Title
Modeling and Optimization for Customized Computing: Performance, Energy and Cost Perspective

Author
Zhou, Peipei

Publication Date
2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Modeling and Optimization for Customized Computing: Performance, Energy and Cost Perspective

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

by

Peipei Zhou

2019

ABSTRACT OF THE DISSERTATION

Modeling and Optimization for Customized Computing:
Performance, Energy and Cost Perspective

by

Peipei Zhou
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Jingsheng Jason Cong, Chair

This dissertation investigates design target, modeling, and optimization for field-programmable gate array (FPGA) customized computing at chip-level, node-level and cluster-level. FPGAs have gained popularity in the acceleration of a wide range of applications with 10x-100x performance/energy efficiency over the general-purpose processors. The design choices of FPGA accelerators for different targets at different levels are enormous. To guide the designers to find the best design choices, modeling is inevitable.

Chip-level performance and energy modeling for embedded and low-power devices. We first study the single chip performance and energy model for FPGA-based pipelined design. Customized pipeline designs that minimize the pipeline initiation interval (II) maximize the throughput of FPGA accelerators designed with high-level synthesis (HLS). However, II>1 can reduce dynamic energy below II=1 due to interconnect savings. We use analytic models to describe accelerator performance and energy, explore the trade-offs of energy and accelerator performance. and find the energy

optimal design point.

Chip-level performance and frequency improvement through locality-aware transformation in HLS. We then study timing degradation problems in HLS-based accelerator design and classify four patterns: scatter, gather, broadcast, and reduce in the context of on-chip data movement. We observe that the on-chip data path delay in these patterns scales up when the design size increases, but HLS tools do not estimate the interconnect delay correctly or make a conscientious effort to control or cap the growth of long interconnect delays at the HLS level. We propose a Latte microarchitecture that features pipelined transfer controllers (PTC) to reduce critical path and improves timing by 1.50x on average.

Node-level performance and cost modeling for FPGA-enabled, storage-optimized public cloud instances. At node level, We study performance and cost models for customized computing in light of the fact that performance and cost are primary concerns when deploying applications and services in a pay-as-you-go public cloud. The performance and cost modeling are discussed in two aspects, computation resources, with CPUs and locally PCIe-attached accelerators, and storage resources including SSDs and HDDs.

For computation resources, improved performance using accelerators is accompanied by a higher cost per hour. We discuss the performance and cost modeling of deploying FPGA accelerators, offer insights on accelerator kernel design, and discuss when we should scale up by using FPGA in a node or by choosing a larger instance which has more CPU cores per node. For storage resources, storage systems (SSD/HDD) need to be carefully chosen to match the performance improvement introduced by accelerators while achieving the optimal cost. We conduct quantitative

performance analysis on the Spark-based production-quality genome analysis toolkit. We then propose I/O-aware performance analysis and modeling for a broad set of Spark applications. Based on the model, we optimize the cost of genome sequencing in the public cloud by 38%, compared to a configuration recommended by the Spark Official website.

Cluster-level performance and cost modeling for sharing FPGAs among different instances. From a node-level performance and cost model, we learn that simply offloading accelerated kernels from CPU hosts to PCIe-based FPGAs does not guarantee improvement in terms of out-of-pocket cost when using pay-as-you-go services in a public cloud. We analyze the application execution and conclude that the extra cost is attributable to insufficient application-level speedup by Amdahl's law. To achieve cost saving with the use of FPGA accelerators in the public cloud, we propose to share one FPGA among multiple CPU instances when the number of CPU cores in one instance cannot fully utilize the FPGA accelerator computation resource. By implementing this idea, we present Mocha framework in this dissertation as a distributed runtime system to optimize the out-of-pocket cost while keeping high speedup and throughput.

To demonstrate the performance improvement and cost saving of modeling in customized computing, we use genome pipeline optimization in the public cloud and private cloud as case studies showing how to conduct optimal scheduling under certain constraints. In the public cloud, where cost is the primary concern, we formulate how to select instances and schedule genome stages to achieve the least cost given certain deadline constraints as a MILP (mixed integer linear programming) problem. In a private cloud, where hardware (CPU cores, storage disks) is given, we formulate the scheduling of multiple genomes to achieve the least latency, as a MILP problem.

