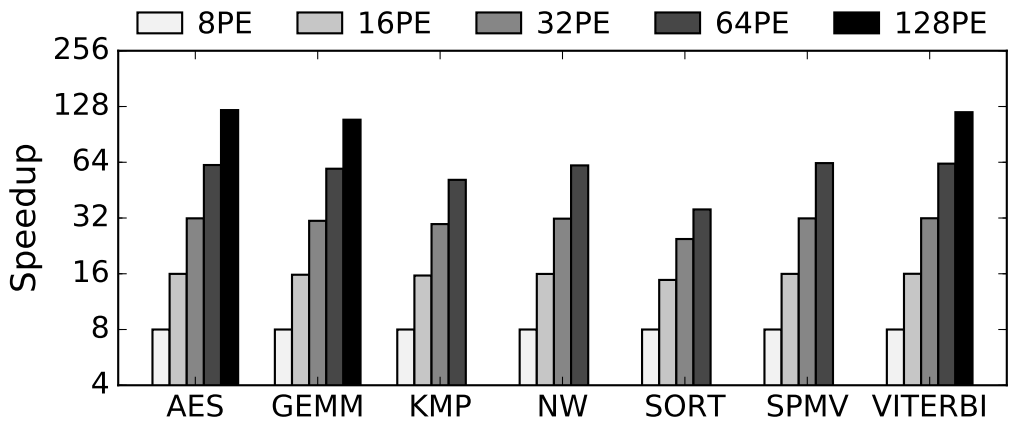


Figure 3.7: Performance speedup on computation lead by PE replication.

over those being performed explicit data caching and start to outperform the CPU core. But the speedups are still considerably far from satisfactory. This is due to the fact that the DRAM access overhead comes back again to play an important role in the overall execution (see Fig. 3.9) after the computation routine is significantly accelerated. Section 3.5 further optimizes the data movement to address this issue.

3.5 Faster Data Movement

We now start from the accelerators that have applied explicit data caching, pipelining and PE duplication. As shown in Fig. 3.9, DRAM access becomes the major performance bottleneck again. Section 3.5.1 and 3.5.2 present the use of double buffering and scratchpad reorganization to increase the DRAM bandwidth utilization from the temporal and spatial aspects, respectively. After such code reconstruction, the performance of the MachSuite accelerators can be further improved by 1.2∼19.2×.

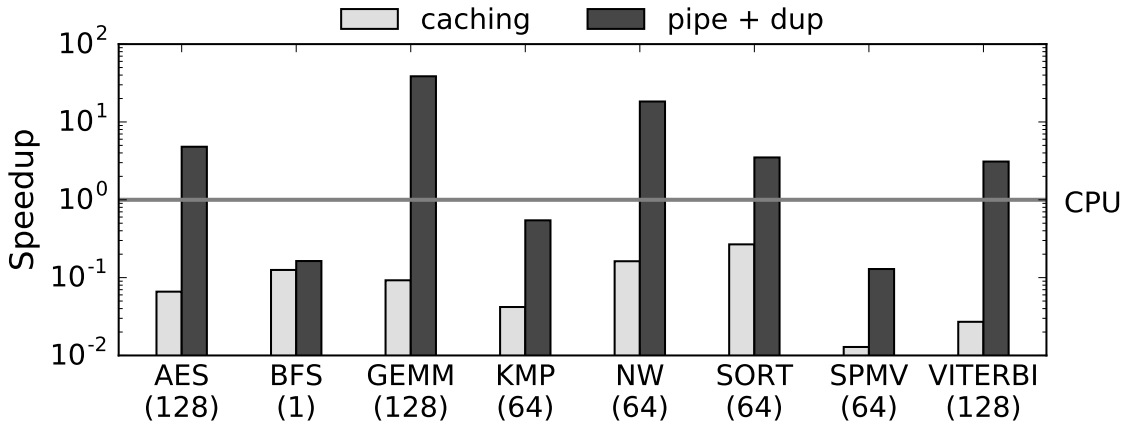


Figure 3.8: Overall performance speedup after applying loop pipelining and PE replication.

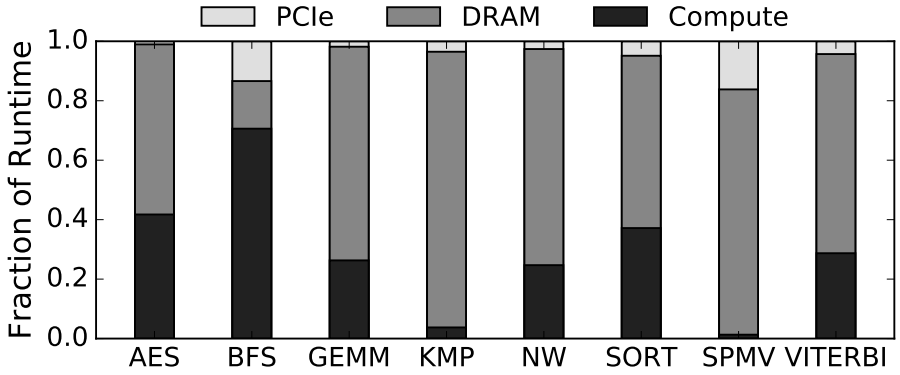


Figure 3.9: Execution time breakdown before applying Strategy #4.

3.5.1 Strategy #4: Double Buffering

As is illustrated in Fig. 3.4(b), although the accelerator designs perform computation in parallel, each PE processes different *load-compute-store* iterations sequentially. In other words, the *N-th* iteration starts to load data after the *(N-1)-th* iteration stores data back to DRAM. However, the data loading of the *N-th* iteration could have

happened earlier, right after the *(N-1)-th* iteration finishes loading data and starts computation, since the read channel of the AXI bus, which interfaces between the accelerator and DRAM, becomes free. In general, the *load*, *compute* and *store* procedures of adjacent iterations can be overlapped to form a 3-stage coarse-grained pipeline, which results in an improved resource utilization as well as a better performance. Our best-effort practice uses the double buffering strategy to realize such a coarse-grained pipeline.

Fig. 3.4(c) illustrates the implementation of double buffering through an update of the code example in Fig. 3.4(b). This update features two major changes. First, the local arrays for explicit data caching are duplicated into three identical copies. This corresponds to the architectural change where the intermediate BRAM layer is duplicated into three identical BRAM buffer groups, as illustrated in Fig. 3.5(c). Second, a switch/case statement is added to schedule the *load*, *compute* and *store* procedures. This schedule is the key to realizing double buffering.

To better explain the scheduling mechanism, we first make the following denotations. We use the variable names in the code example to denote the three buffer groups, i.e., *buf_data[0]*, *buf_data[1]* and *buf_data[2]*, corresponding to the hardware component **X**, **Y** and **Z** in Fig. 3.5(c). We also index the execution phases by letter **i**. In addition, we denote the input and output data of the *k-th* iteration as I_k and O_k. The scheduling mechanism is then described as follows.

- ***i* == *0***: load I_0 into *buf_data[0]*
- ***i* == *1***: load I_1 into *buf_data[1]*; process I_0 in *buf_data[0]*
- ***i* == *2***: load I_2 into *buf_data[2]*; process I_1 in *buf_data[1]*; store O_0 and free

buf_data[0]

- ***i* == *3***: load I_3 into *buf_data[0]*; process I_2 in *buf_data[2]*; store O_1 and free *buf_data[1]*
- ...

As shown in Figure 3.10, double buffering contributes up to 2.1× performance improvement. Most kernels achieve at least a 20% performance improvement. The BFS kernel cannot be benefited from this technique, mainly because the queue-based searching mechanism determines that the compute results of an iteration will affect the input data to load in the next iteration. KMP is another kernel of which the performance is almost not changed, which is mainly due to the fact that the output of KMP is merely an integer representing the number of substrings found in MachSuite.

3.5.2 Strategy #5: Scratchpad Reorganization

When it comes to the spatial aspect, the issue of DRAM bandwidth utilization becomes delicate. We use a piece of C code to reveal the issue. List 3.1 shows four C statements that declare four arrays. While defined in different types with different lengths, all such arrays represent a 1KB contiguous memory space and are equivalent from a CPU programmer's perspective. Specifically, each type of array can be cast to and used as any other type, as shown in List 3.1.

List 3.1: Sample code to demonstrate the difference between CPUs and FPGAs in interpreting arrays.

```
char arr_byte[1024];    // Statement #1
short arr_short[512];   // Statement #2
```

```
int arr_int[256];       // Statement #3
long long arr_ll[128]; // Statement #4
// cast and use int array as char array
char* p = (char *)arr_int;
for (i=0; i<1024; i++) p[i] = 0;
// cast and use short array as long long array
long long* q = (long long *)arr_short;
for (i=0; i<128; i++) q[i] = 0;
```

In FPGA programming using HLS-C, however, the above four arrays are essentially different from each other and cannot be cast to and used as other types. We use Statement #1 and #3 in List 3.1 to explain the difference. Statement #1 defines a 1024-entry byte array, which is synthesized into an on-chip BRAM buffer with width 8 bits and depth 1024. In other words, an accelerator can at most read a byte of data in each FPGA cycle from this buffer, and thus takes at least 1024 cycles to traverse it. In contrast, Statement #3 represents a BRAM buffer with width 32 bits and depth 256, indicating that it can be traversed in 256 FPGA cycles with each cycle reading 32-bit data. In fact, SDAccel supports a BRAM buffer to have up to 512-bit data width, but primitive C types have at most 64-bit width. As a result, the DRAM bandwidth utilization can achieve at most 12.5% of the ideal value, and will be less than 2% of the ideal value if the buffer is defined as the *char* type.

The above analysis makes it clear that programmers can increase the BRAM bandwidth utilization by declaring BRAM buffers used for explicit data caching with larger widths. HLS-C provides a large-width integer type template - *ap_int*$<W>$, where W denotes the width of the data type. Harnessing this template, we propose

the scratchpad reorganization technique to increase DRAM-BRAM transfer bandwidth.

Fig. 3.5(d) illustrates the proposed approach which makes two major changes on the accelerator architecture. First, we replace the three BRAM buffer groups, **X, Y and Z** in Fig. 3.5(c), with three new BRAM buffer groups with the same capacity but larger width. The value of the buffer width is restricted to be the power of two between 8 and 512, so as to be compatible with C types and the AXI bus width (512 bits) between the accelerator and DRAM. Second, we add another BRAM layer which is identical to that in Fig. 3.5(b), i.e., one of the **X, Y and Z** in Fig. 3.5(c). In other words, while the three buffer groups are updated with larger width, we still keep a copy of the original, normal-width buffer group to directly communicate with PEs. Given these architectural changes, the *load*, *compute* and *store* procedures are updated as follows.

- ***Load.*** Loading input data from DRAM to one of the large-width BRAM buffer group
- ***Compute.***

  - Transferring input data from the large-width to normal-width BRAM buffer groups

  - Parallel computing, which remains unchanged

  - Transferring output data from the normal-wdith to large-width BRAM buffer groups
- ***Store.*** Storing output data from one of the large-width BRAM buffer group back to DRAM

The code update to implement this scratchpad reorganization strategy is illustrated in Fig. 3.4(d). This approach increases the computation time by adding two BRAM-BRAM data transfers in the *compute* function. However, since the bottleneck is on the DRAM side, and both the large-width and normal-width buffer groups are partitioned, thus transferring data in parallel, the overhead on computation does not seriously harm the overall performance.

Resource constraints play an important role in applying scratchpad reorganization. Given a certain capacity, a BRAM buffer with a larger width usually consumes more BRAM blocks. Specifically, a BRAM block in the Virtex-7 fabric has a 18Kb capacity with at most 36-bit width. It costs at least 8 blocks to construct a 256-bit BRAM buffer, and 15 blocks for a 512-bit buffer. For an accelerator design with 128 PEs, it costs at least 5760 BRAM blocks to allow 128 large-width BRAM buffers for all three buffer groups, while there are only around 3000 BRAM blocks available on the FPGA fabric. Therefore, a systematic design space exploration approach is necessary to eliminate the manual effort in identifying the optimal resource allocation strategy, which leads to our CPP analytical model presented in Section 4.3.

As shown in Fig.3.10, for the scratchpad reorganization strategy, the KMP and AES kernels achieve significant speedups since their original input/output types are the 8-bit *char* type. A *char*-type BRAM buffer can be enlarged to an *int*-type buffer without even consuming any more BRAM, since a BRAM block can be configured into a up-to-36-bit buffer. As a consequence, these two kernels can be greatly improved via scratchpad reorganization without consuming too much BRAM. On the other hand, kernels such as SPMV and GEMM have already used wider C types, such as int, float and double. Each increment of the buffer width may lead to up to 2× BRAM consumption, and the speedup is thus limited.

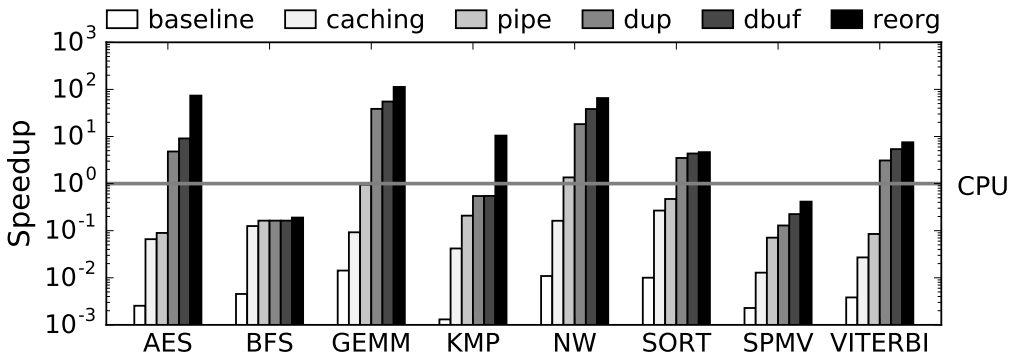


Figure 3.10: Performance improvement by applying five optimization techniques step by step (accumulative).

List 3.2 presents the complete AES kernel code after applying all five optimization strategies, and Fig. 3.10 summarizes the performance improvement of each optimization strategy in the refinement steps. Except for the SPMV and BFS kernels that have been determined non-acceleratable in our experimental platform even before applying any refinement strategy (see Table 3.4 in Section 3.6), all the kernels have outperformed CPU by at least 4.7×. 50% of the kernels achieve at least an order-of-magnitude speedup. Meanwhile, compared to the naive accelerators generated from the original software kernels in MachSuite, the proposed flow brings 42∼29,030× performance improvement, which demonstrates the effectiveness of our best-effort code reconstruction practice of five refinement strategies.

List 3.2: The complete AES kernel code after applying all five strategies.

```
// BATCH_SIZE: the size of data cached on chip
// PE_NUM: the number of PE replicas
// PE_BATCH: the size of data processed by each PE
int PE_BATCH = BATCH_SIZE / PE_NUM;
```

```
// aes(...): performing the main computation; its body is elided
void aes(char data[]) {
  ...
}
// load(...): load off-chip data to on-chip buffer
void load(ap_uint<512> buf[PE_NUM][PE_BATCH/64], char *in) {
  for (int i=0; i<PE_NUM; i++) {
    memcpy(buf[i], in+i*PE_BATCH/64, PE_BATCH);
  }
}
// store(...): store on-chip data off chip
void store(char *out, ap_uint<512> buf[PE_NUM][PE_BATCH/64]) {
  for (int i=0; i<PE_NUM; i++) {
    memcpy(out+i*PE_BATCH/64, buf[i], PE_BATCH);
  }
}
// compute(...): employing a set of PEs for computation
void compute(ap_uint<512> large_buf[PE_NUM][PE_BATCH/64]) {
  char normal_buf[PE_NUM][PE_BATCH];
#pragma HLS array_partition var=normal_buf dim=1
  for (int i=0; i<PE_NUM; i++) {
#pragma HLS unroll
    // SDAccel does not support using memcpy(...) between two arrays
    // We use it here for simplicity
    memcpy(normal_buf[i], large_buf[i], PE_BATCH);
    for (int j=0; j<PE_BATCH; j+=16) {
```

```
#pragma HLS pipeline
      aes(&normal_buf[i][j]);
    }
    // Same as the memcpy(...) call above
    memcpy(large_buf[i], normal_buf[i], PE_BATCH);
  }
}
// kernel(...): the top-level computation kernel function
void kernel(ap_uint<512> *data, int size) {
  ap_int<512> large_buf[3][PE_NUM][PE_BATCH/64];
#pragma HLS array_partition var=buf_data dim=1
#pragma HLS array_partition var=buf_data dim=2
  // Here we omit the code for boundary cases for simplicity
  for (int i=0; i<size/BATCH_SIZE; i++) {
    switch(i % 3) {
      case 0:
        load(large_buf[0], data+i*BATCH_SIZE/64);
        compute(large_buf[1]);
        store(data+i*BATCH_SIZE/64, large_buf[2]);
        break;
      case 1:
         load(large_buf[1], data+i*BATCH_SIZE/64);
        compute(large_buf[2]);
        store(data+i*BATCH_SIZE/64, large_buf[0]);
        break;
      default: // case 2
```

```
        load(large_buf[2], data+i*BATCH_SIZE/64);
      compute(large_buf[0]);
      store(data+i*BATCH_SIZE/64, large_buf[1]);
      break;
    }
  }
}
```

3.6 Discussion

Section 3.3, 3.4 and 3.5 present our best-effort code reconstruction practice. We can see from our step-by-step evaluation results that this practice does benefit a variety of computation kernels. In this section, we further discuss a series of important issues about to what extent this best-effort practice shorten the gap between software programs and hardware behavioral descriptions, what problems it still leave, and in what direction we should move on to further improve the FPGA programmability.

Adaptability. We first discuss what types of computation kernels are better benefited by our best-effort practice. First, kernels that are compute-intensive have the potential to be accelerated by the platform. The two kernels that do not achieve speedup are both communication-intensive kernels. The MachSuite BFS kernel simply traverses all the nodes in a graph without doing any computation, and sparse matrix-vector multiplication is a well-known communication-intensive problem. Such communication-intensive kernels often lead to a serious data transfer overhead compared to the CPU execution, which can probably be detected by the CPU-FPGA

communication time measurement.

While previous work often omits host-device communication when calculating speedup, our study considers the system-level speedup that includes all the computation off- loading overhead from a CPU to an FPGA, which is more practical in real deployment of FPGAs in servers. In our platform that uses the PCIe connection between the CPU and FPGA, we calculate the PCIe transfer time as the elapsed time of data movement from the host memory to the device memory through PCIe-based direct memory access (DMA). Although the optimization on PCIe transfer is beyond the scope of this study (and is addressed in Chapter 5), the PCIe transfer time (or more generally, the CPU-FPGA communication time) serves as a valuable indicator to filter out the FPGA acceleration for communication-bounded kernels before any refinement.

Table 3.4 lists the PCIe transfer time of the MachSuite kernels used in the paper. Each kernel's PCIe transfer time is normalized to its execution time on the Xeon CPU. While most kernels have negligible PCIe transfer time and large speedup potentials, the BFS and SPMV kernels show severe PCIe transfer overheads This explains why the BFS and SPMV kernels are not accelerated in our experimental platform.

Table 3.4: PCIe transfer time normalized to CPU runtime

Kernel	PCIe	Kernel	PCIe	Kernel	PCIe
AES	2.2×10^{-3}	**BFS**	**0.8**	GEMM	6.0×10^{-4}
KMP	5.9×10^{-2}	NW	1.5×10^{-3}	SORT	4.9×10^{-3}
SPMV	**1.3**	VITERBI	1.4×10^{-2}		

Moreover, kernels that conceive massive task-level parallelism can be accelerated.

The FPGA fabric consists of a great many distributed computation and memory building blocks and is naturally fit for applications with a large number of parallel tasks. As will be discussed in Chapter 5, we have explored the integration of FPGA accelerators into Apache Spark applications that feature massive degree of parallelism.

Finally, our best-effort practice benefits kernels that spend a large portion of time on loop blocks. Programmers can customize a hardware pipeline for a loop block through just a pragma, which allows multiple loop iterations to be executed simultaneously in a pipeline. In addition, PE replication that enables task-level parallelism can also be easily programmed through a pragma to unroll the loop with proper memory partitioning pragmas.

Moving Software Programs Closer to Hardware Descriptions. One of the most fundamental differences between software programs and hardware behavioral descriptions is that explicit manipulation of the cache memory system is required in accelerator programming. Both CPU and FPGA architectures require programmers to specify algorithms, but the former virtualize to programmers a low-latency memory system through hardware-supported cache hierarchy. FPGAs, contrarily, supplies programmers with on-chip scratchpads that have more flexibility, but the fulfillment of this flexibility advantage requires explicit program statements. Therefore, almost all the suggested strategies are doing various on-chip scratchpad manipulations, e.g., explicit data caching, memory partitioning, double buffering and scratchpad reorganization. As a result, our best-effort practice effectively shortens the gap.

However, while pinpointing results guide the accelerator refinement to move for-

ward, resource constraints let programmers feed back to the design choice made in prior iterations. This is a critical issue to the best-effort practice since it is inevitable to lead to try-and-fails to find the best resource allocation approach. Worse still, there is an exponential design space of resource distribution to various resource-conflict optimization strategies. As a consequence, a systematic approach for design space exploration is crucial to the improvement of the FPGA programmability.

Most prior work attempts to automatically search for the optimal solution for an individual strategy, e.g., automatic data tiling. As an initial attempt, Wang et al. [WHZ16] also proposes a performance analysis framework to guide designers in choosing proper optimizations for OpenCL applications on Altera FPGAs. However, they still target hardware designers and do not present convincing speedups over CPU. A comprehensive decision-making mechanism across all the strategies applied in an entire design flow was still an open problem, which motivates us to propose our automated accelerator generation methodology to address it.

From Manual Refinement to Design Automation. It is always the ultimate objective to make everything automated. While the paper delivers an encouraging message that a software programmer may also make high-quality FPGA accelerators without systematically learning RTL design expertise, the gap towards complete automation is still considerable. Most existing automation strategies, as we summarize in Section 2.4.1, focus on one optimization problem and more or less make some restrictions to user programs. It implies that these strategies are not only specific but hard to integrate together to form a comprehensive automation tool. For example, if we first apply an optimal auto-caching approach, which tries to utilize as many as memory to minimize computation latency, then other optimizations may have no room to perform. Although some developers attempt to build system-level automa-

tion frameworks [ZHX15, CHP16a], the lack of a mature open source infrastructure and community prevents researchers from contributing their solutions to coordinate with others. Nonetheless, the proposed best-effort practice may serve as a direction for researchers and vendors to develop an effective automation flow that works for a broad class of applications. The AutoAccel framework that is going to be presented in Chapter 4 is one in this direction.

3.7 Conclusion

While the FPGA is changing its role from special-purpose hardware to primary computing resource, we demonstrate that a best-effort code reconstruction practice does produce compelling FPGA accelerators, which lays the foundation for further design automation. For a wide class of applications from a state-of-the-art accelerator benchmark suite MachSuite, our best-effort practice improves the naive accelerator performance by 42∼29,030×, which is 34.4× faster than a Xeon CPU core. Regarding the objective of shortening the gap between software programs and hardware behavioral descriptions, the best-effort practice provides a promising direction, but still leaves a great deal of manual effort. This inspires us to perform design automation on top of it to deliver a nearly push-button experience to end users.

CHAPTER 4

Automated Accelerator Generation

This chapter presents our automated accelerator generation methodology. The methodology is inspired by the pros and cons of the best-effort code reconstruction practice. Specifically, we derive from the practice the composable, parallel, pipeline (CPP) microarchitecture as an accelerator design template as the basis for automation. The proposed CPP analytical model and the automatic code transformation framework AutoAccel avoids the manual try-and-fails for identifying the best design configuration, and deliver a nearly-push button experience to end users[1].

4.1 Overview

Our analysis study in the previous section delivers a best-effort code reconstruction practice of five transformation strategies, which is applicable to a variety of computation kernels and leads to 42∼29,030× performance improvement of the accelerators. We first summarize the transformation strategies as follows:

- Strategy #1: Explicit data caching.
- Strategy #2: Pipelining.

[1]This study is presented in [CWY18b]. I would like to convey my appreciation to all coauthors for their contributions to this study.

- Strategy #3: PE Replication.
- Strategy #4: Coarse-grained pipelining.
- Strategy #5: On-chip memory organization.

Through the demonstration of the best-effort practice and the further discussion, we conclude that performing these transformations still require heavy code reconstruction and intimate knowledge of hardware intricacies. In particular, these transformations are intervened with each other in terms of performance and resource consumption, which considerably increases the complexity of the performance-resource trade-offs. Even for experienced hardware designers, it still requires a great deal of effort to resolve such complicated trade-offs and identify the optimal design choice. For instance, it takes a graduate student with solid HLS programming capabilities many hours to perform the necessary code transformations for the AES kernel, and a few extra days to figure out the optimal design parameters.

Motivated by the pros and cons of the best-effort practice, we propose the composable, parallel and pipeline (CPP) microarchitecture that is derived from the best-effort practice as an accelerator design template. By doing this, we significantly reduce the design space from "anything possible" to only the scope of CPP. Then, we perform design space exploration to realize the optimal configuration of the CPP-based accelerator design to maximize the performance under the resource constraints. In particular, we derive an analytical model to analyze and evaluate the design space as well as the performance and resource consumption, and further propose a series of pruning strategies to reduce the design space so that it can be exhaustively searched within one hour. Finally, we develop the AutoAccel framework to automate the entire accelerator generation process and provide end users with a nearly push-button

experience. Our experiments show that the AutoAccel-generated accelerators outperform their corresponding software implementations by an average of 72× for the MachSuite computation kernels.

4.2 Accelerator Design Template

In this section we present our approach to automatically transform a user C program to a high-quality accelerator behavioral description. We first formulate the problem, and then introduce the CPP microarchitecture that serves as an accelerator design template to address the problem.

4.2.1 Problem Formulation

Formally, this paper aims to solve the following problem: given an input C/C++ computational kernel that satisfies the following constraints, perform automatic code transformation to the kernel under the hardware resource constraints so that the performance of generated accelerator design is maximized.

- ***Synthesizable***. The input kernel must be synthesizable via commercial HLS tools. That is, it should not include recursive function calls or dynamic memory allocation. However, this constraint does not affect the scope of supported kernels since it is always possible for programmers to manually transform such code structures to equivalent, synthesizable structures.
- ***Cacheable***. The memory footprint of any single instance of the top-level loop must be smaller than the FPGA on-chip memory capacity to ensure that the kernel computation and external memory transaction can be fully decoupled.

That is, no matter how large the kernel's input is, we can divide it based on each iteration's need, and cache and process the data needed for at least one loop iteration at a time.

We develop Algorithm 1 to determine whether an input program meets the constraints (we call this process *legalization checking*). Algorithm 1 accepts a HLS-synthesizable computation kernel program as input, and traverses each top-level loop (L_k) to determine whether it can be mapped to the CPP microarchitecture, wherein the determination further depends on the following two factors.

Task-dependent vector chunk size. If the data of an input vector can be split into different chunks, each of which can be processed by an individual PE, then the input vector is called a task-dependent vector. For instance, in the AES kernel the input data to encrypt can always be split into 128-bit chunks to process, no matter how large the entire input data is. For each task-dependent input vector, Algorithm 1 tries to analyze the minimum size of the data chunk to ensure that at least one chunk can be entirely cached on chip. If the size is either not obtainable or too large, then the kernel fails legalization checking.

Task-independent vector capacity. A task-independent vector, on the other hand, is an input vector that is shared by all PEs. The input encryption key of the AES kernel, which is apparently needed by all encryption instances, serves as a perfect example of the task-independent vector. Algorithm 1 tries to analyze the capacity of each task-independent vector to ensure that the on-chip BRAM blocks can hold at least one copy of it. If the capacity is either not obtainable or too large, then the kernel fails legalization checking.

We perform legalization checking by traversing an abstract syntax tree (AST).

We analyze the iteration domain to reason kernel accessed data size by the polyhedral analysis from [PZS13].

Algorithm 1 Legalization Checking

Require: A C computation kernel K that is HLS-synthesizable.

Ensure: For each top-level loop L_k if it can be mapped to the CPP microarchitecture.

for each $L_p \in L_k$ **do**
  $\mathbb{A}_{ref} \leftarrow \varnothing$
  for each $A \in K.InterfaceArrays()$ **do**
    if $I \leftarrow A.GetRefIdxOf(L_p) = \varnothing$ **then**
      $\mathbb{A}_{ref}.insert(A)$
    else
      for each $i \in I$ **do**
        $(a, b) \leftarrow i.GetCoeffAndOffset()$
        if $hasIter(a) \vee hasIter(b) \vee b > a$ **then**
          continue
        end if
      end for
    end if
  end for
  $TotalSize \leftarrow 0$
  for each $A \in \mathbb{A}_{ref}$ **do**
    if $Not(s \leftarrow A.GetArraySize())$ **then**
      continue
    end if
    $TotalSize \leftarrow TotalSize + s$
  end for
  if $TotalSize > BRAMSize$ **then**
    continue
  end if
  return True
end for
return False

Based on our problem formulation, computational kernels featuring extensive random accesses on a large memory footprint, e.g., PageRank [PBM99] and the breadth-

first search (BFS) algorithm, will probably not meet the *Cacheable* constraint. On the contrary, computational kernels that process input data block by block generally meet these constraints. In fact, almost all streaming and batch processing kernels with regular data-level parallelism fall into this category. These kernels are also well-known to potentially benefit from FPGA acceleration. For the kernel that satisfies the above constraints, we implement it using our proposed microarchitecture, which we will discuss in the following section, to bound the design space.

4.2.2 Composable, Parallel, Pipeline Microarchitecture

The composable, parallel, pipeline (CPP) microarchitecture is proposed as a template of accelerator designs. For an input kernel that meets the above constraints, our approach first fits the kernel into the CPP microarchitecture, then performs design space exploration to identify the optimal parameter configuration, and finally transforms the input kernel code to the CPP microarchitecture description code. The CPP microarchitecture guarantees the quality of the output accelerator design by providing a series of features to realize the transformations we summarized in Section 4.1. In the remainder of this section, we introduce the key features of the CPP microarchitecture using the AES benchmark.

Feature #1: Coarse-grained pipeline with data caching. The overall CPP microarchitecture consists of three stages: `load`, `compute` and `store`. The `kernel` function in the AES source code only corresponds to the `compute` module instead of defining the entire accelerator. The input data to encrypt are processed block by block, i.e., iteratively loading a certain number of 128-bit data chunks into on-chip buffers (Stage `load`), encrypting these chunks (Stage `compute`), and storing

the encrypted data chunks back to DRAM (Stage `store`). This feature is derived from *Strategy #1* because off-chip data movement only happens in the `load` and `store` stages, leaving data accesses of computation completely on chip. In general, as far as the input kernel meets the *Cacheable* constraint, it is able to fit into this load-compute-store execution process.

The `load` and `store` modules connect to two input and output DRAM buffers, respectively, through AXI channels. The bit-widths of these buffers, i.e., the data widths of the AXI channels, are decoupled from the type sizes of the top-level function arguments. This allows the off-chip data transfer to be performed with the maximum achievable throughput of the underlying CPU-FPGA platform. Furthermore, if no dependency or only forward dependency exists between different blocks of input, the `load`, `compute` and `store` stages of different blocks can be processed in pipeline, and these three stages then form a coarse-grained pipeline that overlaps computation with off-chip data communication. This feature of the CPP microarchitecture could further improve the effective bandwidth of the accelerator. *Strategy #4* is realized as well.

Feature #2: Loop scheduling. The CPP microarchitecture tries to map every loop statement presented in the computational kernel function to either 1) a circuit that processes different loop iterations in parallel, 2) a pipeline where the loop body corresponds to the pipeline stages, or 3) a combination of both. As for the AES example, the loop statement in the `kernel` function is mapped to a set of PEs to process the sequence pairs in parallel. The loop statements in each PE are mapped to parallel and pipeline circuits as well. This realizes *Strategy #2 and #3*.

Feature #3: On-chip buffer reorganization. In the CPP microarchitecture, all

the on-chip BRAM buffers are partitioned to meet the port requirement of parallel circuits, where the number of partitions of each buffer is determined by the duplication factor of the parallel circuit that connects to the buffer. This feature is used for realizing *Strategy #5*. In the AES example, the on-chip buffers that cache the input and output sequence pairs are partitioned into multiple segments, each segment feeding one PE.

In summary, the CPP microarchitecture provides these features to realize the aforementioned transformations so as to ensure the quality of output accelerator designs. Moreover, the use of an accelerator design template implies a clear design space: all valid configurations of the CPP microarchitecture. We analyze the design space in the following section.

4.2.3 Design Space Analysis

The CPP microarchitecture design space is determined by all its loops and external memory buffers, which is formulated as follows:

$$\mathcal{A} = \{\mathcal{L}, \mathcal{B}\} \tag{4.1}$$

where $\mathcal{A}$ denotes the overall design space, and $\mathcal{L}$ and $\mathcal{B}$ mean the loop set and external memory buffer set, respectively.

We then formulate the possible scheduling of loops as follows:

$$\forall L \in \mathcal{L}, L = \{(\alpha, \beta) \mid 1 < \alpha < L_{tc}, \beta = \{0, 1\}\} \tag{4.2}$$

where α is the integer unroll factor of loop L with trip count L_{tc} as its maximum, and β is a binary variable to indicate if the pipeline scheduling is enabled or not. As a result, the design space complexity of $\mathcal{L}$ is $O(2^m \times \prod_{L \in \mathcal{L}} L_{tc})$ where m denotes the total number of loops.

Finally, the possible design choices for external memory buffers can be represented as follows:

$$\begin{aligned} \forall B \in \mathcal{B}, B = \{(\mu, \nu) \mid 8 \leq \mu \leq 512, 0 \leq \nu \leq C_{BRAM}\} \\ \sum_{B \in \mathcal{B}} B_\nu \leq C_{BRAM} \end{aligned} \tag{4.3}$$

where μ and ν are the integer bit-width and the capacity of the on-chip memory buffer that caches a certain external memory buffer B, respectively. C_{BRAM} denotes the total capacity of all BRAM blocks. Thus, the design space complexity of $\mathcal{B}$ is $O((512 \times C_{BRAM})^n)$, where n denotes the total number of buffers.

Consequently, the overall design space complexity is $O((512 \times C_{BRAM})^n \times 2^m \times \prod_{L \in \mathcal{L}} L_{tc})$, which is too large to be explored exhaustively. In fact, even the NW motivating example contains roughly 1.4×10^{17} design points. To rapidly find the optimal design choice among such a tremendous design space, we analytically model performance and resource utilization in Section 4.3, and introduce our design space exploration flow with a series of pruning strategies in Section 4.4.

4.3 CPP Analytical Model

This section presents our analytical model to quantify performance and resource consumption of the CPP microarchitecture. While a number of previous studies have attempted to model FPGA designs [KDP16, WHZ16, ZMS16, ZPL16, ZPW17, Zha17], our model targets at a well-defined accelerator microarchitecture and thus features a highly accurate modeling of the utilization of the FPGA on-chip resources. On the other hand, some of the existing models for general FPGA accelerator designs focus on only the performance estimation [ZMS16, WHZ16]. Although others also have the model for different kind of resources [KDP16, ZPL16, ZPW17, Zha17], their LUT models are either based on machine learning [KDP16, ZPW17] or even missing [ZPL16, Zha17] (see Section 2.4.1).

4.3.1 Performance Modeling

The performance model estimates an accelerator's overall execution cycle (C) through Eq. 4.4:

$$C = max(C_l + C_s, C_c) \tag{4.4}$$

where C_l, C_c and C_s denote the cycles of the load, compute, and store modules, respectively. Since the load and store modules share the off-chip bandwidth in our experimental platform, we make a maximum operation between the cycles of the load/store modules and that of the compute module.

The execution cycles of the load, compute and store modules, as well as all of their

submodules, can be quantified as the total cycles of all the loops (C_{loop}), submodules (C_{mod}) and standalone logic (C_r), as shown in Eq. 4.5.

$$C_{mod}(M) = \sum_{i \in M.loops} C_{loop}(i) + \sum_{m \in M.mods} C_{mod}(m) + C_r(M) \tag{4.5}$$

where M denotes an arbitrary hardware module.

Then we model the loop execution. Although a loop statement can be scheduled in pipeline, parallel, or the combination of both, the first two can be treated as special cases of the last one, and can together be modeled as Eq. 4.6:

$$C_{loop}(L) = C_{iter}(L) + II(L) \times \frac{TC(L)}{UF(L)} \tag{4.6}$$

where L denotes an arbitrary loop; C_{iter}, II, TC and UF denote the iteration latency, initiation interval, trip count and unroll factor, respectively.

Subsequently, we break down and model the loop iteration in Eq. 4.7, where the loop iteration latency is composed of the total cycles of all the sub-loops, submodules and standalone logic.

$$C_{iter}(L) = \sum_{i \in L.loops} C_{loop}(i) + \sum_{m \in L.mods} C_{mod}(m) + C_r(L) \tag{4.7}$$

Eq. 4.5 and Eq. 4.7 reflect the architecture hierarchy with nested modules and loops. The proposed model recursively traverses all the loops and modules until a loop or module does not contain any sub-structures. In addition, we can find that

Eq. 4.5 and Eq. 4.7 are almost identical. This is because the loop iteration can be treated as a special "module" and modeled in the same way for both performance and resource.

4.3.2 Resource Modeling

The resource model estimates the consumptions of the four FPGA on-chip resources: BRAMs, LUTs, DSPs and FFs. As the DSP model is fairly straightforward and the FF model is similar to the LUT model, we only demonstrate the BRAM and LUT models.

BRAM modeling: The BRAM consumption of a hardware module consists of the BRAM blocks used by all its local buffers (R^{mem}_{buf}) and those used by all its submodules (R^{mem}_{mod}), as shown in Eq. 4.8:

$$R^{mem}_{mod}(M) = \sum_{b \in M} R^{mem}_{buf}(b) + \sum_{m \in M.mods} R^{mem}_{mod}(m) \times DF(m) \tag{4.8}$$

where $DF(m)$ is the duplication factor of submodule m which is equivalent to the unroll factor of the loop that includes this submodule. We use "duplication factor" instead of "unroll factor" since the former is a better fit for depicting hardware modules, and the latter is more suitable for describing loop statements.

Then we model the BRAM consumption of on-chip buffers. A buffer's BRAM consumption is determined by three factors: 1) partition factors on all dimensions, $\prod_{d \in dim(B)} PF(d)$; 2) the size of unit partition, $\lceil \frac{S(B)}{\prod_d PF(d)} \rceil$; and 3) the bit-width of the

buffer, $bw(B)$, as shown in Eq. 4.9:

$$R_{buf}^{mem}(B) = \prod_{d \in dim(B)} PF(d) \times V\left(\lceil \frac{S(B)}{\prod_d PF(d)} \rceil, bw(B)\right) \tag{4.9}$$

Eq. 4.9 adopts a function $V(s, bw)$ in [CWY17] (Eq. 4) to calculate the BRAM consumption of a single partition. The two parameters are the size (s) and the bit-width (bw) of the partition. Eq. 4.10 presents its expression:

$$V(s, b) = \lceil \frac{s}{N_{blk}(b) \times S_{unit}} \rceil \times N_{blk}(b) \tag{4.10}$$

$$N_{blk}(b) = \lceil \frac{b}{b_{phy}} \rceil \tag{4.11}$$

where S_{unit} denotes the size of a BRAM block that is a platform-dependent constant. $N_{blk}(b)$ is also a function borrowed from [CWY17], which calculates the minimum number of BRAM blocks needed to compose a buffer with bit-width b. Eq. 4.11 shows its expression, where b_{phy} is a platform-dependent constant that represents the largest supported bit-width of a BRAM building block.

LUT modeling: The LUT consumption of a hardware module (R_{mod}^{lut}) is composed of the number of LUTs used by all loops, submodules, BRAM buffers (for control

logic) and the standalone logic:

$$R^{lut}_{mod}(M) = \sum_{l \in M.loops} R^{lut}_{iter}(l) \times UF(l) + \sum_{b \in M.bufs} R^{lut}_{buf}(b) + \sum_{m \in M.mods} R^{lut}_{mod}(m) \times DF(m) + R^{lut}_{r}(M) \tag{4.12}$$

where R^{lut}_{iter} depicts the LUT consumption of the loop iteration that is, again, treated and modeled as a special "module." R^{lut}_{r} denotes the LUT consumption of the standalone logic.

Besides, the LUT usage of a loop iteration can be further decoupled and quantified as follows:

$$R^{lut}_{loop}(L) = \sum_{l \in L.loops} R^{lut}_{iter}(l) \times UF(l) + \sum_{m \in L.mods} R^{lut}_{mod}(m) \times DF(m) + R^{lut}_{r}(L) \tag{4.13}$$

which is almost identical to Eq. 4.12. Since we always perform loop-invariant code motion in advance, we guarantee that there has no BRAM used in the loop body.

We then model the LUT consumption of on-chip buffers (R^{lut}_{buf}). It can be decoupled into two parts: 1) the control (R^{lut}_{ctrl}) and data (R^{lut}_{data}) signals of each BRAM partition, and 2) the k-to-1 multiplexer ($R^{lut}_{mux}(k)$) that selects the desired data from all the partitions, as shown in Eq. 4.14:

$$R^{lut}_{buf}(B) = R^{mem}_{buf}(B) \times (R^{lut}_{ctrl} + R^{lut}_{data}) + R^{lut}_{mux} \left(\prod_{d \in dim(B)} PF(d) \right) \times bw(B) \tag{4.14}$$

where $R^{lut}_{mux}(k)$ can be calculated using Eq. 7 in [CWY17]. These equations quantify the relationship between a buffer's LUT consumption and its BRAM usage.

Because of the existence of non-linear equations in the proposed model, the problem of identifying the optimal CPP configuration is formulated as an integer non-linear programming (INLP) problem which is not able to be solved in polynomial time. Fortunately, like [CWY17], we can initialize the model by running HLS once (for cycle and BRAM) or twice (for DSP, LUT and FF) to obtain the values of a subset of parameters, since such parameters remain constant once the CPP microarchitecture is constructed: $C_r(M)$, II, TC, $C_r(L)$, S_{unit}, b_{phy}, $R^{lut}_r(M)$, R^{lut}_{ctrl} and R^{lut}_{data}.

We use the performance modeling for an example kernel in List 4.1 to demonstrate this process. List 4.1 presents a kernel of integer vector addition, which accepts an integer vector as input, adds each element with a fixed integer 7, and writes the results back to the same vector. The macros `NUM_PE` and `PE_SIZE` are design parameters whose different possible values will be explored to find the ones that lead to the optimal performance.

List 4.1: Integer vector addition kernel code.

```
#define OVERALL_SIZE 4194304
#define NUM_PE 2
#define PE_SIZE 1024
#define BATCH_SIZE ((NUM_PE)*(PE_SIZE))
void load(int in_buf[NUM_PE][PE_SIZE], int input, int size, bool flag) {
  if (!flag) return;
```

```
  memcpy(in_buf, input, size);
}
void store(int out_buf[NUM_PE][PE_SIZE], int output, int size, bool flag) {
  if (!flag) return;
  memcpy(output, out_buf, size);
}
void pe(int buf[PE_SIZE]) {
  for (int i=0; i<PE_SIZE; i++) {
  #pragma HLS pipeline
    buf[i] += 7;
}}
void compute(int buf[NUM_PE][PE_SIZE], bool flag) {
  if (!flag) return;
  for (int i=0; i<NUM_PE; i++) {
  #pragma HLS unroll
    pe(buf[i]);
}}
void kernel(int* data) {
  int buf_x[NUM_PE][PE_SIZE];
  #pragma HLS array_partition variable=buf_x dim=1 complete
  int buf_y[NUM_PE][PE_SIZE];
  #pragma HLS array_partition variable=buf_y dim=1 complete
  int buf_z[NUM_PE][PE_SIZE];
  #pragma HLS array_partition variable=buf_z dim=1 complete
  int num_batches = OVERALL_SIZE / BATCH_SIZE;
  for (int i=0; i<num_batches+2; i++) {
```

```
    if (i % 3 == 0) {
      load(buf_x, data+i*BATCH_SIZE, BATCH_SIZE*sizeof(int), i<num_batches);
      compute(buf_z, i > 0 && i < num_batches+1);
      store(buf_y, data+(i-2)*BATCH_SIZE, BATCH_SIZE*sizeof(int), i>1);
    }
    // coarse-grained pipelining, elided for brevity
    else if (i % 3 == 1) {...}
    else {...}
}}
```

The first step of the modeling process is to obtain the kernel code hierarchy by parsing the input program. The output of this step is illustrated in Fig. 4.1 (a). We can see that the trip counts and unroll/pipeline information are collected as well. Next, a C-to-RTL synthesis instance is performed. By parsing the synthesis report, we obtain sufficient information to establish the model. Fig. 4.1 (b) illustrates the cycle data retrieved from the report; Fig. 4.1 (c) illustrates the cycle data (mainly cycles for various standalone logic) derived from them. Finally, when another set of design parameters are explored, e.g., changing `NUM_PE` from 2 to 4 to explore a larger loop unroll factor, the established model can be used to directly calculate the overall execution cycle, as shown in Fig. 4.1 (d).

4.4 Design Space Exploration

Fig. 4.2 illustrates our design space exploration (DSE) flow. The DSE flow first initializes the analytical model by performing HLS synthesis instances and parsing the generated reports, and then fetch the set of design parameters from the C kernel

code. As we pointed out in the previous section, exhaustively searching in such a tremendous design space is impractical. As a result, we propose the following strategies to prune the design space:

Small loop flatten: Empirically, it is better to flatten the innermost loops with fixed, small trip counts. For one thing, it provides more opportunities for HLS to generate a more efficient scheduling. For another, it exerts moderate pressure on the overall resource utilization. As a result, we make an ad hoc strategy to fully unroll innermost loops with trip count less than 16.

Code Hierarchy	*Cycle*	*Other values*
kernel		
l-c-s loop		tripcount: 2050
standalone		
load		
memcpy		tripcount: 2048
standalone		
compute		
loop: U		tripcount: 2
pe		
loop: P		tripcount: 1024
standalone		
standalone		
standalone		
store		
memcpy		tripcount: 2048
standalone		
standalone		

(a) Static code analysis

Code Hierarchy	*Cycle*	*Other values*
kernel	***4223001***	
l-c-s loop	***4223000***	tripcount: 2050
standalone		
load	***2058***	
memcpy	***2049***	tripcount: 2048
standalone		
compute	***1028***	
loop: U		tripcount: 2
pe	***1027***	
loop: P	***1025***	tripcount: 1024 ***II: 1, Iter: 3***
standalone		
standalone		
standalone		
store	***2056***	
memcpy	***2049***	tripcount: 2048
standalone		
standalone		

(b) Retrieving synthesis report

Code Hierarchy	*Cycle*	*Other values*
kernel	4223001	
l-c-s loop	4223000	tripcount: 2050
standalone	***2***	
load	2058	
memcpy	2049	tripcount: 2048
standalone	***9***	
compute	1028	
loop: U	***1027***	tripcount: 2
pe	1027	
loop: P	1025	tripcount: 1024 II: 1, Iter: 3
standalone	***2***	
standalone	***0***	
standalone	***1***	
store	2056	
memcpy	2049	tripcount: 2048
standalone	***7***	
standalone	***1***	

(c) Model establishment

Code Hierarchy	*Cycle*	*Other values*
kernel	***4214809***	
l-c-s loop	***4108***	tripcount: 1026
standalone	2	
load	***4106***	
memcpy	***4097***	tripcount: 4096
standalone	9	
compute	***1028***	
loop: U	***1027***	tripcount: 4
pe	***1027***	
loop: P	***1025***	tripcount: 1024 II: 1, Iter: 3
standalone	2	
standalone	0	
standalone	1	
store	***4104***	
memcpy	***4097***	tripcount: 4096
standalone	7	
standalone	1	

(d) Applying model to new case

Figure 4.1: Step-by-step demonstration of the performance modeling process.

Power-of-two bit-widths: We prune the design space by only searching the power-of-two bit-width values. We note that this pruning strategy covers the optimal design point because the BRAM utilization would be the same for all bit-width values that have the same rounding up the the power of two.

Power-of-two buffer capacities: In general, setting the capacity of each on-chip buffer to be a power-of-two value achieves the highest efficiency for the buffer control logic. In fact, commercial tools such as Xilinx SDAccel usually round up the capacity of user-defined on-chip buffers to the nearest power-of-two value by default. Therefore, it does not lose optimality to apply this pruning strategy.