The dissertation of Peipei Zhou is approved.

Anthony John Nowatzki

Jae Hoon Sul

Glenn D. Reinman

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2019

To Xuxia, Zhongze and my fiancée Hao

for always being there for me

no matter what happens

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Having experienced seven years of pursuing my Ph.D. degree at UCLA, a few paragraphs can not fully express my sincere gratitude to all the people that I want to thank. Out of these, I would like to first express my sincerest gratefulness to my advisor, Professor Jason Cong, for his vision as a computer scientist, patience and support as an advisor, and caring as one would care for a family member. As a new graduate student with barely any research experience, Professor Cong guided me into research areas focused on computer architecture, computer-aided design and analytic modeling. He showed me the systematic way to approach a research problem, the sense of describing a problem from a mathematical modeling point of view, the pursuit of finding the optimal solution. All of these are of the upmost value to me, and I will cherish what I have learned from him in my future life. I feel honored and blessed from bottom of my heart to have Professor Cong as my master's and doctoral advisor.

I am also very grateful to my doctoral committee members, Professor Glenn D. Reinman, Professor Jae Hoon Sul, and Professor Anthony John Nowatzki, for their time, patience and insightful suggestions for improving the quality of my dissertation. I thank Professor Reinman for his insightful instruction when I took an advanced computer architecture course and his invaluable research inputs when we collaborated on the Composable Heterogeneous Accelerator-Rich Microprocessor (CHARM) project in my very early stage of graduate research study. I also thank Professor Jae Hoon Sul for his research suggestions proposed in my oral exams on genome pipeline optimization in the private cloud. Additionally, I thank Professor Tony Nowatzki for

spending valuable time with me discussing the CGRA architecture, which broadened my research horizons in computer architecture.

I wish to thank all my fellow researchers who collaborated with me, contributed to this dissertation and enriched my research experience.

- Bingjun Xiao and Hui Huang are two of the most important fellow researchers who guided me in the CHARM prototyping project—which laid the foundation for my hardware and software design expertise. Bingjun spent an enormous amount of time instructing me on proposing research ideas, approaching research problems, writing research papers, and presenting research ideas.

- Peng Wei, Cody Hao Yu and I shared more than five graduate study years. We collaborated on multiple projects, and I learned so much from these fellow researchers. Projects included CS-BWAMEM, bandwidth optimization, the composable parallel pipeline (CPP) architecture, Latte and the CPU-FPGA communication pipeline.

- Professor Zhenman Fang and I collaborated on projects that included energy efficiency of the full pipeline, ARAPrototyper, Caffeine and Doppio. Zhenman and I spent a large amount of time together, discussing such things as formulations, and from him I learned how to better present research ideas in academic writing.

- Chen Zhang and I worked together on Caffeine. His knowledge in convolutional neural networks inspired me to learn more in this challenging and promising area of research.

- Yu-Ting Chen and I collaborated on CS-BWAMEM and ARAPrototyper. His endurance and work/life fitness attitude influenced me significantly.
- My collaborative work with Zhenyuan Ruan on Doppio and ST-Accel was scientificly rewarding, encouraging, and enlightening.
- Yuze Chi and I worked together on the SODA research. I also thank Yuze for his patience and constant support with the lab servers.
- Di Wu and I collaborated on Mocha. He was a supportive mentor when I worked as a software engineer intern for Falcon Computing Solutions.
- Professor André DeHon and I collaborated on research focused on energy efficiency for the full pipeline. His instructions to me on the analytic modeling of energy efficiency in FPGA architecture laid the foundation for my understanding of computer architecture analytic modeling.

I want to thank other collaborators—including Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Jiayi Sheng, Michael Lo, Brian Hill, Professor Zhiru Zhang, Professor Vivek Sarkar, Professor P. Sadayappan, Louis-Noël Pouchet, Peng Zhang, Xuechao Wei, Yuxin Wang, Max Grossman, Professor Chiyuan Ma, Professor Jie Lei, David Roazen, Megan Shand, Myron Peto, Paul Spellman, Peichen Pan, Professor Guangyu Sun, and Bojie Li.