Loop unroll factor pruning: Loop unroll factors determine the number of on-chip BRAM partitions. This number is bounded by the total number of BRAM blocks available for user-defined accelerators, which is approximately a few thousand. This pruning strategy is particularly beneficial for programs with deep, complicated loop hierarchy.

Saddleback search for loop unroll factors: The search problem of all loop unroll factors can be formulated as finding a particular value in a N-dimension matrix where the values are sorted in each individual dimension. N denotes the total number of loops. The formulation is based on the following theorem.

Theorem 1. *For unroll factor L_α of loop L in the design parameter set, the overall execution cycle C is negatively correlated to L_α; the consumption of any type of resource R is positively correlated to L_α.*

Proof. For an arbitrary loop L in a computation kernel, the partial derivative of its

unroll factor L_{alpha}, based on Eq. 4.6, can be simplified as

$$\frac{d}{dL_\alpha}C = -k \times \frac{1}{L_\alpha^2} \tag{4.15}$$

where k is a positive number. It will remain negative.

Also, the partial derivative for any type of resource, based on Eq. 4.8 (for BRAMs) and Eq. 4.13 (for LUTs), can be simplified as

$$\frac{d}{dL_\alpha}R = k \tag{4.16}$$

where k is a positive number. It will remain positive. □

Theorem 1 indicates the monotonicity of the performance and resource models with respect to any individual loop unroll factor. This property of monotonicity implies two strategies for design space reduction. First, we can perform binary search on the domain of any single loop unroll factor, especially the one with the largest domain. For many computation kernels, e.g., AES, the domain for the unroll factor of the outermost loop is orders-of-magnitude larger than those of all the other loops, where the binary search strategy turns out to be dramatically effective, e.g., 300× design space reduction.

On the other hand, for any two-dimensional $M \times N$ matrix (assuming $M \geq N$) where each row and column is sorted, the matrix can be searched through via an $O(M)$ algorithm, the Saddleback search algorithm [Bir06]. If M is not far larger

than N, Saddleback search will perform better than binary search. By considering both strategies and applying the one in favor in different cases, we can reduce the time complexity of searching all loop unroll factors from $O(\prod_{L\in\mathcal{L}} L_{tc})$ to $O(\prod_{L\in\mathcal{L}\wedge L\notin\{L_p,L_q\}} L_{tc} \times L_p \times \log \frac{L_q}{L_p})$, where L_q and L_p denote the unroll factors of the two loops with the largest trip counts. This strategy works very well for programs with shallow loop hierarchies.

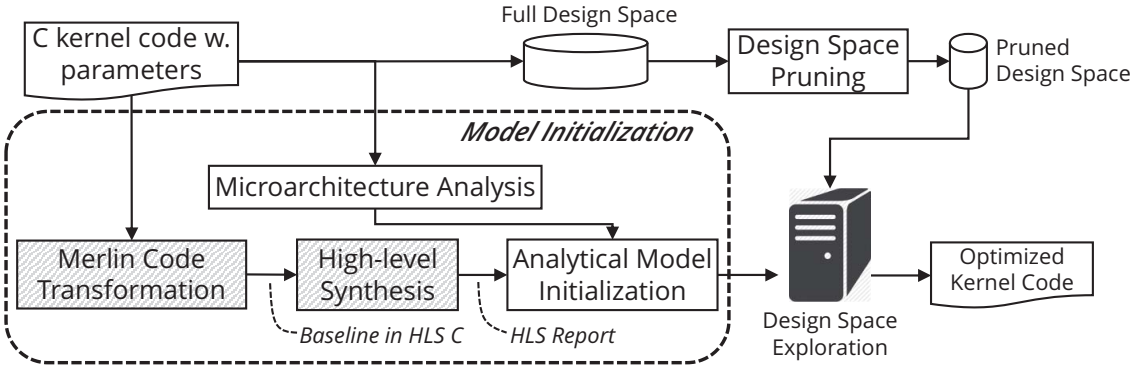


Figure 4.2: Design Space Exploration Flow

Fine-grained pipeline pruning: In general, loop pipelining achieves higher resource utilization and better performance than parallelism in most cases. Formally, we derive the following theorem to realize the loop that is always benefit pipeline.

Theorem 2. *Given a loop L with trip count L_{tc}, iteration latency C_L and resource consumption R_L^{np} before enabling pipelining, and initiation interval II_L and resource consumption R_L^{p} after enabling pipelining. Enabling pipelining is always better if $\frac{L_\alpha}{L_{tc}} \leq (e-1)$ for unroll factor L_α of L, where $e = \frac{C_L/II_L}{R_L^p/R_L^{np}}$.*

Proof. Since the unroll factor L_{alpha} is the number of pipelined PE replicas, the

overall resource consumption (R) of the entire pipelined design is

$$R = L_\alpha \times R_L^p \tag{4.17}$$

The same amount of resource can then generate the following number of non-pipelined PEs (N^{np}):

$$N^{np} = \lfloor \frac{R}{R_L^{np}} \rfloor \leq \frac{R}{R_L^{np}} \tag{4.18}$$

Next, we quantify the execution cycles for the loop L in both pipelined (C_p) and non-pipelined (C_{np}) cases.

$$C_{np} = C_L \times \lceil \frac{L_{tc}}{N^{np}} \rceil \geq C_L \times \frac{L_{tc}}{N^{np}} \geq C_L \times \frac{L_{tc}}{R/R_L^{np}} \tag{4.19}$$

$$C_p = II_L \times \lceil \frac{L_{tc}}{L_\alpha} \rceil \tag{4.20}$$

To ensure that the pipelined design is better, i.e., $C_p \leq C_{np}$, we can have

$$C_L \times \frac{L_{tc}}{R/R_L^{np}} \geq II_L \times \lceil \frac{L_{tc}}{L_\alpha} \rceil \tag{4.21}$$

Denote $\frac{C_L/II_L}{R_L^p/R_L^{np}}$ by e, the above equation is equivalent to

$$e \geq \frac{L_\alpha}{L_{tc}} \times \lceil \frac{L_{tc}}{L_\alpha} \rceil \tag{4.22}$$

The condition $\frac{L_\alpha}{L_{tc}} \leq (e-1)$ can ensure the validation of the above equation.

□

The e in Theorem 2 means the efficiency of enabling pipelining for loop L. Theorem 2 illustrates that when $e \leq 1$, the pipeline implementation is inherently inefficient and should always be disabled. On the other hand, the pipeline implementation is much more efficient than the sequential design and should always be enabled when $e \geq 2$. Finally, when $1 < e < 2$, the unroll factor should not be too large so that the pipelined PE is able to process a sufficient number of loop iterations to ensure the pipeline efficiency.

Although pipelining is generally favorable because of its dramatic performance boost with merely a small amount of resource overhead. However, theorem 2 reveals the fact that it might not always be a preferable design choice. In actuality, applying the pipeline pragma to a loop structure may even decrease the efficiency of resource utilization. List 4.2 demonstrates such an example. It is the code snippet of the most time-consuming loop statement in the kernel. The pipeline pragma of the outermost loop leads to a $II = 16$ pipeline that consumes 75 DSPs; meanwhile, the design without the pragma consumes only 30 DSPs and has a 26-cycle loop iteration latency. That is to say, the 1.6× cycle reduction is traded with a 2.5× resource consumption.

The reason for this inefficient resource utilization is that the optimization objective of the pipelining strategy is to minimize the II of the loop. However, the price to pay for this objective is sometimes too high to afford, just like this example case. Consider the job that we want to schedule two independent floating-point addition operations. Since the latency for a floating-point addition is about eight cycles, we can achieve a minimum eight-cycle latency by employing six DSPs (one adder needs three DSPs). However, if a nine-cycle latency is tolerable, then one adder of 3 DSPs will be sufficient. That is, a one-cycle improvement is traded by 2× DSP consumption.

List 4.2: Code snippet of the VITERBI kernel where applying loop pipelining results in an inefficient design

```
  // Iteratively compute the probabilities over time
  L_timestep: for( t=1; t<128; t++ ) {
#pragma HLS pipeline
    L_curr_state: for( curr=0; curr<5; curr++ ) {
#pragma HLS unroll
      // Compute likelihood HMM is in current state and where it came from.
      L_prev_state: for( prev=0; prev<5; prev++ ) {
#pragma HLS unroll
        p = llike[t-1][prev] +
            transition[prev*5+curr] +
            emission[curr*5+obs[t]];
        if (!prev || p < min_p) {
          min_p = p;
        }
```

```
      llike[t][curr] = min_p;
    }
  }
}
```

Taking our motivating example as an instance, the design space is reduced from 1.4×10^{17} to only 3.2×10^{6} by applying the above strategies. The scale of reduced design space is sufficient to be searched within an hour even using a single modern CPU core.

4.5 Experimental Evaluation

In this section we first present the AutoAccel framework that automates the entire accelerator generation process. Then we describe our experimental setup, followed by the evaluation of the model accuracy as well as the performance of the generated accelerators.

4.5.1 AutoAccel Framework

As shown in Fig. 4.3, we implement a push-button framework called AutoAccel that takes a nested loop in C as input and performs a series of transformations to produce a high-quality FPGA accelerator under the CPP microarchitecture. AutoAccel is implemented on top of the Merlin compiler [mer, CHP16a], a source-to-source transformation tool for FPGA acceleration based on the CMOST [ZHX15] compilation flow. The Merlin compiler takes C/C++ programs with user directives as input, and generates the optimized accelerator kernel code automatically. It provides a li-

brary of code transformation primitives which are leveraged by AutoAccel to agilely construct the CPP microarchitecture. On the other hand, without the need for users to manually insert directives in the input code, the CPP microarchitecture provides an automated way to organize these primitives to come up with high-quality designs. Subsequently, we use static analysis to extract the necessary information (e.g., loop trip count) to form the design space. Then the design space exploration flow we introduced in the previous section is adopted to realize the best design specification in minutes. This design can be directly fed into Xilinx SDAccel to produce a high-quality accelerator bitstream.

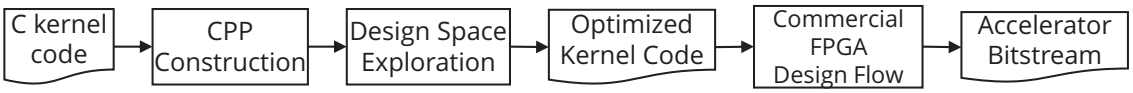


Figure 4.3: AutoAccel Framework Overview

4.5.2 Experimental Setup

The evaluation of AutoAccel is performed on the mainstream PCIe-based CPU-FPGA platform with the Xilinx SDAccel design flow. Table 4.1 lists the detailed hardware and software configuration. An Xeon CPU is connected with a Xilinx Virtex-7 FPGA board through the PCIe interface. For a fair comparison, both the CPU and the FPGA fabric were launched in 2012. On top of the platform hardware, we use Xilinx SDAccel to provide a hardware-software co-design environment. To evaluate the AutoAccel framework, we use the MachSuite benchmark suite that contains a broad class of computational kernels programmed as C functions for accelerator study. For each kernel, MachSuite provides at least one implementation that is programmed without the consideration of FPGA acceleration, which makes it a

natural fit for demonstrating AutoAccel. Please note that for computation kernels like `BFS` that fail legalization checking, AutoAccel simply returns and lets CPU take over their execution.

Table 4.1: Configuration of Hardware and Software

Host CPU Model	Intel Xeon E5-2420 (12 cores) @ 1.9GHz
Host Memory	64GB DDR3-1600
FPGA Fabric	Xilinx Virtex-7
Device Memory	8GB DDR3-1600 (Max Band.: 12.8GB/s)
CPU-FPGA Interface	PCIe Gen3 x8 (Max Band.: 8GB/s)
Transformation Flow	Merlin compiler 2017.1
Synthesis Flow	SDAccel (SDx) 2017.2

4.5.3 Evaluation Results

We first evaluate whether the model-generated results are consistent with those collected from HLS reports. In detail, we randomly select 20 design points for each benchmark, and compare the performance and resource usage for each design point between the model estimation and HLS report. Table 4.2 presents the average absolute difference rates for all cases. We can see that the proposed model aligns with the HLS report accurately on performance and BRAM/DSP usage, and also results in only moderate differences on LUT/FF usage. The differences are lead by the fact that the HLS tool adopts some resource-efficient implementations for its building blocks when a design requires a large proportion of on-board resources. For example, `VITERBI` includes a loop statement with initiation interval equaling to 40 (II=40). The hardware circuit for this loop has some 25-to-1 multiplexers

to select one floating-point number from 25 numbers. We observe that when the number of PEs in the `VITERBI` design grows, the HLS tool automatically replaces a fully-pipelined multiplexer implementation that consumes over 500 LUTs with the implementation that consumes only 32 LUTs to 1) meet the II=40 restriction and 2) save on-board resources. Since such dynamic optimization strategies are hard to be captured in a static analytical model, a few percentages of differences on LUT/FF usage is inevitable.

Table 4.2: Differences Between Model and HLS Reports

Parameter	Perf.	BRAM	DSP	LUT	FF
Avg. error	<1%	<1%	<1%	6.5%	4.3%

We then compare this result with the actual on-board result, and list the error rate for each benchmark in Table 4.3. We can see that the average error rate among all the benchmarks is only 6.2%. We further analyze the benchmarks with over 10% error rate, i.e., `AES` and `KMP`. We find that such a relatively large error rate is mainly because the accelerator designs for these benchmarks have a very small execution time (∼10 ms). For these time frames, the start-up and end overhead bias the time significantly. On the contrary, we also observe that the error rate of the model to on-board execution is always less than 5% when a design has an over 100-millisecond execution time. Hence, the proposed model is able to accurately predict the on-board execution time of a design given that its execution time is tens of milliseconds or larger.

We then evaluate the performance improvement of the generated FPGA accelerator designs. Fig. 4.4 compares the performances between the naive implementation of MachSuite, AutoAccel-generated accelerator designs, manual HLS designs and the

Table 4.3: Differences Between Model and On-board Results

Bench.	AES	SPMV	KMP	FFT
Avg. err.	13.5%	9.5%	12.2%	0.1%
Bench.	VITERBI	NW	STENCIL	GEMM
Avg. err.	2.1%	1.1%	7.7%	3.3%

OpenMP-based multicore CPU implementations, all of which are normalized to the performances of the corresponding single-core software implementations. We can clearly see that AutoAccel-generated accelerators outperform the naive implementations by 27,000×, indicating that AutoAccel dramatically improves the quality of accelerator designs without manual programming effort. Meanwhile, the AutoAccel-generated accelerators also outperform the single-core software implementations by 72×, indicating that our approach does lead to competitive accelerator designs. Even compared with the OpenMP-based multicore (12 cores in our experimental platform) implementations, the FPGA accelerators are still competitive. The multicore implementations win the FPGA accelerators in merely two cases: SPMV and FFT. We can clearly see that these two cases are the ones that the accelerators have the least amounts of speedups, 1.6x and 3.8x over their single-core counterparts, respectively. Both SPMV and FFT are floating-point intensive and require large memory bandwidths, which makes the multicore implementations the favorite.

We can also see that the manual designs only outperform the AutoAccel-generated designs by an average 2.5×, even after we spent several days to weeks performing more sophisticated code reconstruction to each kernel. In fact, the AutoAccel-generated designs for the `AES`, `SPMV`, `KMP` and `STENCIL` kernels have already achieved the optimal performance since they have fully utilized the off-chip bandwidth. Al-

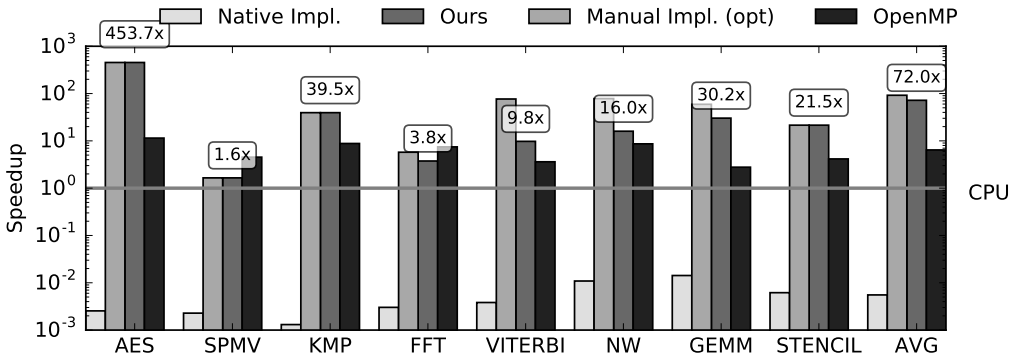


Figure 4.4: Speedup over an Intel Xeon CPU Core

though we are able to further improve the performance of other kernels by manually applying very specialized circuit designs not covered by AutoAccel, e.g., Race Logic [MSS14] for the NW kernel, AutoAccel still preserves a high quality of results while substantially reducing the programming effort.

Finally, we analyze the energy efficiency gain of AutoAccel-generated designs. We estimate the energy efficiency (performance per watt) of our experiments by considering execution time and thermal design power (TDP). The TDP of the Intel Xeon CPU and the Xilinx FPGA used in this comparison is 80W and 25W, respectively. Accordingly, AutoAccel-generated designs can achieve up to a 1677.9× energy efficiency improvement, and 260.4× on average.

4.6 Conclusion

While the CPU-FPGA heterogeneous architectures are becoming a promising paradigm for providing continued performance and energy improvement in modern datacenters, accelerator programming arises as a serious challenge to application

developers. Based on our analysis study and the best-effort code reconstruction practice, we propose the AutoAccel framework to provide a nearly push-button experience on mapping C functions into high-quality FPGA accelerator designs for general datacenter application developers. Featuring the CPP microarchitecture, analytical-based design space exploration and automatic code transformation, AutoAccel achieves 72× speedup for a broad class of computation kernels.

Furthermore, we believe that the design principles of AutoAccel can be further generalized to stimulate more research on the adoption of FPGAs in datacenters. The CPP microarchitecture serves as a proof-of-concept that using accelerator design templates as specifications of the program-to-behavioral-description transformation fundamentally reduces the design space while preserving the accelerator quality. Future work could be to support more microarchitectural templates and more sophisticated code transformation techniques to improve the coverage of computation kernels. Hence, more microarchitectures, with their analytical models and code transformation techniques, might be added in AutoAccel to improve the coverage of computation kernels. Also, more sophisticated, high-abstract code transformations (e.g., loop permutation) are able to be supported in the future, along with polyhedral analysis, to form a larger design space and create more optimization opportunities.

CHAPTER 5

CPU-FPGA Integration

In this chapter, we focus on the issues in the integration of FPGA accelerators into conventional CPU systems, with a key focus on the JVM-FPGA integration since many prevalent datacenter programming frameworks are based on the Java Virtual Machine (JVM) [PHA17]. We start from our application showcase, the acceleration for the genome sequencing application, and present the host program and the FPGA accelerator design in Section 5.1. Based on the issues found in the integration, we then present our quantitative analysis study on the microarchitectures of five state-of-the-art CPU-FPGA platforms (Section 5.2). The study reveals three key factors that affect the efficiency of the integration. We then get back to the application showcase, and present our solutions for each of these three factors (Section 5.3 and 5.4).

5.1 Genome Sequencing Acceleration: The Story Begins

5.1.1 Overview

With the ever-growing volume, variety and velocity of data, scaling out big-data computation into a datacenter scale has attracted increasing attention from both academia and industry. There has been great success in programming frameworks

that enable efficient development and deployment of large-scale applications in conventional datacenters composed of general-purpose processors. Examples include the pioneering MapReduce framework [DG08] initially proposed by Google, the open-source Hadoop MapReduce framework [Whi12], and the more recent Apache Spark framework [ZCD12] that improves the performance of Hadoop by up to 100× through in-memory cluster computing.

Meanwhile, the power and energy efficiency of general-purpose processors have become two of the primary constraints that limit the performance scaling of conventional datacenters. To provide continued performance and energy efficiency improvement in datacenters, harnessing FPGA-based heterogeneous platforms is considered one of the most promising approaches since FPGAs provide low power and high energy efficiency, as well as reprogrammability. With the emerging FPGA-enabled datacenter trend, one key question is: *how can we efficiently integrate FPGAs into state-of-the-art big-data computing frameworks like Spark?* Our goal is to provide a comprehensive analysis of challenges and generalized insights for efficient integration of FPGA accelerators into the widely used big-data computing framework, Spark. Our approach is to conduct an in-depth case study for the acceleration of an important and representative application: next-generation DNA sequencing [SJ08][1].

We choose the next-generation DNA sequencing application as a representative case study for the following reasons. First, it is a very important application that has been widely used in medical and biological research and is transitioning into real clinical use where sequencing time is a matter of life and death. Second, it is a real-world big-data application that needs the power of cluster-scale comput-

[1]This study is presented in [CCF16a]. I would like to convey my appreciation to all coauthors for their contributions to this study.

ing. Even a single human genome occupies more than 300GB data and aligning it to the golden reference genome—a key step in sequencing that we focus on—using state-of-the-art aligners like BWA-MEM [Li13] on a modern multicore server takes tens of hours. Third, there are already available cloud-scale solutions (e.g., CS-BWAMEM [CCL15b]) using CPU-only clusters and pure FPGA accelerator solutions (e.g., [CCL15a]). Therefore, we can leverage these existing solutions and focus on the integration part not well studied in prior work. Finally and most importantly, the FPGA accelerator for this application represents a category of fine-grained accelerators that impose further challenges for the integration. Unlike conventional coarse-grained accelerators, these fine-grained accelerators execute for a very short time (e.g., a microsecond or so), and thus a straightforward offloading of the CPU computation onto the FPGA could significantly degrade the overall performance due to the overwhelming JVM-FPGA communication overhead (e.g., a few milliseconds for data to be transferred from JVM to native machine and then to FPGA).

To provide more general insights into efficient Spark-FPGA integration methodology, throughout this case study we systematically analyze the integration challenges at three levels.

- **Single-thread level**. First, state-of-the-art big-data computing frameworks like Spark rely on Java Virtual Machines (JVMs) for ease of deployment; while FPGA accelerators are typically manipulated by C/C++ programs. To enable JVMs to manipulate FPGA accelerators, we leverage the support of Java Native Interface (JNI) [Lia99]. Second, although JNI can make the integration work, a straightforward integration for the fine-grained FPGA accelerators could actually slow down the overall system performance due to the overwhelm-

ing JVM-FPGA communication overhead.

- **Single-node multi-thread level**. The first challenge is how to efficiently share an FPGA accelerator among multiple CPU threads. Second, more threads impose more thread contention and increasing memory pressure, which puts more constraint.

- **Multi-node level**. Multi-nodes further introduce potential inter-node communication overhead that may hurt the system performance. In general, inter-FPGA communication is rarely needed in a Spark-FPGA integration. The reason is that an FPGA accelerator is usually designed for a computational kernel that resides in a Spark program's map function, which inherently has no communication with other map functions. Nevertheless, in our case study, we do observe a system-wide communication overhead when broadcasting massive read-only input data, which is caused by Spark's broadcasting mechanism instead of the Spark-FPGA integration.

In this section, we mainly focus on the identification of the problem and its quantitative impact. The findings motivate us to perform the CPU-FPGA analysis study to find the key factors of integration, and our proposed techniques to address them.

5.1.2 Experimental Setup

Before presenting challenges and solutions with quantitative analysis for efficient Spark and FPGA integration, we first describe the software and hardware setup of our experimental system used throughout the study. Our experimental system

comprises a cluster of 1 master node and 6 worker nodes, as shown in Fig. 5.1. Except for the master node of the Spark framework, all Spark's worker nodes are equipped with a PCIe-attached Alpha Data ADM-PCIE-7V3 FPGA board [Xil17]. Table 5.1 lists the detailed configuration of each server.