Thanks also to my amazing fellow researchers—Young-kyu Choi, Bo Yuan, Mo Xu, Karthik Gururaj, Yuchen Hao, Zhengrong Wang, Weikang Qiao, Jason Lau, Licheng Guo, Libo Wang, Junyi Xie, Zhe Chen, Xinfeng Xie, Bug Huang, Atefeh Sohrabizadeh, Karl Marrett, and Nikola Samardzic.

I want to express my gratitude to Alexandra Luong for her seven years of fantastic work preparing the paperwork related to my graduate studies, supporting the CDSC reviews, and many other lab events.

I also want to express my thankfulness to Janice Martin-Wheeler for her incredible work editing my research papers and dissertation. She is always the most trustworthy one.

Thanks also to my fellow researchers and best friends Jie Wang, Muhuan Huang, and Meng Li for supporting me and cheering me on through all the challenges and down moments.

I especially thank my parents, Zhongze and Xuxia. They are the best role models. Their love and understanding mean everything to me.

I want to thank myself for never giving up.

Finally, I would like to quote the song lyrics (with adaptions) from *Can't Stop Love* [Dar10] given by my favorite singer Darin, "We stand here today, together as one. Hao, you brighten my days, just like the sun. They said this love was the impossible kind. But we are strong enough to fight for this life. I can't stop this love. No matter what they say. I love you."

The research studies in this dissertation are partially supported by the Customizable Domain-Specific Computing Project (NSF-0926127); the Intel Corporation with matching funds from the NSF under the Innovation Transition (InTrans) Program (CCF-1436827); funding from CDSC industrial partners including Samsung, Google, Huawei, Baidu, Fujitsu Labs, Mentor Graphics, and VMWare; C-FAR, one of the six centers of STARnet—a Semiconductor Research Corporation program sponsored by MARCO and DARPA; and AWS Research Credits on Amazon EC2 services.

VITA

2008–2012 B.S., School of Electronic Science and Technology,
Chien-Shiung Wu Honor College, Southeast University, Nanjing, China.

2012–2014 M.S., Electrical Engineering Department,
University of California, Los Angeles, U.S.A.

2014–present Graduate Student Researcher, Computer Science Department,
University of California, Los Angeles, U.S.A.

PUBLICATIONS

Y. Chi, J. Cong, P. Wei, P. Zhou, SODA: stencil with optimized dataflow architecture, *ICCAD*, 2018. **Best Paper Nominee**

Y. Chi, P. Zhou, J. Cong, An Optimal Microarchitecture for Stencil Computation with Data Reuse and Fine-Grained Parallelism (Poster), *DAC*, 2018 and *FPGA*, 2018.

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

Z. Ruan, T. He, B. Li, P. Zhou, J. Cong, ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA, *FCCM*, 2018.

P. Zhou*, Z. Ruan, Z. Fang, M. Shand, D. Roazen, J. Cong, Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-Memory Computing Framework, *ISPASS*, 2018. **Best Paper Nominee**

C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, J. Cong, Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks, *TCAD*, 2018 and ICCAD, 2016. **2019 TCAD Donald O. Pederson Best Paper Award**

J. Cong, P. Wei, H. Yu, P. Zhou, Bandwidth Optimization Through On-Chip Memory Restructuring for HLS, *DAC*, 2017.

P. Zhou*, H. Park, Z. Fang, J. Cong, A. DeHon, Energy Efficiency of Full Pipelining: A Case Study for Matrix Multiplication, *FCCM*, 2016.

Y. Chen, J. Cong, Z. Fang, and P. Zhou, ARAPrototyper: Enabling Rapid Prototyping and Evaluation for Accelerator-Rich Architecture (Poster), *FPGA*, 2016.

Y. Chen, J. Cong, J. Lei, S. Li, M. Peto, P. Spellman, P. Wei, and P. Zhou, CS-BWAMEM: A fast and scalable read aligner at the cloud scale for whole genome sequencing (Poster), *HiTSeq*, 2015. **Best Poster Award**

J. Cong, H. Huang, C. Ma, B. Xiao, P. Zhou*, A Fully Pipelined and Dynamically Composable Architecture of CGRA, *FCCM*, 2014.