Table 5.1: Experimental setup

Host CPU	two 6-core Xeon E5-2620v3@2.40GHz
Host Memory	48GB DDR3-1600
FPGA Fabric	Xilinx Virtex 7@200MHz
CPU ↔ FPGA	PCIe Gen3 x8, 8GB/s as advertised
FPGA Device Memory	16GB DDR3-1600
Development Environment	SDAccel 2017.1

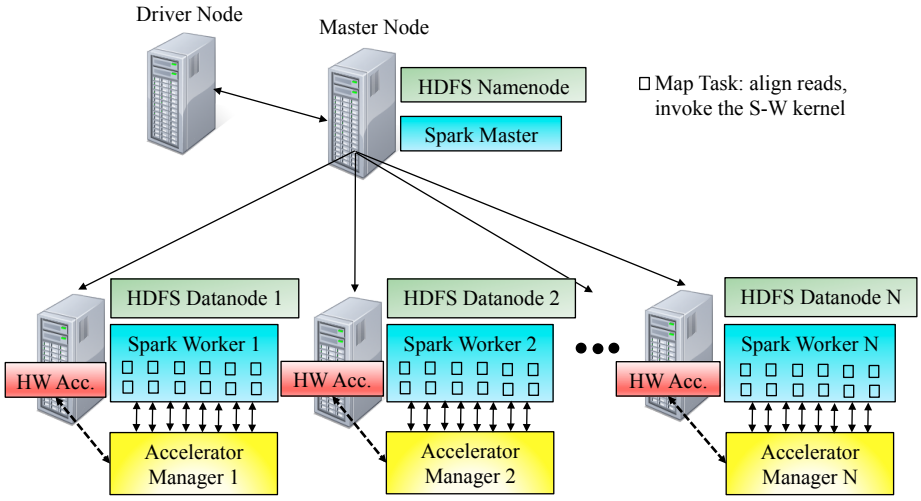


Figure 5.1: An overview of the Spark-FPGA cluster

Software configuration. We use Spark 1.5.1 as our cluster computing framework and HDFS 2.5.2 as our underlying distributed file system, and run CS-BWAMEM 0.2.2 on top of them. As illustrated in Fig. 5.1, each read is aligned by

a CS-BWAMEM's map function that invokes the Smith-Waterman kernel. Our test cases are derived from the genome sample of a human with breast cancer (HCC1954 [GKV98]). The sample contains almost 1 billion reads, each with 101 nucleotides (denoted by a 101-character ASCII string).

Accelerator configuration. We adopt the FPGA accelerator for the Smith-Waterman algorithm proposed in [CCL15a] since it is dedicated to the customized Smith-Waterman kernel in BWA and CS-BWAMEM, and no change of the accelerator is needed. We configure the accelerator to contain 40 processing elements (PEs), and it is synthesized using the Xilinx SDAccel Development Environment [sda]. Without considering any communication overhead, in our system, we observe around 120× and 10.5× speedup for the FPGA Smith-Waterman accelerator itself, compared to the single-thread CPU and 16-thread CPU (the best performance setting with hyper-threading, according to Section 5.3.2), respectively.

Profiling methodology. The distributed computing nature of Spark applications makes them relatively harder to profile compared to single-thread applications. In order to conduct a quantitative analysis, we implement an embedded MapReduce profiler along with CS-BWAMEM. In detail, we embed a profiling *map* function inside CS-BWAMEM's original map function to collect profiling data, and embed a profiling *reduce* function inside CS-BWAMEM's original reduce function to accumulate each measurement across all map function calls. To compare the performance of different implementations, we use the notation of "kilo reads per second (KRPS)," which is referred to as the number of kilo-reads processed in a second.

For convenience, we will denote the original CS-BWAMEM program as CS-BWAMEM/CPU, and the CS-BWAMEM program with the Smith-Waterman ac-

celerator as CS-BWAMEM/FPGA in the rest of this dissertation. CS-BWAMEM will be also used in the scenarios where both CS-BWAMEM/CPU and CS-BWAMEM/FPGA fit.

5.1.3 Harnessing FPGA in JVM

To harness FPGAs' low power and high energy efficiency in a Spark cluster, we have to enable Spark to manipulate FPGA accelerators, i.e., make JVMs and FPGAs work together. To make things easier, we start with a single-thread integration by leveraging the Java Native Interface (JNI) to connect a JVM instance with an FPGA accelerator. Nevertheless, a straightforward JNI integration for fine-grained FPGA accelerators could actually slow down the overall system performance. A quantitative analysis reveals a preliminary reason: the data transfer overhead from a JVM to a native machine (through JNI) and then to an FPGA (through PCIe) could overwhelm the benefits of short-executed FPGA accelerators. The following text presents the problem and the preliminary reasoning.

Spark programs are mainly written in Java and/or Scala, and run on JVMs. FPGA accelerators are typically manipulated by C/C++ programs, and JVMs do not support the use of FPGAs in default. Therefore, the first challenge in the single-thread level is essentially to bridge the gap between Java/Scala and C/C++.

Our proposed approach leverages JNI to enable Spark programs to indirectly invoke FPGA accelerators from JVM. JNI is a programming interface that empowers Java/Scala applications to call and be called by applications written in other languages. We create a C-based native library that uses the OpenCL APIs provided by Xilinx SDAccel to manipulate the Smith-Waterman accelerator, and modify

CS-BWAMEM/CPU's map function to call the library whenever the map function invokes the Smith-Waterman kernel.

The above approach produces a working solution to the Spark-FPGA integration problem, and could be relatively efficient if the computation time of each accelerator invocation is long enough (i.e., coarse-grained accelerators as discussed in prior work) to ignore the JVM-FPGA communication overhead. However, in our case study, the fine-grained Smith-Waterman FPGA accelerator executes for a very short time (a microsecond or so) but will be invoked hundreds of millions of times. This straightforward integration significantly degrades the overall performance. We compare the performance of CS-BWAMEM/CPU and CS-BWAMEM/FPGA, and find that CS-BWAMEM/CPU achieves 2.1 KRPS (kilo reads per second) while CS-BWAMEM/FPGA, with the straightforward Spark-FPGA integration, reaches merely 1.6×10^{-3} RPS. In other words, the straightforward integration does not fulfill the 120× speedup, but instead decreases the overall performance by three orders of magnitude.

After a quantitative analysis, we find that the main reason for the performance degradation is the tremendous JVM-FPGA communication overhead aggregated through all the invocations of the Smith-Waterman accelerator. To be specific, one read produces 24 Smith-Waterman invocations (either software or hardware implementation) on average, and it takes about 480μs for the software to process them in JVM. That is, each Smith-Waterman invocation of the software version should cost no more than 20μs on average. Meanwhile, a complete routine of a Smith-Waterman accelerator invocation involves: 1) data copy between a JVM and a native machine, 2) DMA transfer between a native machine and an FPGA board though PCIe, and 3) computation on the FPGA board. The communication process, including 1) and

2), costs over 25ms per invocation. That is, even if an accelerator could reduce the computation time of the Smith-Waterman kernel down to 0, the communication overhead would degrade the performance by 1000×.

5.1.4 Conclusion

We have now identified the most critical issue to address in the JVM-FPGA integration process. Given a high-performance application and an accelerator, we surprisingly obtain a 1000× system-wide slowdown. Although the preliminary analysis explains to us that it is the extra JVM-FPGA data communication overhead that is going to be blamed. However, we need more deep understanding on the CPU-FPGA communication so as to actually find a way to resolve this overhead. In the following section, we conduct a quantitative analysis on five state-of-the-art CPU-FPGA platforms to acquire a better understanding. Based on this, we propose our techniques for an efficient integration, and continue from the breakpoint to improve our integration results.

5.2 The Mystery of CPU-FPGA Communication

5.2.1 Overview

With the trend of adopting FPGAs in datacenters, various CPU-FPGA acceleration platforms with diversified microarchitectural features have been developed. We classify state-of-the-art CPU-FPGA platforms in Table 5.2 according to their physical integration and memory models. Traditionally, the most widely used integration is to connect an FPGA to a CPU via the PCIe interface, with both components

equipped with private memories. Many FPGA boards built on top of Xilinx or Intel FPGAs use this way of integration because of its extensibility. The customized Microsoft Catapult board integration is such an example. Another example is the Alpha Data FPGA board [Xil17] with the Xilinx FPGA fabric that can leverage the Xilinx SDAccel development environment [sda] to support efficient accelerator design using high-level programming languages, including C/C++ and OpenCL. The Amazon F1 instance also adopts this software/hardware environment to allow high-level accelerator design. On the other hand, vendors like IBM tend to support a PCIe connection with a coherent, shared memory model for easier programming. For example, IBM has been developing the Coherent Accelerator Processor Interface (CAPI) on POWER8 [SBJ15] for such an integration, and has used this platform in the IBM data engine for NoSQL [BRH15]. Meanwhile, the CCIX consortium has proposed the Cache Coherent Interconnect for Accelerators which can connect FPGAs with ARM processors through the PCIe interface with coherent shared memory as well [cci18]. More recently, closer CPU-FPGA integration becomes available using a new class of processor interconnects such as Front-Side Bus (FSB) and the newer QuickPath Interconnect (QPI), and provides a coherent, shared memory, such as the FSB-based Convey machine [Bre10] and the Intel Xeon+FPGA Accelerator Platform [harb]. While the first generation of the Xeon+FPGA platform (Xeon+FPGA v1) connects a CPU to an FPGA only through a coherent QPI channel, the second generation of the Xeon+FPGA platform (Xeon+FPGA v2) adds two non-coherent PCIe data communication channels between the CPU and the FPGA, resulting in a hybrid CPU-FPGA communication model.

The evolution of various CPU-FPGA platforms brings up a challenging question: which platform should we choose to gain better performance and energy efficiency

Table 5.2: Classification of modern CPU-FPGA platforms

	Separate Private Memory	Shared Memory
PCIe Peripheral Interconnect	Alpha Data [Xil17], Microsoft Catapult [PCC14], Amazon F1 [amab]	IBM CAPI [SBJ15], CCIx [cci18]
Processor Interconnect	N/A	Intel Xeon+FPGA v1 [harb] (QPI), Convey HC-1 [Bre10] (FSB)
Hybrid	N/A	Intel Xeon+FPGA v2 (QPI, PCIe)

for a given application to accelerate? There are numerous factors that can affect the choice, such as platform cost, programming models and efforts, logic resource and frequency of FPGA fabric, CPU-FPGA communication latency and bandwidth, to name just a few. While some of them are easy to figure out, others are nontrivial, especially the communication latency and bandwidth between CPU and FPGA under different integration. One reason is that there are few publicly available documents for the newly announced platforms like the Xeon+FPGA family, CAPI and Amazon F1 instance. More importantly, those architectural parameters in the datasheets are often advertised values, which are usually difficult to achieve in practice. Actually, sometimes there could be a huge gap between the advertised numbers and practical numbers. For example, the advertised bandwidth of the PCIe Gen3 x8 interface is 8GB/s; however, our experimental results show that the PCIe-equipped Alpha Data platform can only provide 1.6GB/s PCIe-DMA bandwidth using OpenCL APIs implemented by Xilinx (see Section 5.2.2). Quantitative evaluation and in-depth analysis of such kinds of microarchitectural characteristics could aid CPU-FPGA platform users to accurately predict the performance of a computation kernel to accelerate on various candidate platforms, and make the right choice. Furthermore, it could also benefit CPU-FPGA platform designers for identifying performance bottlenecks and

providing better hardware and software support.

Motivated by those potential benefits to both platform users and designers, this study aims to discover what microarchitectural characteristics affect the performance of modern CPU-FPGA platforms, and evaluate how they will affect that performance. We conduct our quantitative comparison on five state-of-the-art CPU-FPGA platforms: 1) the Alpha Data board and 2) Amazon F1 instance that represent the conventional PCIe-based platform with private device memory, 3) IBM CAPI that represents the PCIe-based system with coherent shared memory, 4) Intel Xeon+FPGA v1 that represents the QPI-based system with coherent shared memory, and 5) Xeon+FPGA v2 that represents a hybrid PCIe (non-coherent) and QPI (coherent) based system with shared memory[2]. These five platforms cover various CPU-FPGA interconnection approaches, and different memory models as well.

In summary, this study makes the following contributions.

- The first quantitative characterization and comparison on the microarchitectures of state-of-the-art CPU-FPGA acceleration platforms—including the Alpha Data board and Amazon F1 instance, IBM CAPI, and Intel Xeon+FPGA v1 and v2—which covers the whole range of CPU-FPGA connections. We quantify each platform's CPU-FPGA communication latency and bandwidth and the results are summarized in Fig. 5.2.

- An in-depth analysis of the big gap between advertised and practically achievable performance (Section 5.2.2), with step-by-step decomposition of the inefficiencies.

[2]This study is presented in [CCF16b]. I would like to convey my appreciation to all coauthors for their contributions to this study.

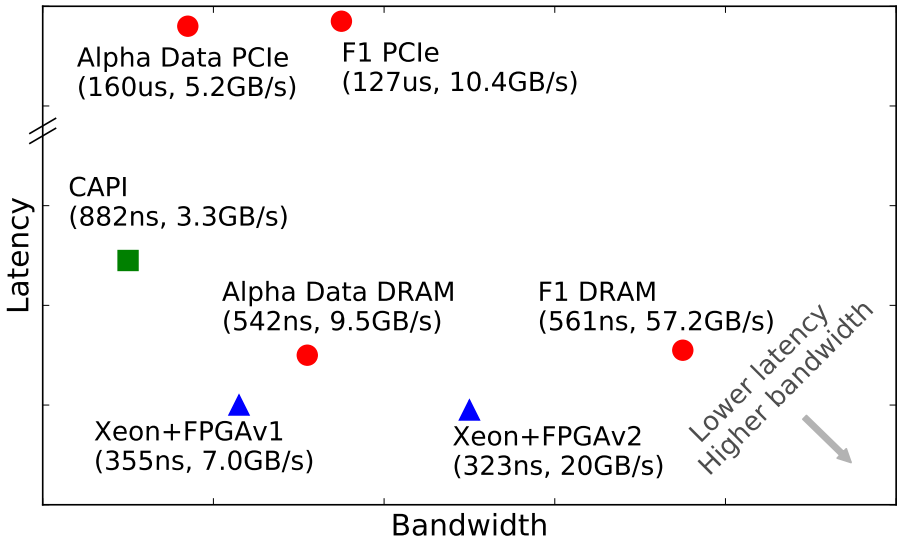


Figure 5.2: Summary of CPU-FPGA communication bandwidth and latency (not to scale)

- Seven insights for both application developers to improve accelerator designs and platform designers to improve platform support (Section 5.2.3). Specifically, we suggest accelerator designers avoid using the advertised platform parameters to estimate the acceleration effect, which almost always leads to (*extremely*) overly optimistic estimation. Moreover, we analyze the trade-off between private-memory and shared-memory platforms, and analytically model the effective bandwidth with the introduction of the memory data reuse ratio r. We also propose the metric of computation-to-communication (CTC) ratio to measure when the CPU-FPGA communication latency and bandwidth are critical. Finally, we suggest that the complicated communication stack and hard-to-use coherent cache system may subject to improve in the next-generation of CPU-FPGA platforms.

5.2.2 Characterization of CPU-FPGA Microarchitectures

This work aims to reveal how the underlying microarchitectures, i.e., processor or peripheral interconnect, and shared or private memory model, affect the performance of CPU-FPGA platforms. To achieve this goal, in this section, we quantitatively study those microarchitectural characteristics, with a key focus on the effective bandwidth and latency of CPU-FPGA communication on five state-of-the-art platforms: Alpha Data, F1 instance, CAPI, Xeon+FPGA v1 and v2.

To measure the CPU-FPGA communication bandwidth and latency, we design and implement our own microbenchmarks, based on the Xilinx SDAccel SDK 2017.4 [sda] for Alpha Data and F1 instance, Alpha Data CAPI Design Kit [IBM15] for CAPI, and Intel AALSDK 5.0.3 [Int16] for Xeon+FPGA v1 and v2. Each microbenchmark consists of two parts: a host program and a computation kernel. Following each platform's typical programming model, we use the C language to write the host programs for all platforms, and describe the kernel design using OpenCL for Alpha Data and F1 instance, and Verilog HDL for the other three platforms.

The hardware configurations of Alpha Data, F1 instance, CAPI, Xeon+FPGA v1 and v2 in our study are listed in Table 5.3.

5.2.2.1 Effective Bandwidth

Effective Bandwidth for Alpha Data. Traditional CPU-FPGA platforms like Alpha Data contain two communication phases: 1) PCIe-based direct memory access (DMA) between host memory and device memory, and 2) device memory access. We measure the effective bandwidths with various payload sizes for both phases. The measurement results are illustrated in Fig. 5.3. Since the bandwidths for both

Table 5.3: Platform configurations of Alpha Data, F1, CAPI, Xeon+FPGA v1 and v2

Platform	Alpha Data	CAPI	Amazon EC2 F1	Xeon+FPGA v1	Xeon+FPGA v2
Host CPU	Xeon E5-2620v3 @2.40GHz	Power8 Turismo @4GHz	Xeon E5-2686v4 @2.3GHz	Xeon E5-2680v2 @2.80GHz	Xeon E5-2600v4 @2.4GHz
Host Memory	64GB DDR3-1600	16GB DDR3-1600	64GB DDR4-2133	96GB DDR3-1600	64GB DDR4-2133
FPGA Fabric	Xilinx Virtex 7 @200MHz	Xilinx Virtex 7 @250MHz	Xilinx UltraScale+ @250MHz	Intel Stratix V @200MHz	Intel Arria 10 @400MHz‡
CPU ↔ FPGA	PCIe Gen3 x8, 8GB/s	PCIe Gen3 x8, 8GB/s	PCIe Gen3 x16, 16GB/s	Intel QPI, 12.8GB/s	1× Intel QPI & 2× PCIe Gen3 x8
Device Memory	16GB DDR3-1600	16GB DDR3-1600†	64GB DDR4-2133	N/A	N/A

† The device memory in CAPI is not used in this work.

‡ The user clock can be easily configured to 137/200/273 MHz using the supplied SDK, in addition to max 400MHz frequency.

directions of PCIe-DMA transfer are almost identical (less than 4% difference), we only present the unidirection PCIe-DMA bandwidth in Fig 5.3.

While Fig. 5.3 illustrates a relatively high private DRAM bandwidth (9.5GB/s for read, 8.9GB/s for write[3]); the PCIe-DMA bandwidth (1.6GB/s) reaches merely 20% of PCIe's advertised bandwidth (8GB/s). That is, the expectation of a high DMA bandwidth with PCIe is far away from being fulfilled.

The first reason is that there is non-payload data overhead for the useful payload transfer [Law14]. In a PCIe transfer, a payload is split into small packets, each packet equipped with a header. Along with the payload packets, there are also a large number of packets for control purposes transferred through PCIe. As a result, the maximum supplied bandwidth for the actual payload, which we call as the *theoretical* bandwidth, is already smaller than the advertised value.

Another important reason is that a PCIe-DMA transaction involves not only PCIe transfer, but also host buffer allocation and host memory copy [Coo12]. The

[3]If not specifically indicated, the bandwidth appearing in the rest of this paper refers to the maximum achievable bandwidth.

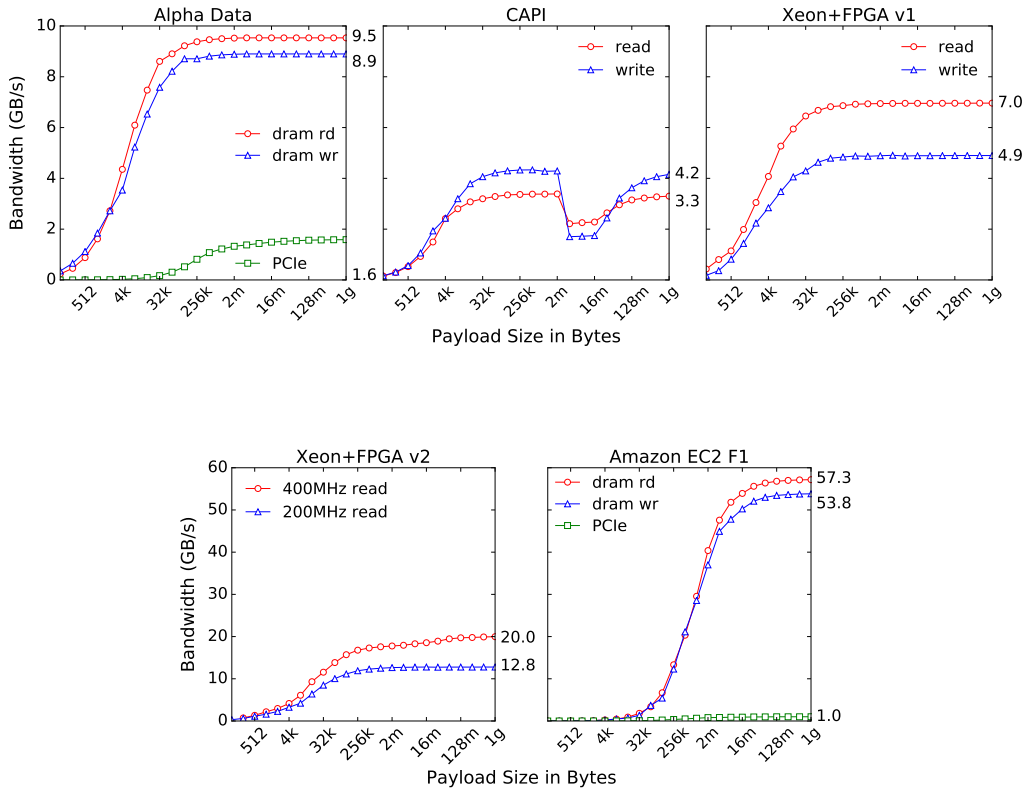
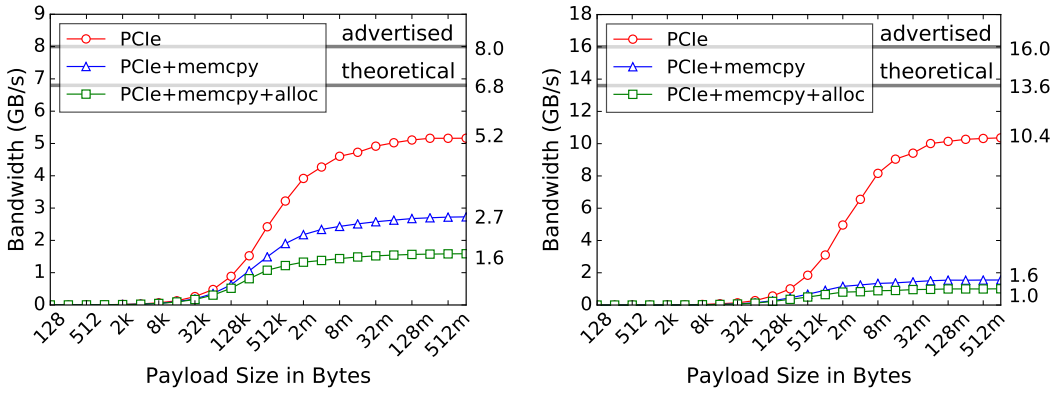


Figure 5.3: Effective bandwidth of Alpha Data, CAPI, Xeon+FPGA v1 and v2, and F1

host memory stores user data in a pageable (unpinned) space from which the FPGA cannot directly retrieve data. A page-locked (pinned), physically contiguous memory buffer in the operating system kernel space then serves as a staging area for PCIe transfer. When a PCIe-DMA transaction starts, a pinned buffer is first allocated in the host memory, followed by a memory copy of pageable data to this pinned buffer. The data is then transferred from the pinned buffer to device memory through PCIe. These three steps—buffer allocation, host memory copy and PCIe transfer—are sequentially processed in Alpha Data, which significantly deceases the PCIe-DMA



(a) Alpha Data (b) Amazon EC2 F1

Figure 5.4: PCIe-DMA bandwidth breakdown

bandwidth.

Moreover, there could be some implementation deficiencies in the vendor-provided environment which serve as another source of overhead[4]. One possibility could be the data transfer overhead between the endpoint of the PCIe channel, i.e., the vendor-provided FPGA DMA IP, and the FPGA-attached device DRAM. Specifically, the device DRAM does not directly connect to the PCIe channel; instead, the data from the host side first reach the on-chip DMA IP, and then are written into the device DRAM through the vendor-provided DRAM controller IP. If this extra step were not well overlapped with the actual data transfer via the PCIe channel through careful pipelining, it would further reduce the effective bandwidth. Our experiments do show that a considerable gap still exists between the measured bandwidth and the theoretical value, indicating that the vendor-provided environment could be potentially improved with further performance tuning.

[4]The Xilinx SDAccel environment is close sourced, so we are not able to precisely reason this overhead. The following is our best guess.

Next, we quantitatively evaluate the large PCIe-DMA bandwidth gap step by step, with results shown in Fig. 5.4.

1. The non-payload data transfer lowers the theoretical PCIe bandwidth to 6.8GB/s from the advertised 8GB/s [Law14].

2. Possible implementation deficiencies in the vendor-provided environment prevent the 6.8GB/s PCIe bandwidth from being fully exploited. As a result, the highest achieved effective PCIe-DMA bandwidth without buffer allocation and host memory copy decreases to 5.2GB/s.

3. The memory copy between the pageable and pinned buffers further degrades the PCIe-DMA bandwidth to 2.7GB/s.

4. The buffer allocation overhead degrades the final effective PCIe-DMA bandwidth to only 1.6GB/s. This is the actual bandwidth that end users can obtain.

Effective Bandwidth for CAPI. CPU-FPGA platforms that realize the shared memory model, such as CAPI, Xeon+FPGA v1 and v2, allow the FPGA to retrieve data directly from the host memory. Such platforms therefore contain only one communication phase: host memory access through the communication channel(s). For the PCIe-based CAPI platform, we simply measure the effective read and write bandwidths of its PCIe channel for a variety of payload sizes, as shown in Fig. 5.3.

Compared to the Alpha Data board, CAPI supplies end users with a much higher effective PCIe bandwidth (3.3GB/s vs. 1.6GB/s). This is because CAPI provides efficient API support for application developers to directly allocate and manipulate pinned memory buffers, eliminating the memory copy overhead between the pageable

and pinned buffers. However, Alpha Data's private *local* memory read and write bandwidths (9.5GB/s, 8.9GB/s) are much higher than those of CAPI's shared *remote* memory access. This phenomenon offers opportunities for both platforms. As is discussed in Section 5.2.3, if an accelerator is able to smartly use the private memory as a "shortcut" for accessing the host memory, it will probably obtain a similar or even higher effective CPU-FPGA communication bandwidth in a traditional platform like Alpha Data than in a shared-memory platform like CAPI or Xeon+FPGA.

Another remarkable phenomenon shown in Fig. 5.3 is the dramatic falloff of the PCIe bandwidth at the 4MB payload size. This could be the side effect of CAPI's memory coherence mechanism. CAPI shares the last-level cache (LLC) of the host CPU with the FPGA, and the data access latency varies significantly between LLC hit and miss. Therefore, one possible explanation for the falloff is that CAPI shares 2MB of the 8MB LLC with the FPGA. The payloads of which the sizes are not larger than 2MB can fit into LLC, resulting in a low LLC hit latency that can be well amortized by a few megabytes of data. Nevertheless, when the payload size grows to 4MB and cannot fit into LLC, the average access latency of the payload data will suddenly increase, leading to the observed falloff. With the payload size further growing, this high latency is gradually amortized and the PCIe bandwidth gradually reaches the maximum value.

Effective Bandwidth for Xeon+FPGA v1. The CPU-FPGA communication of Xeon+FPGA v1 involves only one step: host memory access through QPI; therefore, we just measure a set of effective read and write bandwidths for different payload sizes, as shown in Fig. 5.3. We can see that both the read and write bandwidths (7.0GB/s, 4.9GB/s) are much higher than the PCIe bandwidths of Alpha Data and CAPI. Therefore, the QPI-based CPU-FPGA integration does demonstrate a higher

effective bandwidth than the PCIe-based integration. However, the *remote* memory access bandwidths of Xeon+FPGA v1 are still lower than those of Alpha Data's *local* memory access. That is, similar with CAPI, Xeon+FPGA v1 can possibly be outperformed by Alpha Data if an accelerator keeps reusing the data in the device memory as a "shortcut" for accessing the host memory.

We need to mention that Xeon+FPGA v1 provides a 64KB cache on its FPGA chip for coherency purposes [harb]. Each CPU-FPGA communication will first go through this cache and then go to the host memory if a cache miss happens. Therefore, the CPU-FPGA communication of Xeon+FPGA v1 follows the classic cache access pattern. Since the bandwidth study mainly focuses on large payloads, our microbenchmarks simply flush the cache before accessing any payload to ensure all requests go through the host memory. The bandwidths illustrated in Fig. 5.3 are, more accurately, miss bandwidths. Section 5.2.2.2 discusses the cache behaviors in detail.

Effective Bandwidth for Xeon+FPGA v2. While its CPU-FPGA communication involves only one phase as well, Xeon+FPGA v2 allows the user accelerator to work on different frequencies. Therefore, we measure the effective bandwidths of various sizes of payloads under 200MHz and 400MHz, respectively. 200MHz is the frequency that is also used by Alpha Data and Xeon+FPGA v1; 400MHz is the maximal frequency supported by Xeon+FPGA v2. Note that this change does not affect the frequency of the internal logic of the communication channels, but just the interface between the channels and the user accelerator. Fig. 5.3 illustrates the measurement results. We can see that Xeon+FPGA v2 outperforms the aforementioned three CPU-FPGA platforms in terms of effective bandwidth (20GB/s at 400MHz, 12.8GB/s at 200MHz). This is due to the fact that Xeon+FPGA v2 connects the

CPU and the FPGA through three communication channels: one QPI channel and two PCIe channels, resulting in a significantly high aggregate bandwidth.

Same as its first generation, Xeon+FPGA v2 also provides a 64KB on-chip cache for coherency purposes. However, this cache maintains coherence only for the QPI channel, and the two PCIe channels are of no coherence properties. As a consequence, Xeon+FPGA v2 actually delivers a partially-coherent memory model. We will discuss the cache behaviors of the Xeon+FPGA family in Section 5.2.2.2, and the coherence issue in Section 5.2.3.

Effective Bandwidth for F1 Instance. The Amazon EC2 F1 instance represents the state-of-the-art advance of the canonical PCIe-based platform architecture. Same as the Alpha Data board, it connects the CPU with the FPGA through the PCIe channel, with private memory attached to both components. It is powered by the Xilinx SDAccel environment as well to achieve the behavior-level hardware accelerator development. Both the PCIe channel and the private DRAM are upgraded to supply higher bandwidths. However, the end-to-end bandwidth delivered to the end user is rather surprising. As illustrated in Fig. 5.3, the effective PCIe bandwidth of the F1 instance turns out to be even worse than that of the Alpha Data board which adopts an old-generation technique. The breakdown in Fig. 5.4 shows that the PCIe bandwidth does become doubled without the consideration of the buffer allocation and memory copy overhead. In other words, the buffer allocation and memory copy impose more overhead on the F1 instance than on the Alpha Data board. It might be due to the virtualization overhead of the F1 instance.

Recall the phenomenon that the CPU-FPGA bandwidth of Xeon+FPGA v1 lies between the PCIe bandwidth and the private device DRAM bandwidth of the

Alpha Data board, which provides opportunities for both platforms. The F1 instance and Xeon+FPGA v2 form another (more advanced) pair of CPU-FPGA platforms that follows such a relation. As discussed in Section 5.2.3, we expect that this relation between a private-memory platform and a shared-memory platform will continue to exist in future CPU-FPGA platform advances, and thus provide opportunities for both kinds.

5.2.2.2 Effective Latency

Coherent Cache Behaviors. As described in Section 5.2.2.1, the QPI channel of the Xeon+FPGA family includes a 64KB cache for coherence purposes, and the QPI-based communication thus falls into the classic cache access pattern. A cache transaction is typically depicted by its hit time and miss penalty. We follow this traditional methodology for cache study and quantify the hit time and miss latency of the Xeon+FPGA coherent cache, as shown in Table 5.4.

Table 5.4: CPU-FPGA access latency in Xeon+FPGA

Access Type	Latency (ns)
Read Hit	70
Write Hit	60
Read Miss	avg: 355
Write Miss	avg: 360

A noteworthy phenomenon is the long hit time – 70ns (14 FPGA cycles) for read hit and 60ns (12 FPGA cycles) for write hit – in this 64KB cache. We investigate this phenomenon by decomposing the hit time into three phases — address translation, cache access and transaction reordering — and measuring the elapsed time of each

phase, as shown in Table 5.5.

Table 5.5: Hit latency breakdown in Xeon+FPGA

Access Step	Read Latency (ns)	Write Latency (ns)
Address Translation	20	20
Cache Access	35	35
Transaction Reordering	15	5

The data demonstrate a possibly exorbitant price (up to 100% extra time) paid for address translation and transaction reordering. Worse still, the physical cache access latency is still prohibitively high — 35ns (7 FPGA cycles). Given this small but long-latency cache, it is extremely hard, if not impossible, for an accelerator to harness the caching functionality.

It is worth noting that the Xeon+FPGA v2 platform supports higher clock frequencies than its first generation, and thus can potentially lead to a lower cache access latency. However, this does not fundamentally change the fact that the latency of accessing the coherent cache is still much longer than that of accessing the on-chip BRAM blocks. Therefore, Xeon+FPGA v2 does not fundamentally improve the usability of the coherent cache. As is discussed in Section 5.2.3, we still suggest accelerator designers sticking to the conventional FPGA design principle that explicitly manages the on-chip BRAM resource.

Communication Channel Latencies. We now compare the effective latencies among the PCIe transfer of Alpha Data, F1 instance and CAPI, the device memory access of Alpha Data and F1 instance, and the QPI transfer of the Xeon+FPGA family[5]. Table 5.6 lists the measured latencies of all five platforms for transferring a

[5]For simplicity, we mainly discuss the CPU-to-FPGA read case, the observation is similar for

single 512-bit cache block (since all of them have the same 512-bit interface bitwidth). We can see that the QPI transfer enjoys orders-of-magnitude lower latency compared to the PCIe transfer, and is even smaller than that of Alpha Data or Amazon F1's private DRAM access. This rather surprising observation is due largely to the implementation of the vendor-provided environment. In particular, Xilinx SDAccel connects the accelerator circuit to the FPGA-attached DRAM through not only the DRAM controller but an AXI interface that is implemented on the FPGA chip. The data back and forth through the AXI interface impose a significant overhead to the effective device DRAM access latency[6], resulting in the fact that the *local* DRAM access latency of Alpha Data is even longer than the *remote* memory access latency of Xeon+FPGA. This phenomenon implies that a QPI-based platform is preferable for applications with fine-grained CPU-FPGA interaction. In addition, we can see that CAPI's PCIe transfer latency is much lower than that of the Alpha Data board. This is due to the fact that the Alpha Data board harnesses the SDAccel SDK that enables accelerator design and integration through high-level programming languages. Such a higher level of abstraction introduces an extra CPU-FPGA communication overhead in processing the high-level APIs.

Table 5.6: Latencies of transferring a single 512-bit cache block

Platform	Alpha Data	CAPI	Xeon+FPGA v1	Xeon+FPGA v2	Amazon EC2 F1
Latency	PCIe: 160μs DRAM: 542ns	882ns	355ns	323ns	PCIe: 127μs DRAM: 561ns

the FPGA-to-CPU write case.

[6]While not being able to perfectly reason the long latency of the Xilinx platforms, we have confirmed with Xilinx that the phenomenon is observed by Xilinx as well and the AXI bus is one of the major causes.

5.2.3 Analysis and Insights

Based on our quantitative studies, we now analyze how these microarchitectural characteristics can affect the performance of CPU-FPGA platforms, and propose seven insights for both platform users to optimize their accelerator designs, as well as platform designers to improve the hardware and software support in future CPU-FPGA platform development.

Insight 1*: application developers should be cautioned use the advertised communication parameters, but measure the practically achievable parameters to estimate the CPU-FPGA platform performance.*

Experienced accelerator designers are generally aware of the data transfer overhead led by the non-payload data, e.g., packet header, checksum, control packets, etc., and expect approximately 10% to 20% or even less bandwidth degradation. Quite often, this results in a significant overestimation of the end-to-end bandwidth, due to the unawareness of the overhead led by the system software stack, like the host-to-kernel memory copy pointed out by this paper. As analyzed in Section 5.2.2.1, the effective bandwidth provided by a CPU-FPGA platform to end users is often *far* worse than the advertised value that reflects the physical limit of the communication channel. For example, the PCIe-based DMA transfer of the Alpha Data board fulfills only 20% of the 8GB/s bandwidth of the PCIe Gen3 x8 channel; the Amazon F1 instance that adopts a more advanced data communication technique delivers an even worse effective bandwidth to the end user. Evaluating a CPU-FPGA platform using these advertised values will probably result in a significant overestimation of the platform performance. Worse still, the relatively low effective bandwidth is not always achievable. In fact, the communication bandwidth for a small size of payload

is up to two orders of magnitude smaller than the maximum achievable effective bandwidth. A specific application may not always be able to supply each communication transaction with a sufficiently large size of payload to reach a high bandwidth. Platform users need to consider this issue as well in platform selection.

Insight 2*: In terms of effective bandwidth, both the private-memory and shared-memory platforms have opportunities to outperform each other. The key metric is the device memory reuse ratio* r.

Bounded by the low-bandwidth PCIe-based DMA transfer, the Alpha Data board generally reaches a lower CPU-FPGA effective bandwidth than that of a shared-memory platform like CAPI or Xeon+FPGA v1. The higher private memory bandwidth, however, does provide opportunities for a private-memory platform to perform better in some cases. For example, given 1GB input data sent to the device memory through PCIe, if the FPGA accelerator iteratively reads the data for a large number of times, then the low DMA bandwidth will be amortized by the high private memory bandwidth, and the effective CPU-FPGA bandwidth will be nearly equal to the private memory bandwidth which is higher than that of the shared-memory platform. Therefore, the data reuse of FPGA's private DRAM determines the effective CPU-FPGA bandwidth of a private-memory platform, and whether it can achieve higher effective bandwidth than a shared-memory platform.

Quantitatively, we define the device memory reuse ratio, r, as:

$$r = \frac{\sum_{dev} S_{dev}}{\sum_{dma} S_{dma}}$$

where $\sum_{dev} S_{dev}$ denotes the aggregate data size of all device memory accesses, and $\sum_{dma} S_{dma}$ denotes the aggregate data size of all DMA transactions between the host

and the device memory.

Then, the effective CPU-FPGA bandwidth for a private-memory platform, $bw_{cpu-fpga}$, can be defined as:

$$bw_{cpu-fpga} = \frac{1}{\frac{1}{r * bw_{dma}} + \frac{1}{bw_{dev}}}$$

where bw_{dma} and bw_{dev} denote the bandwidths of the DMA transfer and the device DRAM access, respectively.

According to the above formula, the larger r is, the higher the effective CPU-FPGA bandwidth is, and the better the performance could possibly be. It is worth noting that since the FPGA on-chip BRAM data reuse is typically important for FPGA design optimization, the above finding suggests that accelerator designers using a private-memory platform need to consider both on-chip BRAM data reuse and off-chip DRAM data reuse. Moreover, by comparing this effective CPU-FPGA bandwidth of a private-memory platform to the DRAM bandwidth of a shared-memory platform, we can get a threshold of device memory reuse ratio, $r_{threshold}$. If the r value of an application is larger than $r_{threshold}$, the private-memory platform will achieve a higher bandwidth; and vice versa. This could serve as an initial guideline for application developers to choose the appropriate platform for a specific application. Also, this phenomenon continues to exist between the next-generation platforms, i.e., the Amazon EC2 F1 instance and Xeon+FPGA v2, meaning that the proposed device memory reuse ratio is not merely applicable to the evaluated platforms in this paper, but provides guidance to the selection of the private-memory and shared-memory CPU-FPGA platforms across generations.

Fundamentally, the device memory reuse ratio quantifies the trade-off between private-memory and shared-memory platforms based on the following two ground truths. First, local memory access tends to be faster than remote memory access. Using the same technology, the private-memory platform can always come up with a higher device memory access bandwidth compared to the shared-memory platform which retrieves data from the CPU-attached memory. Second, the end-to-end CPU-FPGA data transfer routine of the shared-memory platform is a subset of that of the private-memory platform. Specifically, the routine of the private-memory platform contains data transfers 1) from CPU-attached memory to FPGA, 2) from FPGA to FPGA-attached memory, and 3) from FPGA-attached memory to FPGA, where the shared-memory platform performs only the first step. While both of them are well known and do not need to be reiterated, the proposed device memory reuse ratio serves as a way to utilize them for helping application developers choose the right platform. Furthermore, since these truths will not change over time, we expect that the trade-off between these two kinds of platforms will still exist and the device memory reuse ratio will remain to be a critical parameter.

Insight 3*: In terms of effective latency, the shared-memory platform generally outperforms the private-memory platform, and the QPI-based platform outperforms the PCIe-based platform.*

As shown in Table 5.6, the shared-memory platform generally achieves a lower communication latency than the private-memory platform with the same communication technology (CAPI vs Alpha Data). This is due to the fact that the private-memory platform first caches the data in its device memory, and then allows the FPGA to access to the data, resulting in a longer communication routine. This advantage, together with easier programming model, motivates the new trend of

CPU-FPGA platforms with a PCIe connection and coherent shared memory, such as CAPI and CCIx.

Meanwhile, compared to the PCIe-based platform, the QPI-based platform brings the FPGA closer to the CPU, leading to a lower communication latency. Therefore, a QPI-based, shared-memory platform is preferred for latency-sensitive applications, especially those that require frequent (random) fine-grained CPU-FPGA communication. Some examples like high-frequency trading (HFT), online transaction processing (OLTP), or autonomous driving might benefit from the low communication latency of the QPI channel. Compared to Xeon+FPGA systems, the major drive of PCIe-based shared memory system is its extensibility for more FPGA boards in larger scale systems.

Insight 4*: CPU-FPGA communication is critical to some applications, but not all. The key metric is the computation to communication ratio* CTC*.*

Double buffering and dataflow are well-used techniques in accelerator design optimizations. Such techniques can realize a coarse-grained data processing pipeline so as to overlap the computation and data communication processes. As a result, the performance of the FPGA accelerator is generally bounded by the coarse-grained pipeline stage that consumes more time. Based on this criterion, FPGA accelerators can be roughly categorized into two classes: 1) computation-bounded ones where the computation stage takes longer time, and 2) communication-bounded ones where the communication stage takes longer time.

If an accelerator is communication-bounded, then a better CPU-FPGA communication stack will greatly improve its overall performance. In this case, the high-bandwidth F1 instance and Xeon+FPGA v2 platform are preferred. On the other

hand, if an accelerator is computation-bounded, then switching to another platform with a better communication stack does not make considerable difference. In this case, the traditional PCIe-based private-memory platform may be preferred because of its good compatibility. One may even prefer not to choose a platform with state-of-the-art CPU-FPGA communication technology for cost efficiency. Application developers should be aware of the condition whether the application to accelerate is compute-intensive or communication-intensive, in order to select the appropriate platform.

Quantitatively, we use the computation-to-communication (C2C) ratio [ZLS15] (which is also named "operational intensity" in [WWP09]) to justify whether a computation kernel is computation/communication bounded. Specifically, the C2C ratio is defined as the division of the computation throughput and the data transfer throughput:

$$C2C\ ratio = \frac{Throughput_{compute}}{Throughput_{transfer}}$$

The computation throughput is referred to as the speed of processing a certain size of input for a given FPGA accelerator; the data transfer throughput is referred to as the speed of transferring this certain size of input into or out of the FPGA fabric. When the C2C ratio of a kernel is above 1, this kernel is then computation bounded; otherwise, it is communication bounded. In general, the data transfer throughput is linearly proportional to the input size. Therefore, a computation kernel with super-linear time complexity, such as matrix multiplication, is computation bounded. Meanwhile, computation kernels like matrix-vector multiplication that is of linear time complexity are often bounded by the CPU-FPGA communication. For

computation bounded kernels, the CPU-FPGA communication bandwidth is not the performance bottleneck, so the accelerator designers do not need to chase for high-end communication interfaces. On the other hand, the CPU-FPGA communication is critical to communication bounded kernels with C2C ratio less than 1, and the efficiency of the communication interface is then a key factor in platform selection.

Insight 5*: CPU-FPGA memory coherence is promising, but impractical to be used in accelerator design on existing platforms.*

The newly-announced CPU-FPGA platforms, including CAPI, CCIx and the Xeon+FPGA family, attempt to provide memory coherence support between the CPU and the FPGA either through PCIe or QPI. Their implementation methodology is similar: constructing a coherent cache on the FPGA fabric to realize the classic snoopy protocol with the last-level cache of the host CPU. However, although the FPGA fabric supplies a few megabytes of on-chip BRAM blocks, only 64KB (the Xeon+FPGA family) or 128KB (CAPI) of them are organized into the coherent cache. That is, these platforms maintain memory coherence for less than 5% of the on-chip memory space, leaving the majority as scratchpads of which the coherence needs to be maintained by application developers. Although users may choose to ignore the 95% scratchpads and store data on chip only through the coherent cache to obtain transparent coherence maintenance, this approach is apparently inefficient. For one thing, the coherent cache has a much longer access latency than that of the scratchpads. Also, the coherent cache provides much fewer parallel access capability compared to the scratchpads that can potentially feed thousands data per cycle. As a consequence, application developers may still have to stick to the conventional FPGA accelerator design principle to explicitly manage the coherence of the scratchpad data.

While the current implementation of the CPU-FPGA memory coherence seems to be impractical due to the aforementioned prohibitive overhead, the methodology does conceive great potential to ease the FPGA programming. The coherent cache is particularly beneficial for computation kernels with unforeseeable memory access patterns such as hashing. As will be discussed in Insight 7, implementing the coherent cache on the FPGA fabric considerably restricts its capacity, latency and bandwidth. Should it be implemented as a dedicated ASIC in future platforms, application developers could harness the power of the cache coherence. ***Insight 6****: The coherent cache design could be greatly improved.*

The coherent cache of the recently-announced CPU-FPGA platforms aims to provide the classic functionalities of CPU caches: data caching and memory coherence that are transparent to programmers. However, the long latency and small capacity make this component impractical to be used in FPGA accelerator design.

One important reason for such a long-latency, small-capacity design is that the coherent cache is implemented on the FPGA fabric. Therefore, compared to the CPU cache counterpart, the FPGA-based coherent cache has a much lower frequency and thus a much worse performance. One possible approach to address this issue is to move the coherent cache module out of the FPGA fabric as a hard ASIC circuit instead. This could potentially catch up with the scratchpad latency and realize a larger capacity, so as to be more practical to adopt in accelerator designs.

Another important reason is that the cache structure generally has very limited number of data access ports. On the contrary, the thousands of BRAM blocks on the FPGA fabric can potentially supply thousands of data in parallel. It is very common for an FPGA accelerator to have over a hundred processing elements each

of which has a dedicated BRAM buffer to achieve parallel data supply. Since massive parallelism is a widely adopted way for FPGA acceleration, the future cache design may also need to take this into consideration, e.g., a distributed, many-port cache design might be preferred to a centralized, single-port one.

Insight 7*: There still exists a large room for improvement to bridge the gap between the practically achieved bandwidth and the physical limit of the communication channel.*

For example, neither Alpha Data nor CAPI fulfills the 8GB/s PCIe bandwidth, even without considering the overhead of pinned memory allocation and pageable-pinned memory copy, so does the Amazon F1 instance. Meanwhile, it proves to be a good alternative to alleviate the communication overhead by allowing direct pageable data transfer through PCIe, which is realized in the CAPI platform. Another alternative is to offer end users the capability to directly manipulate pinned memory space. For example, both the Xeon+FPGA family and unified virtual addressing (CUDA for GPU) provide efficient and flexible API support to allow software developers to operate on allocated pinned memory arrays or objects just like those allocated by `malloc/new` [Nvi09]. Nevertheless, these solutions result in "fragmented" payloads, i.e., the payload data may be stored in discrete memory pages, causing reduced communication bandwidth.

Another alternative optimization is to form the CPU-FPGA communication stack into a coarse-grained pipeline, like the CUDA Streams technique in GPUs. This may slightly increase the communication latency for an individual payload, but could significantly boost the throughput of CPU-FPGA communication for concurrent transactions.

Both solutions should work, and actually have been proved by two of our follow-up studies. Guided by this insight, [RHL18] proposes a completely new environment that achieves a much higher effective bandwidth in the Amazon F1 instance; in Section 5.4 we propose a deep pipeline stack to overlap the communication and computation steps.