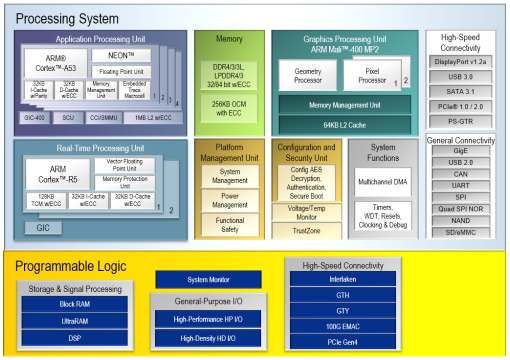
CHAPTER 1

Introduction

The classic von Neumann architecture allows efficient sharing of the executions of different instructions on a common pipeline. However, general-purpose computing still faces challenges in heat dissipation, performance, and cost issues. One architectural trend is to offload most computation tasks from general-purpose CPU cores to accelerators. Field programmable gate arrays (FPGAs) have been widely used as hardware accelerators for different applications by customization before task executions; this brings 10x-100x performance/energy efficiency [CSR11] over the general-purpose processors. FPGA accelerators span a broad spectrum, and can be categorized by design levels into chip level, server node level, and cluster level. We first discuss FPGA accelerators in each design level and identify corresponding design targets. Then we discuss modeling and optimization for each design level in the following subsections.

Chip Level: Embedded FPGAs are FPGA chips or FPGA IP blocks that are integrated in the system-on-chip (SoC) used in mobile devices, Internet-of-Things (IoT) or other battery-backed devices. Figure 1.1 the shows Xilinx Zynq UltraScale+ multiprocessor system-on-chip (MPSoC) and its architecture diagram [Xilb]. For accelerators in embedded FPGAs, performance and power efficiency are the primary design issues. We first study the energy efficiency of full pipelining and propose

(a) Zynq UltraScale+ ZU9EG MPSoC chip



(b) Zynq UltraScale+ MPSoC Architecture

Figure 1.1: Xilinx Zynq UltraScale+ multiprocessor system-on-chip (MPSoC) comprised of ARM cores and 16nm programmable logic [Xilb].

performance and energy modeling for chip-level design. Then we study timing degradation problems in the HLS-based accelerator design. We propose the Latte microarchitecture to add pipelined transfer controllers in data paths to improve timing of baseline designs.

Node Level: The inclusion of FPGA into public data centers marks the beginning of a new era of democratizing customizable computing. For example, Amazon Web Services (AWS) Elastic Compute Cloud (EC2) provides F1 FPGA instances (as shown in Figure 1.2 [Amaa]). AWS F1 instance features a CPU host and locally PCIe-attached FPGA (Xilinx Virtex Ultrascale+ VU9P) [Amaa]. It is priced higher than CPU-only general-purpose instances. For FPGA accelerators at the server node level, performance and cost efficiency are the primary issues.

Cluster Level: In addition to treating FPGAs as local coprocessors connected to CPUs through the PCIe interface, it is also possible to share FPGAs among different nodes. For example, as shown in Figure 1.3, we can allocate one host CPU thread in an Amazon AWS F1 instance to communicate with local FPGA and