5.2.4 The JVM-FPGA Communication Routine

Based on the above analysis, we now present the entire JVM-FPGA communication routine, and the key factors that affect the communication efficiency. We use the conventional PCIe-based platform to demonstrate the routine, which can be easily adapted to other platforms. Fig. 5.5 illustrates the entire JVM-FPGA data movement process of the conventional PCIe-based CPU-FPGA platform. In the beginning, the accelerator input data, in the form of Java objects, are packed together to be transferred out of JVM (①). The accelerator host program that directly manipulates the FPGA accelerator then receives the data from JVM (②), and initiates a PCIe-based direct memory access (DMA) to send the data to the FPGA off-chip memory (④). This DMA transfer is coupled with a host-side memory copy (③) from the pageable space to the pinned space, as analyzed in Section 5.2.2.1. The data sent into the off-chip memory has to be loaded to the FPGA on-chip registers and block RAM (BRAM) (⑤), and finally be seen by the accelerator compute logic (⑥). Moreover, the generated output will go through all the above steps in the reverse direction to reach JVM (⑦-⑪), contributing the other half of the communication routine.

We now summarize the three factors that affect the JVM-FPGA communication

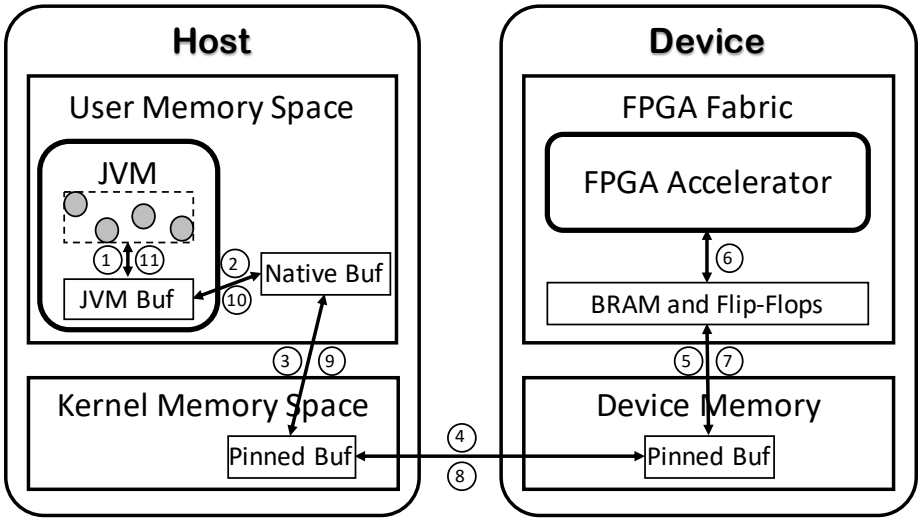


Figure 5.5: JVM-FPGA Data Communication Routine

efficiency as follows.

- **The payload size of each transaction.** As illustrated in Fig. 5.3, the maximum achievable bandwidth is much different from that with small sizes of payloads for every CPU-FPGA platform. For example, a designer can enjoy the 1.6 GB/s bandwidth if ensuring that every transaction conceives a few MBs of data, but has to take the pain of a 100× smaller bandwidth if sending data KBs by KBs.

- **The complicated communication routine.** Fig. 5.5 reveals to us with a complicated, multi-stage JVM-FPGA communication routine. The fact that these steps are performed sequentially further worsens the overall communication throughput. Our experiments show that the throughput of the overall JVM-FPGA communication routine is only a few tens to hundreds of MB/s even if we keep the payload size large.

- **Sharing of the FPGA resource among CPU threads.** State-of-the-art CPU-FPGA platforms generally have only one FPGA fabric, which is going to be shared by all CPU threads [7]. An efficient resource sharing strategy is a must to the JVM-FPGA integration, since JVM-based frameworks like Apache Hadoop and Spark are all multi-threaded.

5.2.5 Conclusion

In this section we present our analysis study that aims to evaluate and analyze the microarchitectural characteristics of state-of-the-art CPU-FPGA platforms in depth. The study covers all the latest-announced shared-memory platforms as well as the traditional private-memory Alpha Data broad and the Amazon EC2 F1 instance, with detailed data published (most of which not available from public datasheets). We found that the advertised communication parameters are often too ideal to be delivered to end users in practice, and suggest application developers avoiding over-estimation of the platform performance by considering the effective bandwidth and the communication payload. Moreover, we demonstrate that the communication-bounded accelerators can be significantly affected by different platform implementations, and propose the device memory reuse ratio as a metric to quantify the boundary of platform selection between a private-memory platform and a shared memory platform. Also, we demonstrate that the CPU-FPGA communication may not matter for computation-bounded applications where the data movement can be overlapped by the accelerator computation, and propose to use the computation to communication ratio CTC to measure it. In addition, we point out that the trans-

[7] Amazon EC2 supplies an eight-board F1 instance, but it is very expensive and the number of FPGAs is still much smaller than that of CPU threads.

parent data caching and memory coherence functionalities may be impractical in the current platforms, because of the low-capacity and high-latency cache design.

We believe these results and insights can aid platform users in choosing the best platform for a given application to accelerate, and facilitate the maturity of CPU-FPGA platforms. Furthermore, based on the analysis results, we present the entire JVM-FPGA communication routine and summarize the key factors that affect the integration efficiency. Since FPGA accelerators, compared to CPUs, work at a much lower frequency and utilize deep pipelining and extensive parallelism to achieve high performance, they in turn demands high-throughput data transfer to achieve large speedup. This explains why we observe a 1000× system-wide slowdown when integrating our Smith-Waterman accelerator into CS-BWAMEM straightforwardly. In the following section, we will switch back to the application showcase, propose our techniques to these three factors, respectively, and demonstrate how they reverse the significant slowdown back to 3.5× system-wide speedup.

5.3 Batch Processing and FaaS: the Story Continues

5.3.1 JVM-FPGA Communication Reduction

To make the integration work efficiently, we have to find a way to alleviate the tremendous JVM-FPGA communication overhead. Section 5.2 has revealed to us that the overhead can be amortized by a large number of FPGA accelerator invocations if they are grouped together and offloaded to the FPGA board at a time, i.e., processed in batches. Therefore, we propose batch processing to reduce the communication overhead. Next we discuss opportunities and challenges of batch processing

for the CS-BWAMEM/CPU and Smith-Waterman accelerator integration.

1. Each Smith-Waterman task has only 1-2KB input data and 20B output data. These small payloads result in an extremely low communication bandwidth utilization rate and leave opportunities for batch processing. Batching a large number of Smith-Waterman tasks and transferring their input and output data together can significantly increase the communication payload size and thus improve the bandwidth utilization of both DRAM (from a JVM to a native machine) and PCIe (from a native machine to an FPGA).

2. A Spark MapReduce program inherently offers a massive degree of parallelism. All map function calls in a map stage are completely independent of each other, which leaves opportunities for batch processing. To be specific, we merge a certain number of CS-BWAMEM/FPGA's map tasks derived from the straightforward integration into a new map function, and conduct a series of code transformations to batch the Smith-Waterman kernel invocations from different map tasks together.

3. There is a delicate issue in CS-BWAMEM that imposes challenges in the code transformation for batch processing, however. We use an example in Fig. 5.6 to illustrate this issue in the CS-BWAMEM algorithm, and how we design our batch processing accordingly. First, a read generates N leftward/rightward extending tasks, indicating that a map function of CS-BWAMEM (before the code transformation for batch processing) needs to process N Smith-Waterman tasks (a row in Figure 5.6), where N is highly varied for different reads. Moreover, all these Smith-Waterman tasks generated in the same read are completely chain-dependent (a row in Figure 5.6)—in any two neighbors, the successor depends

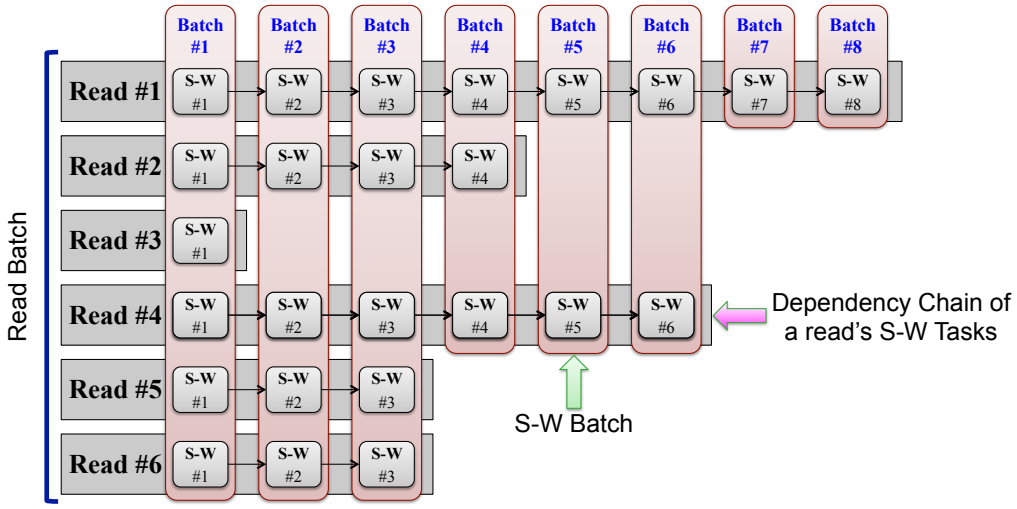


Figure 5.6: Batch processing of multi-reads in CS-BWAMEM

on the predecessor—and thus cannot be batched together. Therefore, a batched map function has to consider multiple reads (rows) as a group and produce multiple Smith-Waterman batches (columns) for this group, as illustrated in Figure 5.6: the first Smith-Waterman batch contains the first Smith-Waterman tasks of all the reads (first column); the second Smith-Waterman batch contains the second Smith-Waterman tasks of all the reads (second column); and so forth. To make it clear, we define two batch sizes: one is the read batch size, i.e., number of reads (rows) in a batch; the other is the Smith-Waterman batch size, i.e., number of Smith-Waterman tasks in a column.

To achieve an optimal performance of batch processing, we have to carefully select the right read batch size, i.e., number of reads to batch at a time. On one hand, if the read batch size is too small, the batch cannot effectively amortize the JVM-FPGA communication overhead. On the other hand, a larger read batch size comes at the cost of more memory consumption, since the host machine has to store

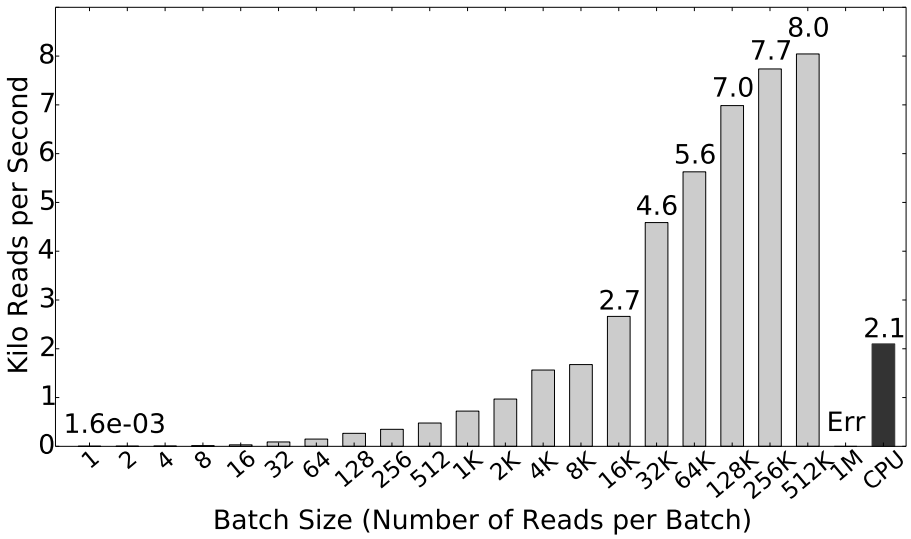


Figure 5.7: Performance under different read batch sizes

the intermediate data generated by all the batched reads. This situation will become even worse in the multi-thread scenario, since each thread obtains much less memory.

Fig. 5.7 compares the performance of processing a set of reads in CS-BWAMEM/FPGA and with different read batch sizes, as well as the performance of CS-BWAMEM/CPU. We can see that the FPGA integration starts to outperform the CPU-only version when the read batch size reaches 16k. The performance continues to increase until the program runs out of memory, where the batch size exceeds 512k.

Ideally, if all the reads in a read batch have the same Smith-Waterman task chain length and generate the same number of Smith-Waterman tasks, the batched map function will produce a number of unified-size Smith-Waterman batches, which could maximize the effectiveness of batch processing. However, the number of Smith-Waterman tasks generated by each read varies drastically, ranging from 1 to several hundred. As a result, the Smith-Waterman batch within these batched reads (each

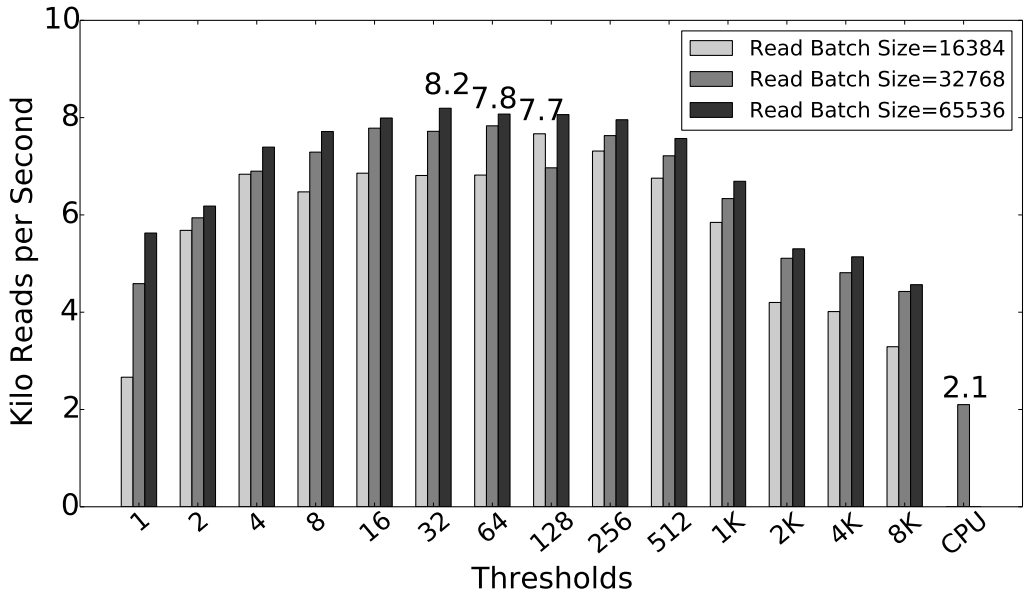


Figure 5.8: Performance under different thresholds of Smith-Waterman batch size to decide FPGA offloading

column in Figure 5.6) will become smaller when approaching the tail of these Smith-Waterman task chains, as illustrated in Figure 5.6.

To address this issue, we further optimize the batched map function by adding a threshold to the Smith-Waterman batch size: only Smith-Waterman batches (within reads) with batch size no less than the threshold should be offloaded to the FPGA accelerator. Then, we simply process all the smaller Smith-Waterman batches on CPU. Fig. 5.8 shows the performance of CS-BWAMEM/FPGA with different Smith-Waterman batch size thresholds. We only show results for read batch sizes of 16k, 32k and 64k because larger read batch sizes will lead to an "out of memory" error in the single-node multi-thread case. In addition, although a precise threshold depends on the read batch size, a threshold of 64 generally achieves close-to-optimal performance, which is around 8 KRPS and over 4x faster than CS-BWAMEM/CPU in the single-

thread case.

5.3.2 Sharing FPGA Among Multiple Threads

Due to the high performance of FPGA accelerators, offloading a single-thread CPU workload onto the FPGA usually makes the FPGA underutilized, which leaves opportunities for FPGA accelerators to be shared by multiple threads in a single node. The major challenge is how to efficiently manage the FPGA accelerator resources among multiple CPU threads. To tackle this challenge, we propose an FPGA-as-a-Service (FaaS) framework and implement the FPGA management in a node-level accelerator manager.

The FaaS framework abstracts the FPGA accelerator and its management software on the CPU (called Accelerator Manager (AM)) as a *server*, and treats each CPU thread as a *client*. Client threads communicate with AM via a hybrid of JNI and network sockets. Different client threads send requests independently to the AM to accelerate Smith-Waterman batches, and the AM processes the requests in a first-come-first-serve way. Fig. 5.9 describes the functionality and the detailed implementation of the FaaS framework.

In the single-thread integration, CS-BWAMEM/FPGA's map functions have been modified into batched map functions. The Smith-Waterman tasks from these map functions have also been reorganized into a series of Smith-Waterman batches, which are sent through JNI to the native library that manipulates the FPGA accelerator. The FaaS framework extends the native library into AM, and extends the communication mechanism between the batched map function and AM as follows

1. When a batched map function in a CPU thread needs to use the FPGA ac-

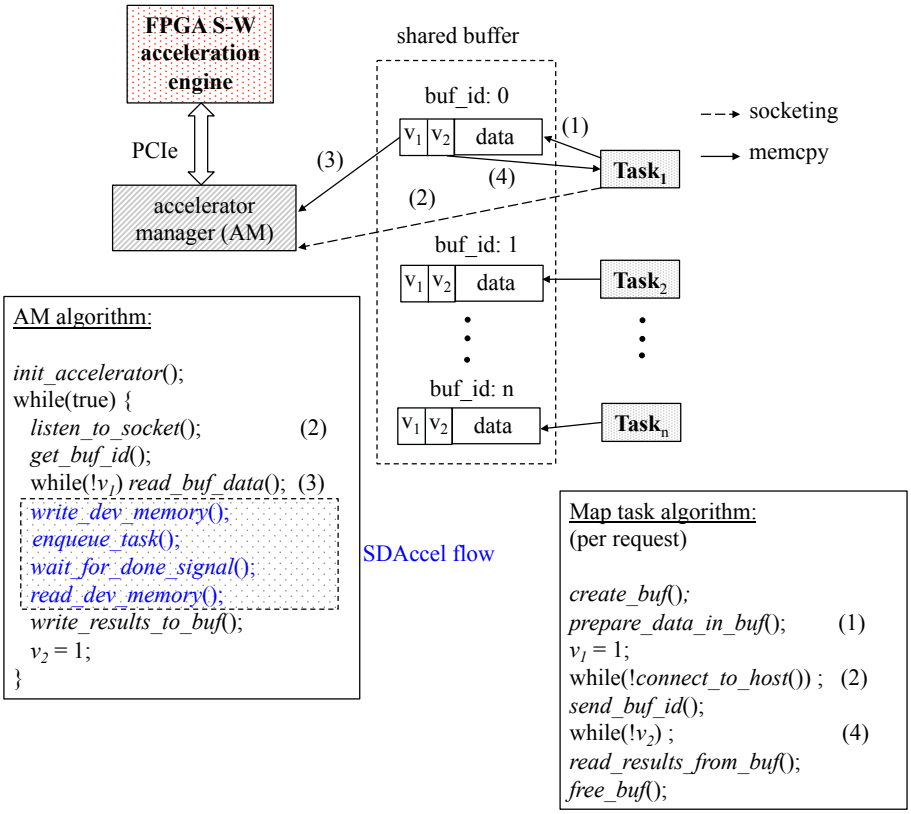


Figure 5.9: FPGA as a Service (FaaS) framework [CCF16a]

celerator, it will first allocate a shared memory buffer and then send the input data of the Smith-Waterman batch from JVM to this buffer through JNI.

2. The batched map function then sends a request to AM through a socket in order to use the accelerator. The request in this step contains only the address of the shared buffer created in Step 1, thus generating negligible overhead.

3. If the accelerator is available, it will be locked and start to process the Smith-Waterman batch; otherwise, the batched map function waits in a spin loop until it successfully gets permission to use the accelerator.

4. After the accelerator completes the current Smith-Waterman batch, it will write the output data back to the shared memory buffer created in Step 1, and become available again to accept another request.

When the number of CPU threads that share the FPGA accelerator increases, thread contention and corresponding memory pressure will become more serious. If the processing speed of the FPGA Smith-Waterman accelerator cannot match the producing speed of Smith-Waterman task requests by CPU threads, it will lead to a performance decrease. To understand the impact of thread contention, we compare the performance of CS-BWAMEM/FPGA and CS-BWAMEM/CPU with various numbers of CPU threads (there are 24 hyper-threads in a 12-core server). To address the increasing memory pressure issue caused by multi-threads, we adapt our read batch size to 32k to avoid running out of memory. As shown in Fig. 5.10, the performance of CS-BWAMEM/FPGA starts to decrease when 20 threads share the FPGA board. Meanwhile, the performance of CS-BWAMEM/CPU sightly decreases at this point as well, which indicates that hyperthreading does not always help performance improvement for CS-BWAMEM. Therefore, we will use 16 threads per CPU throughout this paper since it achieves the best performance for both CS-BWAMEM/CPU and CS-BWAMEM/FPGA. Under this 16-thread configuration, the FPGA integration achieves almost 3x speedup.

5.3.3 Scaling FPGAs into Cluster Scale

Finally, we address challenges when scaling FPGA integration into cluster scale. Two well-known major performance challenges in conventional datacenters are load balancing and communication optimization. In this paper we mainly focus on com-

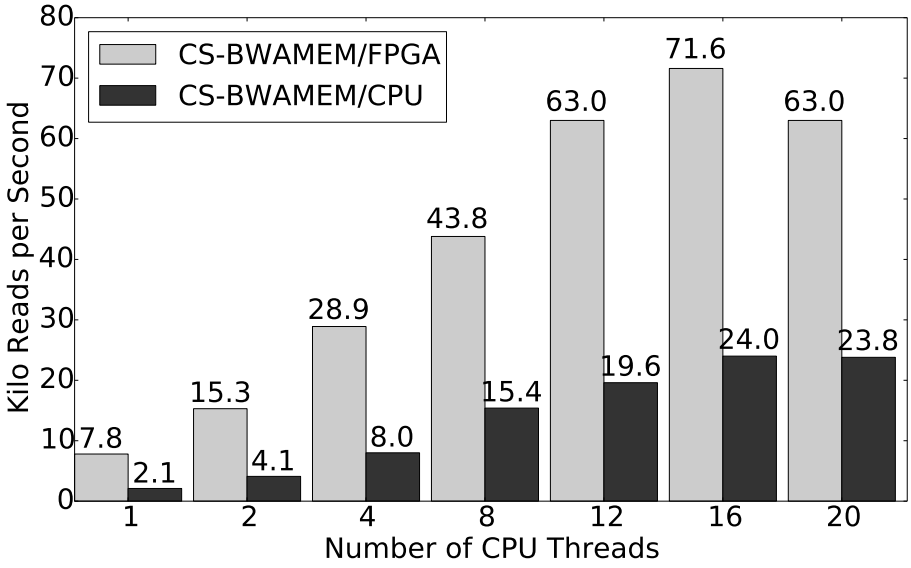


Figure 5.10: Impact of thread contention

munication optimization since we did not observe serious load balancing issues in our case study. Generally, the inter-node communication in a Spark program occurs 1) between the driver node and map/reduce stages, i.e., broadcast; and 2) between map stages and reduce stages, i.e., shuffle. Since FPGA accelerator integration usually resides entirely in map functions, like the Smith-Waterman accelerator integration in CS-BWAMEM, we only need to optimize the broadcast communication.

In CS-BWAMEM, a read-only human reference genome is needed as the input data of all the Smith-Waterman tasks. First, this reference data is very large, which costs about 10GB memory space to store in a JVM, and needs to be shared among all Spark's map functions. Second, it is read-only, so all the CPU threads in the same node need only one shared copy. The original CS-BWAMEM/CPU leverages Spark's *broadcast variables* to implement this data-sharing functionality. Before launching the map stage, Spark loads the reference data from the driver node and

sends a copy to each worker node (only once) through the network. Though this mechanism avoids creating a copy for each map function, it still costs a considerable amount of time to broadcast the 10GB data through the network, especially for a large-scale cluster. While it takes only 40 minutes to sequence a whole-genome sample through CS-BWAMEM/CPU in a 32-worker cluster, we observed a 5.5-minute overhead generated by broadcasting, i.e., a 14% performance overhead. This overhead would occupy an even larger percentage in CS-BWAMEM/FPGA where the computation is further accelerated.

We propose a broadcast-avoidance approach to address this issue. This approach is based on the observation that the human reference genome is very stable: it is updated once or twice per year. Therefore, we can store a copy of the reference genome in every worker node and reuse it for a great many DNA sequencing processes until there is an update for the reference genome released. Below, we describe how our broadcast-avoidance approach works.

1. We copy the disk files storing the reference data across all the worker nodes in the cluster, and locate them in the same absolute path.

2. Then we extend Spark's Broadcast class to implement the following mechanism. When a broadcast object needs to be processed, we first check to determine if the reference file can be found based on its "absolute_path" member variable. If so, we directly load the data from the path and register the variable in Spark's block manager; otherwise, we follow the original broadcast mechanism of Spark.

3. Finally, the reference genome is copied to the FPGA device memory for the accelerators through PCIe.

Clearly, our approach makes broadcast a constant-time of local file loading that takes only tens of seconds. More importantly, it is transparent to Spark's management; i.e., Spark can use the variable in the exact same way as its original broadcast variable.

After overcoming all the challenges in the single-thread, single-node multi-thread and multi-node levels, now we have an efficient integration of CS-BWAMEM/FPGA. As shown in Figure 5.11, the performance of Spark-FPGA integration scales well with one to six worker nodes, where each node runs 16 threads. Through the efficient integration of FPGA accelerators, CS-BWAMEM/FPGA improves the overall system performance of a 6-worker cluster by 2.6x, compared to CS-BWAMEM/CPU. Under the same configuration, CS-BWAMEM/FPGA consumes only 8% additional power per worker node. That is, CS-BWAMEM/FPGA achieves 2.4x energy efficiency improvement and 2.6x performance speedup. This result goes along with Microsoft's findings for the ranking stage of the Bing search engine where the performance is improved by 2x while consuming 10% more power per server. It is quite promising that FPGAs can greatly improve performance and energy efficiency in datacenters.

5.3.4 Analysis of Communication Overhead

To better demonstrate the effectiveness of FPGA acceleration, we present the detailed execution time breakdown (normalized to the CS-BWAMEM/CPU baseline) of our 6-worker Spark-FPGA system in Figure 5.12. The upper bar illustrates that the Smith-Waterman accelerator targets at 86% of the overall execution time, where the rest 14% of time mainly involves Spark's task scheduling and the Smith-Waterman tasks that are processed on CPU due to the Smith-Waterman batch size thresh-

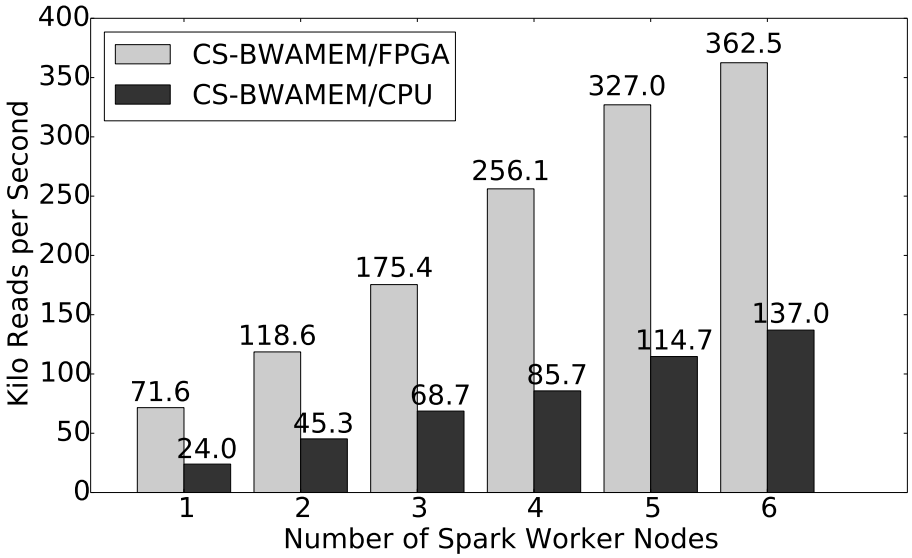


Figure 5.11: Performance of CS-BWAMEM running in a cluster w/ & w/o FPGA integration

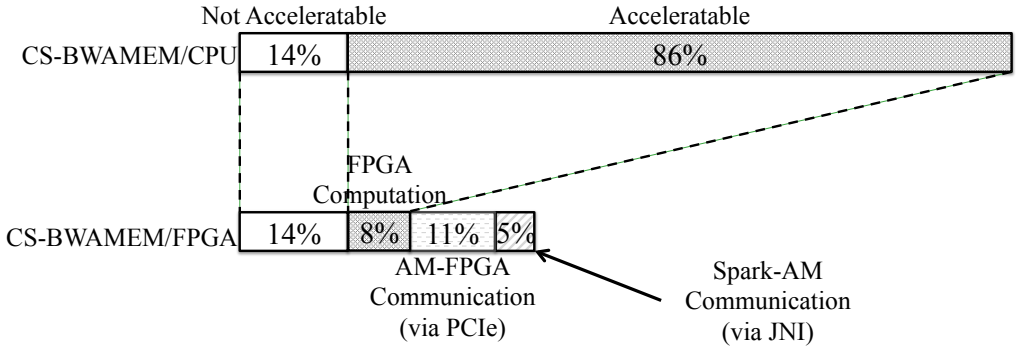


Figure 5.12: CS-BWAMEM execution time breakdown

old mechanism presented in Section 5.3.1. As shown in the lower bar, the Smith-Waterman accelerator reduces the acceleratable part (86%) to 8% while paying a 16% communication overhead. The communication between the Spark program and AM through JNI introduces 5% overhead, and the communication between AM and the FPGA accelerator through PCIe introduces another 11% overhead. The existence of

the above communication overhead reduces the overall system speedup to 2.6×. We can see that there is still room to improve the overall performance if we are able to further reduce the CPU-FPGA communication overhead.

5.3.5 Conclusion

In this section we continue to the JVM-FPGA integration process. With the findings in our analysis study on CPU-FPGA platform microarchitectures, we propose the batch processing and FaaS techniques to address the challenges at single-thread, single-node multi-thread, and multi-node levels. Using the next-generation DNA sequencing application CS-BWAMEM and its Smith-Waterman accelerator integration as a case study, we demonstrated how we turned a 1000× slowdown of the straightforward integration into an efficient integration with 2.6× overall system performance improvement step-by-step, at the cost of consuming only 8% more power per worker node. We believe that our methodology and insights can be applied to more general cases where a fine-grained FPGA accelerator with short execution time would be invoked by a Spark program a large number of times.

Meanwhile, our evaluation results show that the JVM-FPGA communication still leaves room for further improvement. In the following section, we propose our fully-pipeline JVM-FPGA data communication stack to further improve the integration efficiency.

5.4 Fully-Pipelined Communication Stack: The Story Ends.

5.4.1 Overview

As illustrated in Fig. 5.5, two impediments still exist to prevent the JVM-FPGA integration efficiency from being further improved: 1) the overall routine is fairly complex and involves many steps of data movement, and 2) these steps are performed sequentially. To resolve these impediments, in this section we proposes a high-bandwidth JVM-FPGA communication stack. Specifically, we propose a fully pipelined JVM-FPGA communication stack that allows different jobs to be transferred and processed simultaneously, i.e., overlapping different data movement steps and the computation step. As a result, the JVM-FPGA communication throughput is greatly improved to several GB/s. Furthermore, to free users from implementing the pipeline stack that involves 1) concurrent programming in Java, C and hardware description languages, 2) FPGA runtime management, and even 3) circuit design, we propose a programming framework to automatically generate most of the pipeline code, leaving only a simple Java interface to users.

One key feature of our proposed pipeline stack is that different pipeline stages can be configured with different data transfer granularities, i.e., different payload sizes, to achieve the optimal throughput because the payload size of a data transfer stage generally determines its data transfer throughput. While it is nontrivial for programmers to manually identify the best configuration of payload sizes, we formulate the problem of pipeline throughput optimization into an integer linear programming (ILP) problem and apply its solution to pipeline code generation to achieve the optimal throughput.

While implemented for generic Java programs, the proposed pipeline stack could be particularly beneficial for cloud computing frameworks, e.g., Apache Hadoop and Spark that feature a massive degree of data-level parallelism. We discuss as future work the potential integration of the pipeline stack into these frameworks. In summary, this study makes the following contributions:

- A JVM-FPGA communication pipeline that overlaps multiple communication and computation steps.
- A programming framework to automatically generate most of the pipeline code, freeing users from the bothersome concurrent and hardware intricacies.
- An ILP formulation of the pipeline optimization problem and automation of the optimization process.

Our experiments show that our approach achieves 4.9× speedup for a variety of computation kernels. By applying this technique into the application showcase, we further improve the integration efficiency and achieve a 3.5× speedup.

5.4.2 Pipelined Communication Stack

In this section we present our fully pipelined JVM-FPGA communication stack[8]. Section 5.4.2.1 describes its overall architecture and major components; Section 5.4.3 introduces our user programming model.

[8]This study is presented in [CWY18a]. I would like to convey my appreciation to all coauthors for their contributions to this study.

5.4.2.1 System Overview

In a nutshell, the proposed approach aims to form different JVM-FPGA data movement steps and the computation step into a multistage pipeline, so the overall system performance could be determined only by the stage with the longest latency, instead of the latency of the entire JVM-FPGA routine. Fig. 5.13 illustrates the overall architecture of the proposed 7-stage fully pipelined JVM-FPGA communication stack. The pipeline accepts a series of Java objects that contain the input data of the FPGA accelerator, transfers the data through three pipeline stages to the FPGA fabric, performs the computation, and finally transfers the output data back to JVM through another three pipeline stages. Each stage corresponds to one or two data movement steps illustrated in Fig. 5.5. Every two adjacent stages are glued by a concurrent queue structure which may be implemented as software lock-free queues or hardware FIFO channels. Since the last three stages are symmetric to the first three stages, we only describe the detailed functionalities of the first four stages in the remainder of this section.

Pack. The pack stage performs data reorganization. It corresponds to ① in Fig. 5.5. Specifically, it retrieves the necessary input data from Java objects and puts them into a Java byte array—so it happens completely inside JVM. The byte array is then pushed into the *send queue*, a fixed-size, lock-free Java queue structure, and finally moved to the FPGA accelerator for computation. One objective of the pack stage is to achieve batch processing, i.e., batching the input of many jobs together to form a large payload to improve the data transfer throughput.

Send. The send stage accepts byte arrays from the head of the send queue, and sends them to the FPGA accelerator management program via socket. Since the

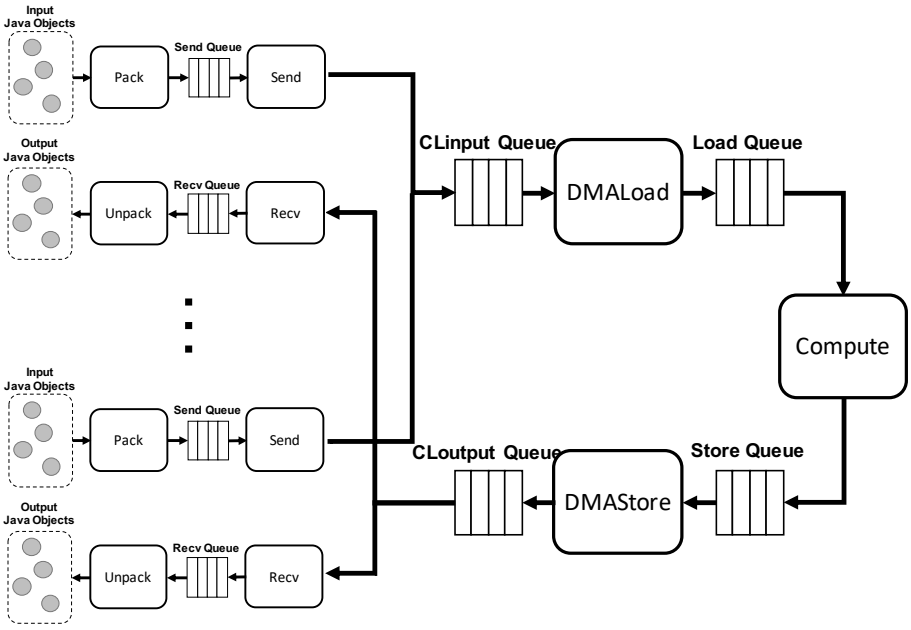


Figure 5.13: JVM-FPGA Pipeline Architecture

host Java program, e.g., a Hadoop or Spark program, may have multiple threads using the FPGA accelerator simultaneously, we use our FPGA-as-a-service (FaaS) framework [CCF16a, HWY16] to realize such resource sharing. The accelerator manager in FaaS collects the data from different threads and pushes them into the *gather queue* that is a fixed-size, lock-free C++ queue structure storing OpenCL memory objects. These OpenCL memory objects are managed by the Xilinx SDAccel runtime environment [sda], and stored in the pinned memory space to be transferred to the FPGA memory via PCIe. The entire stage corresponds to ②③ in Fig. 5.5.

DMALoad. The DMALoad stage accepts OpenCL memory objects from the gather queue and performs two data transfers. First, an OpenCL object is sent to the FPGA off-chip memory via the PCIe interface. Next, it is loaded streamingly from the off-chip memory to the *load queue* that resides in the FPGA on-chip block

RAM (BRAM). The entire stage corresponds to ④⑤ in Fig. 5.5. The load queue is a hardware FIFO channel that connects the off-chip memory to the on-chip BRAM.

Compute. The compute stage performs the actual computation of the FPGA accelerator. It loads input data from the off-chip memory via the load queue, and stores output data back to the off-chip memory via the *store queue* that is symmetric to the load queue. The output data are then transferred through the *DMAStore*, *Recv* and *Unpack* stages back to JVM, which completes the JVM-FPGA routine.

In summary, the proposed JVM-FPGA communication stack pipelines the computation and the data transfers crossing a variety of layers, including JVM, host native memory space, FPGA off-chip memory space and on-chip BRAM. While significantly improving the JVM-FPGA communication efficiency, this heterogeneous pipeline is not easy to be manually implemented. The following section presents our programming model for the system to significantly simplify user efforts.

5.4.3 Programming Model

Our programming model only requires programmers to implement an application-specific interface for the *Pack* and *Unpack* stages. For example, the interface of the *Pack* stage outputs an iterator with a series of byte arrays, as shown in Code 5.1. In this example, we assume an Advanced Encryption Standard (AES) accelerator with two arguments: `key` and `value`. The two arguments correspond to a user-defined class `StringWithKey` (line 1-4), where `value` is object-specific and `key` is shared by all `StringWithKey` objects. As can be seen in Code 5.1, the programmer only needs to implement a `PackIterator` with two methods. In particular, the `next` method (lines 13-29) returns one byte array at a time, where the first byte specifies which

accelerator argument the byte array corresponds to. The *Pack* stage will invoke `UserPacker` iteratively and pack byte arrays with a certain size and push them to the send queue. Note that to avoid sending the shared data (i.e., `key`) redundantly, our interface provides a field `isFirstObject` to indicate whether the shared data have been sent out before.

By using our programming interface to specify how to pack/unpack Java objects and byte arrays, the remainder of the pipeline stack will be automatically generated. The remaining issue in pipeline generation is to determine the data transfer granularity, i.e., payload size, which determines the throughput of its corresponding pipeline stage. Since it is nontrivial for users to find the best payload size for each stage, we hide the payload size tuning from users and present our approach for automatically identifying the best configuration of payload sizes to maximize the pipeline throughput in the next section.

List 5.1: Programming Model with AES Example

```
class StringWithKey {
  String key = ...;
  String value = ...;
}
class UserPacker implements PackIterator {
  int ptr = 0;
  StringWithKey data;

  public UserPacker(StringWithKey data) {
    this.data = data;
  }
```

```
  public boolean hasNext() { return (ptr < 2); }
  public Byte[] next() {
    if (ptr == 0 && !this.isFirstObject)
      return // Convert key to byte array
    else if (ptr == 1)
      return // Convert value to byte array
    ptr++;
    return null;
}}
```

5.4.4 Pipeline Throughput Optimization

In this section we focus on the optimization of the overall pipeline throughput, i.e., the identification of the best payload sizes for all the pipeline stages. Section 5.4.4.1 first analyzes the impact of the payload size on pipeline throughput. According to the analysis, we formulate the problem to an integer linear programming (ILP) in Section 5.4.4.2 to find the best payload sizes.

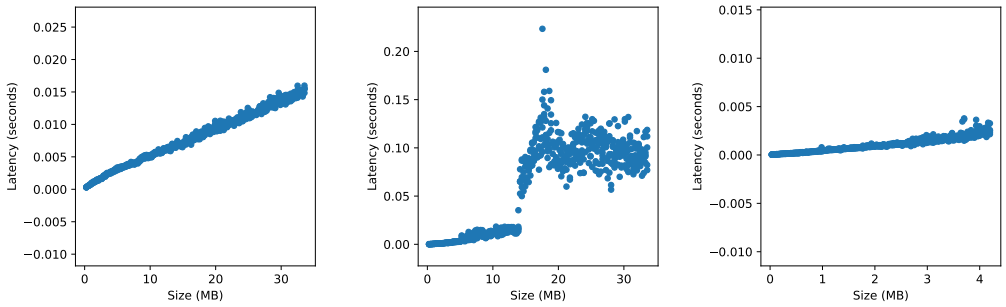
5.4.4.1 Analysis of Data Transfer Throughput

In general, the latency of transferring a certain size of payload can be decoupled into two parts: 1) a constant time setup overhead, and 2) the payload movement time that is proportionate to the size of the payload. Because of the setup overhead, the data transfer throughput grows rapidly with respect to the payload size when it is small, and gradually reaches a stable value since the impact of the setup overhead is amortized. Some of the pipeline stages, e.g., the *DMALoad* stage, follow this rule

very well, as is demonstrated in Fig. 5.14 (a). In this case, a larger payload size is always favored.

Not all the pipeline stages, however, deliver a perfect linear relation. Fig. 5.14 (b) shows the changes of latency with respect to the payload size for the *Send* stage. The payload size ranges from 0 to 32 MBs. We can see that the linear trend is not overall applicable, but still persists when the payload size is below a few megabytes, as shown in Fig. 5.14 (c). A key reason is that the last-level cache is not able to hold all the intermediate data any more with the growing payload size, resulting in the sharp performance degradation in Fig. 5.14 (b). In detail, the PCIe-based CPU-FPGA data transfer is implemented as a direct memory access (DMA) which utilizes the cycle stealing technology [Mar80] to efficiently share the host-side memory bandwidth between regular CPU memory accesses and the DMA. The efficiency of this sharing, therefore, depends on the memory bandwidth requirement of the CPU. If the CPU suffers a high last-level cache (LLC) miss rate, i.e., a large number of memory accesses, then the cycle stealing mechanism will not work well. By using the Linux `perf` tool to profile the LLC cache miss rate for different payload sizes, we observe a greatly increased LLC miss rate: 19.8% for 1MB payload, and 33.2% for 4MB payload. As a consequence, the throughput optimization problem becomes more complicated when the memory constraint is taken into consideration. Consequently, the payload size should be allocated wisely among different stages for global optimality given certain memory constraints (in our case the LLC capacity).

However, as shown in Fig. 5.14 (c), even for data transfer stages that show obvious non-linear trend with large payload sizes, they still persist linear trend with small sizes of payload. Therefore, by confining the payload sizes inside certain upper bounds, the problem can still be linearized. Such confinement does not lose opti-



(a) DMALoad (b) Send (0∼32MB) (c) Send (0∼4MB)

Figure 5.14: Latency-Size Curve for Different Stages

mality since for any point in Fig. 5.14 (b) that falls in the large-size area, we can always replace it with a point inside the small-size area which has an even larger throughput. The following section presents our ILP-based approach to solve this problem mathematically.

5.4.4.2 Payload Size Tuning

In a nutshell, we attempt to formulate the problem of tuning the payload size of each pipeline stage into an ILP problem in which the solution can be obtained via a standard ILP solver.

Problem Formulation: Given a computation kernel K, find a set of payload sizes $S = \{S_{pack},\ S_{send},\ ...,\ S_{unpack}\}$ so as to maximize the overall throughput T_K. Since the throughput of a pipeline is bounded by the stage that has the minimal throughput, the overall throughput can be modeled via Eq. 5.1:

$$T_K = Min(T_{pack},\ T_{send},\ ...,\ T_{unpack}) \tag{5.1}$$

where T_{pack}, ..., T_{unpack} denote the throughputs of the seven stages, respectively. Also, we know that the throughput of a stage T_{stage} is inversely proportional to its latency L_{stage}, which can be represented as a function related to the payload size S_{stage}:

$$T_{stage} = \frac{1}{L_{stage}} = \frac{1}{f_{stage}(S_{stage})} \tag{5.2}$$

Therefore, to solve the problem, we need to determine each function f_{stage}.

Integer Linear Programming Formulation: To form an ILP, we model f_{stage} for each stage to a linear function while preserving the practicality and optimality.

First, the data transfer stages, i.e., *Pack/Unpack, Send/Recv* and *DMALoad/DMAStore*, have linear relations between the payload size and the latency. For the *Send/Recv* stage where the latency increases dramatically after the payload size hits a certain threshold, these large sizes can be filtered out since we can always find a better (smaller) size with a similar or higher data transfer throughput. Therefore, we are able to formulate function f_{stage} for these six stages as linear functions. Note that the *Pack/Unpack* stages are application-specific, so we profile the the application with a small dataset. The other four stages, however, are platform-specific, so we only need to profile the platform once to derive f_{stage}, which is then used for all applications running on this platform.

We then model the *Compute* stage. Depending on the time complexity of the accelerator, the computation latency may not have linear dependency to the input size. To address this issue, we profile the compute time with a set of factor-of-two

input sizes, since factor-of-two data sizes generally achieve high efficiency in circuit design. Subsequently, the accelerator latency can be represented by the following linear equation:

$$L_{compute} = \sum_i p_i \times L_{S_i},\ where\ \sum_i p_i = 1,\ p_i \in \{0, 1\} \tag{5.3}$$

where L_{S_i} denotes the latency of the i-th profiled performance point; p_i is a binary variable for each point and only one of them will be 1, i.e., only one profiled performance point with the best input size will be delivered.

Finally, we specify the memory constraint. It indicates that the overall sizes of all the queue structures cannot exceed a given memory capacity, as shown in Eq. 5.4:

$$\sum S_{Q_{stage}} = \sum_i (S_{stage} \times D_{stage}) \leq S_{capacity} \tag{5.4}$$

where $S_{Q_{stage}}$ denotes the overall size of the queue structures for each stage and is determined by the size of each entry (S_{stage}) as well as the queue depth (D_{stage}, fixed in the proposed pipeline). Note that the software and hardware queues occupy different memory space and thus are evaluated separately.

The list below summarizes the complete analytical model. By constraining the overall memory footprint not exceeding the capacity of the last-level cache, all equations, in particular the latency equations for all the data communication stages, are linear, so the payload sizes can be determined with an ILP solver.

$$max : T_K = Min(T_{pack},\ T_{send},\ ...,\ T_{unpack}) \tag{5.5}$$

subject to

$$T_{stage} = \frac{1}{L_{stage}},\ where\ stage \in \mathcal{S} = \{pack, send, ..., unpack\} \tag{5.6}$$

$$L_{comm} = L_{init} + S_{comm} \times L_{comm}^{per\_byte},\ where\ comm \in \mathcal{S} - \{compute\} \tag{5.7}$$

$$L_{compute} = \sum_i p_i \times L_{S_i},\ where\ \sum_i p_i = 1,\ p_i \in \{0, 1\} \tag{5.8}$$

$$\sum S_{Q_{stage}} = \sum_i (S_{stage} \times D_{stage}) \leq S_{capacity} \tag{5.9}$$

5.4.5 Experiments

We perform the experiments based on the PCIe-based CPU-FPGA platform that connects a Xeon CPU (E5-2420) and an Xilinx FPGA board (Alpha Data ADM-PCIE-7V3 [Xil17]) via the PCIe interface (Gen3 x8). On top of it, we use the Xilinx SDAccel runtime environment v2017.2 [sda] to drive the FPGA acceleration. On the host side, we use a set of computation kernels from the MachSuite benchmark suite [RAS14], to demonstrate the effectiveness of the pipeline stack on variant types of kernels. We demonstrate the effectiveness of the proposed pipeline by writing a single-thread Java program for each kernel to continuously invoke its FPGA acceleration routine.