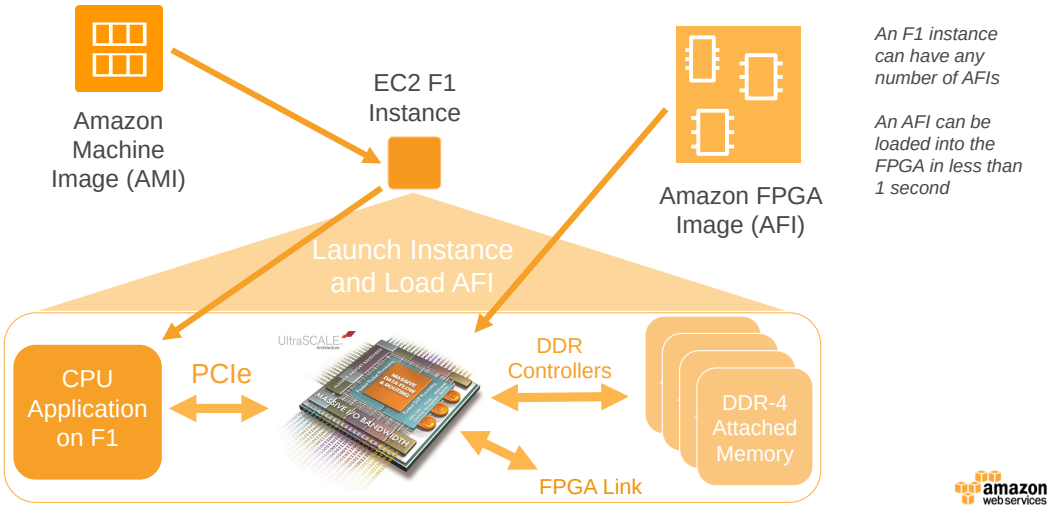


Figure 1.2: Amazon AWS EC2 F1 FPGA instance [Amaa].

multiple general-purpose instances, e.g., M2 instances, through the network. Thus, we achieve sharing FPGAs among nodes to enable full pipelining of CPU and FPGA computation. Another finer-grained sharing can be achieved in architectures where FPGA resources are decoupled from the CPU hosts [WAH15, CCP16]. For example, Microsoft proposes a configurable cloud [CCP16] where the FPGA is placed between the network interface card and network switch, as shown in Figure 1.4. Each FPGA can process network packets without host intervention.

Similar to accelerators in the node level, for accelerators deployed at the cluster level, performance and cost efficiency are among the most important issues. Matching throughput of CPUs and FPGAs among nodes improves the overall cost efficiency.

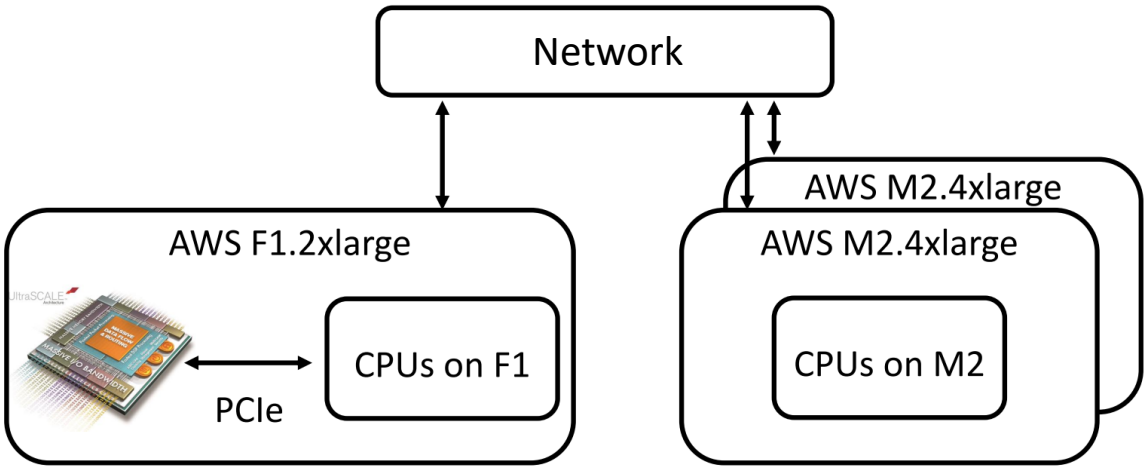


Figure 1.3: Sharing FPGA among instances through network.

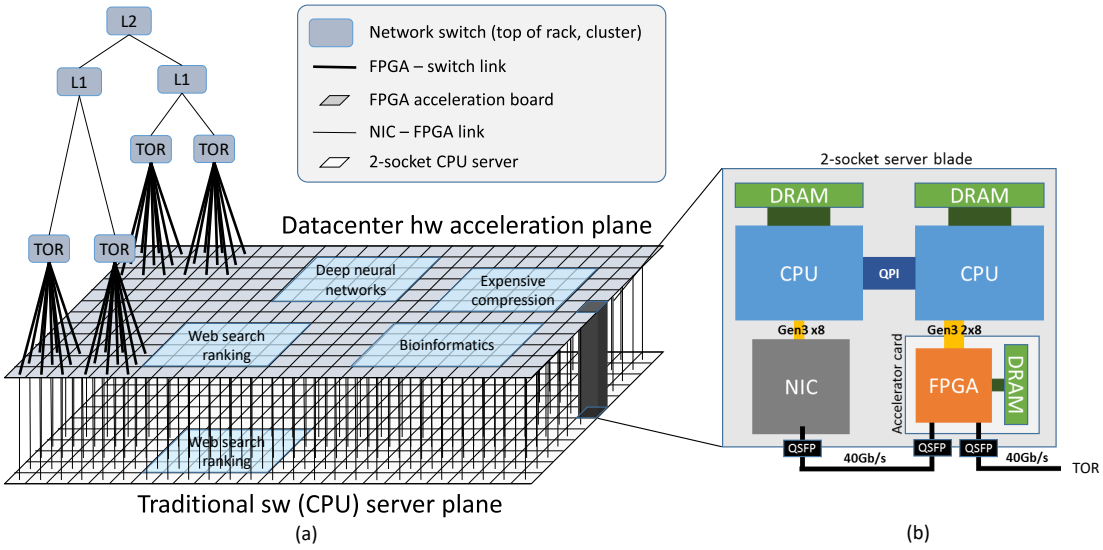


Figure 1.4: Microsoft configurable cloud [CCP16].

1.1 Chip-Level Performance and Energy Modeling

1.1.1 Energy Efficiency of Full Pipelining

To meet the ever-increasing demand for high computation performance and energy efficiency, numerous commodity acceleration platforms have been proposed and developed, including the well-known many integrated cores (MICs, or Intel Xeon Phi processors), graphical processing units (GPUs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs) [DK12, CLS08, PYC06]. The utilization wall [Tay12] has stimulated interest in FPGAs, since FPGAs provide both low power and customization capability to accelerate different applications (more flexible than ASICs).

Compared to the general-purpose MICs and GPUs, FPGAs allow designers to look beyond parallelization and customize accelerators. The customized pipeline design has been one of the most successful and widely used optimizations to improve the performance of FPGA accelerators [LZP15, CHM14, CJL11]. At the same time, the recent success of commercial HLS tools like Xilinx Vivado HLS [Xila] has made design space exploration for a customized pipeline easier compared to conventional register transfer level (RTL) designs.

Among various tunable parameters in such pipeline customization, the pipeline initiation interval (II)—which is defined as the number of cycles between two consecutive pipeline iterations [Lam88]—is one of the most important customization parameters since it reflects the throughput of the pipeline design and has been widely studied (e.g., [LZP15, CHM14, CJL11]). Most prior studies, except [LZP15], have focused on minimizing the pipeline II so as to maximize the throughput of FPGA

accelerators. Meanwhile, there are also examples [CHM14] indicating that a smaller pipeline II can reduce the energy consumption of FPGA logic gates. Motivated by these studies, Chapter 2 begins to explore a key question: *Does a customized pipeline with the minimum II always minimize energy? If not, how does the pipeline II affect energy consumption?*

To get initial insight into this question, we focus on the classical matrix-multiplication (MM) algorithm specified in C for HLS. We build an analytical model of the energy consumption for the kernel as a function of matrix dimension, N, and pipeline II, including the effects of computation, interconnect, memory, and leakage energy. This allows us to identify how the energy should scale with problem size and II. We synthesize the HLS kernel with Vivado HLS and fit constants within the analytical model.

We find that energy for the logic component remains flat, while energy for the memory and leakage components increases with II, but interconnect energy can decrease with increasing II. Interconnect savings are large enough that we can identify cases where $II > 1$ minimizes energy. Nonetheless, we see that the increasing energy term is modest, such that the $II = 1$ case is always within a few percent of energy optimal design point for MM that can fit on a single FPGA today. The energy framework identified here should translate to other HLS kernels, but will have different compute and interconnect scaling that should be characterized and better understood in future work.

In summary, Chapter 2 makes the following contributions.