Fig. 5.15 compares the execution time between the proposed pipeline stack and the conventional sequential stack with 512KB, 1MB and 2MB payload sizes. We can see that the proposed pipeline stack achieves significant performance improvement on

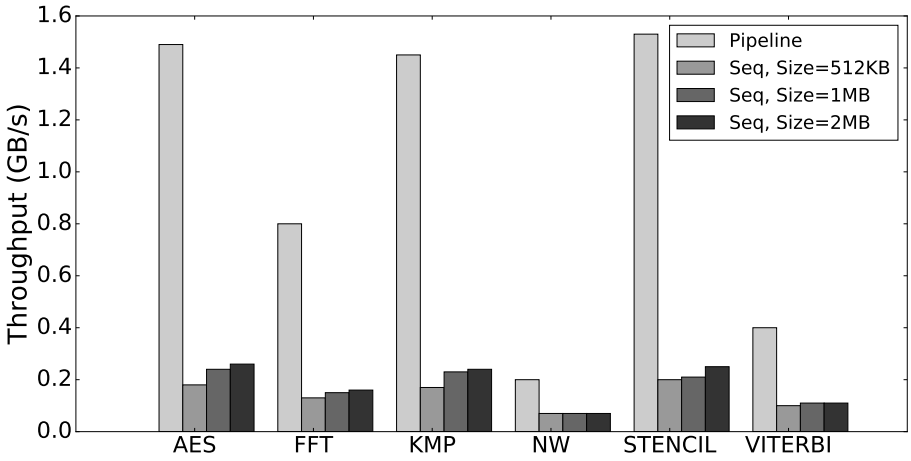


Figure 5.15: Throughput comparison between the pipeline and sequential JVM-FPGA communication stacks.

all kernels (4.9× on average), especially AES, KMP and STENCIL (5.7× to 6.1×). This is because these computation kernels are of linear time complexity, and the effective computation time after the FPGA acceleration is smaller than any of the data movement steps. The other kernels, i.e., FFT, NW and VITERBI, have super-linear time complexity, and the computation time still takes an important portion of the overall routine. Therefore, the achieved speedup (2.8× to 5.0×) is smaller, but still remarkable.

This trend is also exhibited in the throughput optimization results. Fig. 5.16 illustrates the performance difference between our ILP-based approach and the ones using constant-size payloads (512KB, 1MB or 2MB). We can see that the proposed approach is particularly beneficial for AES, KMP and STENCIL (34% to 65% improvement), but has moderate impact on FFT, NW and VITERBI (up to 7% improvement). This is because in the former three kernels the *Compute* stage is fully overlapped by the communication stages, allowing us to do more on throughput op-

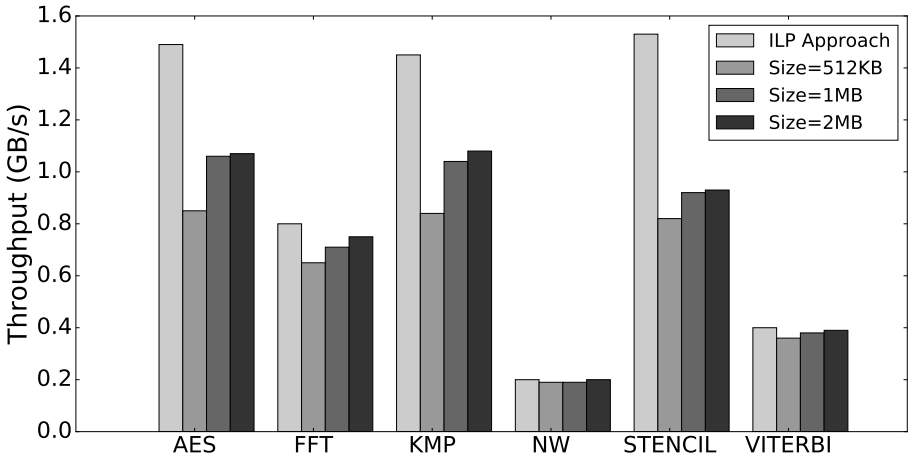


Figure 5.16: Throughput comparison between the proposed approach and the ad hoc solutions.

timization via changing the payload sizes. On the other hand, *Compute* is still the most time-consuming stage in the latter three kernels, so having payloads with a reasonable constant size is sufficient.

Finally, we apply the pipeline technique to the genome sequencing acceleration process. The results show that we further improve the system-wide performance to 3.5×, which demonstrates the effectiveness of the pipeline solution in real applications.

5.4.6 Conclusion

This study concludes our JVM-FPGA integration methodology. In summary, we propose three techniques to address the three factors that affect the integration efficiency. The batch processing technique aims to create large sizes of payloads to better utilize the communication channels. The FaaS framework aims to share the FPGA resource among multiple CPU threads. The pipeline stack aims to overlap multiple

communication stages and the compute stage to improve the overall throughput. Our application showcase demonstrates that these three techniques can be combined and applied together, and reverse a 1000× slowdown to a 3.5× system-wide speedup.

CHAPTER 6

Conclusions

This dissertation is dedicated to facilitating the current trend of adopting FPGAs in datacenters. We focus on the two major issues that prevent general datacenter application developers from accepting FPGAs: the notoriously poor programmability and the severe CPU-FPGA integration overhead, and propose our methodology to address both of them. For the first issue, we find that a best-effort code reconstruction practice can actually lead to high-quality accelerator designs from software programs for a variety of computation kernels. Inspired by this finding, we derive the best-effort practice into the CPP microarchitecture as a template of accelerator designs. The beauty of introducing such a template is to significantly reduce the design space from "anything possible" to only the scope of CPP. Moreover, this well-defined microarchitecture enables us to develop an analytical model to quantify the performance-resource trade-offs among different configurations of CPP. The CPP model in turn leads to fast design space exploration to identify the optimal CPP configuration in a reasonable time. With all these advantages, we implement the AutoAccel framework to automate the entire accelerator generation process. This delivers to general datacenter application developers a nearly push-button experience to obtain FPGA accelerators with good performance.

We identify the second issue via a case study for the genome sequencing accelera-

tion. When integrating a high-quality accelerator into a high-performance datacenter application based on the Apache Spark framework, we find that such an "alliance between giants" actually leads to a 1000× system-wide performance degradation. By conducting a quantitative analysis on the microarchitectures of five state-of-the-art CPU-FPGA platforms, we identify three key factors that affect the efficiency of CPU-FPGA integration: the payload size of each transaction, the multi-stage communication routine, and the sharing of the FPGA resource among CPU threads. We then propose three techniques, batch processing, the fully-pipelined communication stack, and the FaaS framework, to address these factors, respectively. By applying them back to our case study for genome sequencing acceleration, we reverse the drastic slowdown back to a system-wide speedup.

We believe that our methodologies have effectively addressed these two issues, and could inspire more research in both directions. For the AutoAccel framework, it serves as a proof of concept that a template-based design automation approach does work well. However, the CPP microarchitecture does not cover the entire spectrum of computation kernels, and thus more templates are expected to be proposed, with their own analytical models and code transformation practice. For the CPU-FPGA integration, the constant evolution of CPU-FPGA platforms makes it always a "hot topic". For instance, the Amazon EC2 F1 instance brings virtualization into consideration, and we expect more FPGA instances on a variety of public clouds in the future. It is still an open topic to consider both FPGA features and system virtualization for a more efficient and secure integration.

REFERENCES

[ABC10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures." In *OOPSLA*, 2010.

[amaa] "Amazon EC2.".

[amab] "Amazon EC2 F1 Instance.".

[Arv03] Arvind. "Bluespec: A language for hardware design, simulation, synthesis and verification Invited Talk." In *MEMOCODE*, 2003.

[ASH15] Nauman Ahmed, Vlad-Mihai Sima, Ernst Houtgast, Koen Bertels, and Zaid Al-Ars. "Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm." In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2015.

[ATL13] J. Arram, K.H. Tsoi, Wayne Luk, and P. Jiang. "Hardware Acceleration of Genetic Sequence Alignment." In Philip Brisk, JosGabriel de Figueiredo Coutinho, and PedroC. Diniz, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7806 of *Lecture Notes in Computer Science*, pp. 13–24. Springer Berlin Heidelberg, 2013.

[BFR12] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[Bir06] Richard S. Bird. "Improving Saddleback Search: A Lesson in Algorithm Design." In *Mathematics of Program Construction*. Springer, 2006.

[Bre10] Tony M Brewer. "Instruction set innovations for the Convey HC-1 computer." *IEEE Micro*, (2):70–79, 2010.

[BRH15] Brad Brech, Juan Rubio, and Michael Hollinger. "IBM Data Engine for NoSQL - Power Systems Edition." Technical report, IBM Systems Group, 2015.

[Bri12] Pierre Bricaud. *Reuse methodology manual: for system-on-a-chip designs*. Springer Science & Business Media, 2012.

[BVR12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. "Chisel: constructing hardware in a scala embedded language." In *DAC-49*, 2012.

[CCA11] 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 *FPGA*, 2011.

[CCF16a] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. "When apache spark meets FPGAs: a case study for next-generation DNA sequencing acceleration." In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, pp. 64–70. USENIX Association, 2016.

[CCF16b] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. "A quantitative analysis on microarchitectures of modern CPU-FPGA platforms." In *Proceedings of the 53rd Annual Design Automation Conference*, p. 109. ACM, 2016.

[cci18] "Cache Coherent Interconnect for Accelerators.", 2018.

[CCL15a] Yu-Ting Chen, J. Cong, Jie Lei, and Peng Wei. "A Novel High-Throughput Acceleration Engine for Read Alignment." In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 199–202, May 2015.

[CCL15b] Yu-Ting Chen, Jason Cong, Sen Li, Myron Peto, Paul Spellman, Peng Wei, and Peipei Zhou. "CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing." *High Throughput Sequencing Algorithms and Applications (HITSEQ)*, 2015.

[CCP16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. "A Cloud-Scale Acceleration Architecture." In *MICRO-49*, 2016.

[CFH17] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. "Supporting Address Translation for Accelerator-Centric Architectures." In *HPCA-23*, 2017.

[CFH18] Jason Cong, Zhenman Fang, Yuchen Hao, Peng Wei, Cody Hao Yu, Chen Zhang, and Peipei Zhou. "Best-Effort FPGA Programming: A Few Steps Can Go a Long Way." *arXiv preprint arXiv:1807.01340*, 2018.

[CG15] Alessandro Cilardo and Luca Gallo. "Improving Multibank Memory Access Parallelism with Lattice-Based Partitioning." *TACO*, 2015.

[CGG13] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity." In *ISLPED*, 2013.

[CHP16a] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. "Source-to-Source Optimization for HLS." In *FPGAs for Software Programmers*. Springer International Publishing, 2016.

[CHP16b] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. "Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper." In *ISLPED*, 2016.

[CHZ14] Jason Cong, Muhuan Huang, and Peng Zhang. "Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications." In *FPGA*, 2014.

[CJL11] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization." 2011.

[CLN11] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment." 2011.

[CMD15] Emilio G Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P Carloni. "An analysis of accelerator coupling in heterogeneous architectures." In *DAC*, 2015.

[Coo12] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[CSR11] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. "Customizable domain-specific computing." *IEEE Design & Test of Computers*, **28**(2):6–15, 2011.

[CTI15] Nanchini Chandramoorthy, Giuseppe Tagliavini, Kevin Irick, Antonio Pullini, Siddharth Advani, Sulaiman Al Habsi, Matthew Cotter, John Sampson, Vijaykrishnan Narayanan, and Luca Benini. "Exploring architectural heterogeneity in intelligent vision systems." In *HPCA*, 2015.

[CWY17] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. "Bandwidth optimization through on-chip memory restructuring for HLS." In *DAC*, 2017.

[CWY18a] Jason Cong, Peng Wei, and Cody Hao Yu. "From JVM to FPGA: Bridging Abstraction Hierarchy via Optimized Deep Pipelining." In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. USENIX Association, 2018.

[CWY18b] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. "Automated accelerator generation and optimization with composable, parallel and pipeline architecture." In *Proceedings of the 55th Annual Design Automation Conference*, p. 154. ACM, 2018.

[CZZ12] Jason Cong, Peng Zhang, and Yi Zou. "Optimizing Memory Hierarchy Allocation with Loop Transformations for High-level Synthesis." In *DAC*, 2012.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM*, **51**(1):107–113, 2008.

[DZZ18] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline FY Young, and Zhiru Zhang. "Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning." In *International Symposium on Field-Programmable Custom Computing Machines*, 2018.

[FMY15] Zhenman Fang, Sanyam Mehta, Pen-Chung Yew, Antonia Zhai, James Greensky, Gautham Beeraka, and Binyu Zang. "Measuring microarchitectural details of multi-and many-core memory systems through microbenchmarking." *ACM Transactions on Architecture and Code Optimization (TACO)*, **11**(4):55, 2015.

[GBS15] M. Grossman, M. Breternitz, and V. Sarkar. "HadoopCL2: Motivating the Design of a Distributed, Heterogeneous Programming System

With Machine-Learning Applications." *Parallel and Distributed Systems, IEEE Transactions on*, **PP**(99):1–1, 2015.

[GGL99] William D Gropp, William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[GKV98] Adi F Gazdar, Venkatesh Kurvari, Arvind Virmani, Lauren Gollahon, Masahiro Sakaguchi, Max Westerfield, Duli Kodagoda, Victor Stasny, H Thomas Cunningham, Ignacio I Wistuba, et al. "Characterization of paired tumor and non-tumor cell lines established from patients with breast cancer." *International journal of cancer*, **78**:766–774, 1998.

[GWC16] Xitong Gao, John Wickerson, and George A Constantinides. "Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis." In *FPGA*, 2016.

[hara] "Intel to Start Shipping Xeons With FPGAs in Early 2016.".

[harb] "Xeon+FPGA Platform for the Data Center.".

[HWB09] A. Hagiescu, W. F. Wong, D. F. Bacon, and R. Rabbah. "A computing origami: Folding streams in FPGAs." In *DAC*, 2009.

[HWY16] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. "Programming and runtime support to blaze FPGA accelerator deployment at datacenter scale." In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 456–469. ACM, 2016.

[IBM15] IBM. *Coherent Accelerator Processor Interface Users Manual Xilinx Edition*, 2015. Rev. 1.1.

[int] "Intel SDK for OpenCL." https://software.intel.com/en-us/intel-opencl.

[Int16] Intel. *BDW+FPGA Beta Release 5.0.3 Core Cache Interface (CCI-P) Interface Specification*, 9 2016. Rev. 1.0.

[KDP16] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware." In *ISCA-43*, 2016.

[Kim11] Maria Kim. "Accelerating Next Generation Genome Reassembly in FPGAs: Alignment Using Dynamic Programming Algorithms." *University of Washington, master thesis*, 2011.

[Law14] Jason Lawley. *Understanding Performance of PCI Express Systems*. Xilinx, 10 2014. Rev. 1.2.

[LBC15] J. Liu, S. Bayliss, and G. A. Constantinides. "Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS." In *FCCM*, 2015.

[LC13] Zhongduo Lin and Peter Chow. "ZCluster: A Zynq-based Hadoop cluster." In *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 450–453. IEEE, 2013.

[LD09] Heng Li and Richard Durbin. "Fast and accurate short read alignment with Burrows–Wheeler transform." *Bioinformatics*, **25**(14):1754–1760, 2009.

[LD10] Heng Li and Richard Durbin. "Fast and accurate long-read alignment with Burrows–Wheeler transform." *Bioinformatics*, **26**(5):589–595, 2010.

[LGJ14] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. "CGPA: Coarse-Grained Pipelined Accelerators." In *DAC-51*, 2014.

[LH10] Heng Li and Nils Homer. "A survey of sequence alignment algorithms for next-generation sequencing." *Briefings in bioinformatics*, 2010.

[Li13] Heng Li. "Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM." *arXiv preprint arXiv:1303.3997*, 2013.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[LL15] Peilong Li and Yan Luo. "HeteroSpark: A Heterogeneous CPU/GPU Spark Platform for Deep Learning Algorithms." *Spark Summit 2015e*, 2015.

[llv07] "LLVM Language Reference Manual." 2007. http://llvm.org/docs/LangRef.html.

[LS12] Ben Langmead and Steven L Salzberg. "Fast gapped-read alignment with Bowtie 2." *Nature methods*, **9**(4):357–359, 2012.

[LWC16] J. Liu, J. Wickerson, and G. A. Constantinides. "Loop Splitting for Efficient Pipelining in High-Level Synthesis." In *FCCM*, 2016.

[LYB14] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[Mar80] George B Marenin. "Microprocessor architecture with integrated interrupts and cycle steals prioritized channel.", January 1 1980. US Patent 4,181,934.

[mer] "Merlin Compiler." http://www.falcon-computing.com/index.php/solutions/merlin-compiler.

[MHB10] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, et al. "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data." *Genome research*, **20**(9):1297–1303, 2010.

[MK15] P. Moorthy and N. Kapre. "Zedwulf: Power-Performance Tradeoffs of a 32-Node Zynq SoC Cluster." In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 68–75, May 2015.

[MNH13] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, André Schumacher, Anthony D Joseph, and David A Patterson. "Adam: Genomics formats and processing patterns for cloud scale computing." *EECS Department, University of California, Berkeley, Tech. Rep.*, 2013.

[MPA16] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. "TABLA: A unified template-based framework for accelerating statistical machine learning." In *HPCA-22*, 2016.

[MSS14] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. "Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms." In *ISCA*, 2014.

[NMG15] K. Neshatpour, M. Malik, M.A. Ghodrat, and H. Homayoun. "Accelerating Big Data Analytics Using FPGAs." In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pp. 164–164, May 2015.

[NSS16] E. Nurvitadhi, Jaewoong Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC." In *FPL*, 2016.

[Nvi09] Nvidia. "NVIDIA's Next Generation CUDA Compute Architecture: FERMI." *Comput. Syst*, **26**:63–72, 2009.

[OKC12] Corey B Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L Ruzzo. "Hardware acceleration of short read mapping." In *FCCM, IEEE 20th Annual International Symposium on*, 2012.

[OLQ14] Jian Ouyang, Shiding Lin, Wei Qi, Yong Wang, Bo Yu, and Song Jiang. "Sda: Software-defined accelerator for largescale dnn systems." In *Hot Chips*, 2014.

[ORK15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. "Toward accelerating deep learning at scale using specialized hardware in the datacenter." In *Hot Chips*, 2015.

[PBD08] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures." In *FPL*, 2008.

[PBM99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank citation ranking: Bringing order to the web." Technical report, Stanford InfoLab, 1999.

[PCC14] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. "A reconfigurable fabric for accelerating large-scale datacenter services." In *ISCA*, 2014.

[PHA17] Johan Peltenburg, Ahmad Hesam, and Zaid Al-Ars. "Pushing Big Data into Accelerators: Can the JVM Saturate Our Hardware?" In *International Conference on High Performance Computing*, pp. 220–236. Springer, 2017.

[PKB16] Raghu Prabhakar, David Koeplinger, Kevin J Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. "Generating configurable hardware from parallel patterns." In *ASPLOS-XXI*, 2016.

[PSK15] Nam Khanh Pham, Amit Kumar Singh, Akash Kumar, and Mi Mi Aung Khin. "Exploiting Loop-array Dependencies to Accelerate the Design Space Exploration with High Level Synthesis." In *DATE*, 2015.

[PZS13] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. "Polyhedral-based Data Reuse Optimization for Configurable Computing." In *FPGA*, 2013.

[RAS14] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. "Machsuite: Benchmarks for accelerator design and customized architectures." In *IISWC*, 2014.

[RHL18] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. "ST-Accel: A high-level programming platform for streaming applications on FPGA." In *FCCM*, 2018.

[Rog13] Phil Rogers. "Heterogeneous system architecture overview." In *Hot Chips*, 2013.

[ros00] "Rose Compiler Infrastructure." 2000. http://rosecompiler.org/.

[Ryf] Ryft. "Real-time Big Data Search and Analytics.".

[SBC] Adrian Sampson, James Bornholt, and Luis Ceze. "Hardware-Software Co-Design: Not Just a Cliché." In *1st Summit on Advances in Programming Languages (SNAPL 2015)*.

[SBJ15] J Stuecheli, Bart Blaner, CR Johns, and MS Siegel. "CAPI: A Coherent Accelerator Processor Interface." *IBM Journal of Research and Development*, **59**(1):7–1, 2015.

[SCN15] Oren Segal, Philip Colangelo, Nasibeh Nasiri, Zhuo Qian, and Martin Margala. "SparkCL: A Unified Programming Framework for Accelerators on Heterogeneous Clusters." *CoRR*, **abs/1505.01120**, 2015.

[sda] "SDAccel Development Environment." http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[SJ08] Jay Shendure and Hanlee Ji. "Next-generation DNA sequencing." *Nature biotechnology*, **26**(10):1135–1145, 2008.

[Sol] Falcon Computing Solutions. "Enable Customized Computing in Datacenter-Scale Applications." http://www.falcon-computing.com/.

[SPA16] Hardik Sharma, Jongse Park, Emmanuel Amaro, Bradley Thwaites, Praneetha Kotha, Anmol Gupta, Joon Kyung Kim, Asit Mishra, and Hadi Esmaeilzadeh. "DNNWEAVER: From High-Level Deep Network Models to FPGA Acceleration." In *MICRO-49*, 2016.

[SW81] T.F. Smith and M.S. Waterman. "Identification of common molecular subsequences." *Journal of Molecular Biology*, **147**(1):195 – 197, 1981.

[SWY10] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. "FPMR: MapReduce framework on FPGA." In *Proceedings of the 18th annual ACM/SIGDA international symposium on FPGA*. ACM, 2010.

[SYZ16] Jincheng Su, Fan Yang, Xuan Zeng, and Dian Zhou. "Efficient Memory Partitioning for Parallel Data Access via Data Reuse." In *FPGA*, 2016.

[TL10] Kuen Hung Tsoi and Wayne Luk. "Axel: A Heterogeneous Cluster with FPGAs and GPUs." In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2010.

[TLZ15] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. "ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests." In *ICCAD*, 2015.

[TM08] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[VHS15] Anand Venkat, Mary Hall, and Michelle Strout. "Loop and Data Transformations for Sparse Matrix Code." In *PLDI*, 2015.

[WH13] Gabriel Weisz and James C. Hoe. "C-to-CoRAM: Compiling Perfect Loop Nests to the Portable CoRAM Abstraction." In *FPGA*, 2013.

[Whi12] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[WHZ16] Zeke Wang, Bingsheng He, Wei Zhang, and Shunning Jiang. "A performance analysis framework for optimizing OpenCL applications on FPGAs." In *HPCA-22*, 2016.

[WLC14] 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*, pp. 199–208. ACM, 2014.

[WLZ13] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. "Memory Partitioning for Multidimensional Arrays in High-level Synthesis." In *DAC*, 2013.

[WPS10] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. "Demystifying GPU microarchitecture through microbenchmarking." In *ISPASS*, 2010.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures." *Communications of the ACM*, **52**(4):65–76, 2009.

[Xil17] Xilinx. *ADM-PCIE-7V3 Datasheet*, 1 2017. Rev. 1.3.

[YOK15] Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Andrey Ayupov, Steven Burns, and Ozcan Ozturk. "Hardware Accelerator Design for Data Centers." In *ICCAD*, 2015.

[YTT08] Jackson HC Yeung, CC Tsang, Kuen Hung Tsoi, Bill SH Kwan, Chris CC Cheung, Anthony PC Chan, and Philip HW Leong. "Map-reduce as a programming model for custom computing machines." In *FCCM, 16th International Symposium on*. IEEE, 2008.

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing." In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012.

[Zha17] Jieru Zhao et al. "COMBA: A Comprehensive Model-Based Analysis Framework for High Level Synthesis of Real Applications." In *ICCAD*, 2017.

[ZHX15] Peng Zhang, Muhuan Huang, Bingjun Xiao, Hui Huang, and Jason Cong. "CMOST: a system-level FPGA compilation framework." In *DAC-52*, 2015.

[ZLC13] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. "Improving Polyhedral Code Generation for High-level Synthesis." In *CODES+ISSS*, 2013.

[ZLS15] 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 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pp. 161–170, New York, NY, USA, 2015. ACM.

[ZMS16] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs." In *SC*, 2016.

[ZPL16] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. "Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators." In *DAC*, 2016.

[ZPW17] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. "Design Space Exploration of FPGA-based Accelerators with Multi-level Parallelism." In *DATE*, 2017.

[ZTG07] Peiheng Zhang, Guangming Tan, and Guang R. Gao. "Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform." In *Proceedings of the 1st International Workshop on High-performance Reconfigurable Computing Technology and Applications*, HPRCTA '07, pp. 39–48. ACM, 2007.