1. The first set of high-level analytical model that studies the impact of the pipeline II on the energy consumption for a customized MM accelerator pipeline designed

in HLS on commodity FPGAs, which maintains an average error rate within 5%.

2. Insights into choosing the pipeline II to minimize energy for the MM accelerator—when interconnect energy decreases at first and other parts of energy remain flat, we can achieve energy optimal design where $II > 1$.

3. Observations of limitations in commercial HLS tools—inefficient register sharing, and lack of power-gating for unused memory banks. We explain the register sharing inefficiency in the baseline MM kernel in Section 2.2.1, show how we fix this by code rewriting in Section 2.2.2, and discuss the energy model before code rewriting in detail in Section 2.5.

   This work is published in *Proceeding of 2016 International Symposium on Field-Programmable Custom Computing Machines* [ZPF16].

1.1.2 Locality-Aware Transformation for High-Level Synthesis

Field-programmable gate arrays (FPGAs) have gained popularity in accelerating a wide range of applications with high performance and energy efficiency. High-level synthesis (HLS) tools, including Xilinx Vivado HLS [Xila] and Intel OpenCL [Int], greatly improve FPGA design feasibility by abstracting away register-transfer level (RTL) details. With HLS tools, a developer is able to describe the accelerator in C-based programming languages without considering many hardware issues, such as clock and memory controller, so the accelerator functionality can be verified rapidly. Furthermore, the developer can rely on HLS pragmas that specify loop scheduling and memory organization to improve the performance. In particular, kernel replication is one of the most effective optimization strategies to reduce the overall cycle latency and improve resource utilization. However, as reported in

previous work [Wan16, Zoh16, Wan17, TT14], the operating frequency of a scaled-out accelerator after place and route (P&R) usually drops, which in the end diminishes the benefit from kernel replication.

Fig. 1.5 illustrates the frequency degradation for a broad class of applications (details in Section 3.3). Each dot point shows frequency and corresponding resource utilization for an application with a certain processing element (PE) number. The (black dashed) trend line characterizes the achieved frequency under certain resource usage. On average, HLS generated accelerators sustain a 200 MHz on 30% resource usage. However, the frequency drops to 150 MHz when usage increases to 74% (shown in two triangle markers). An extreme case is when dot A runs at as low as 57 MHz when using 88% resource.

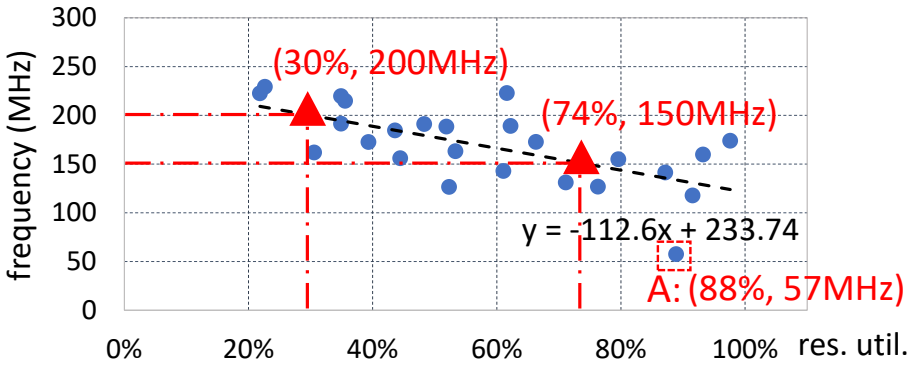


Figure 1.5: Frequency vs. area: Freq decreases as design size scales out.

We investigate such cases and spot the *Achilles' heel* in HLS design that has attracted less attention—in particular, one-to-all or all-to-one data movement between off-chip DRAM interface and on-chip storage logic (BRAM or FF), or inside the computation logic. Using terminology from the message passing interface (MPI) [Sni98], we introduce four collective communication and computation patterns: scatter, gather,

broadcast and reduce. They are used in most, if not all, accelerators. Different from MPI, the four patterns in HLS are in the context of **on-chip data movement**, instead of movement between servers. We observe that the on-chip data path delay in these patterns scales up when the design size increases, but HLS tools do not estimate the interconnect delay correctly or make a conscientious effort to control or cap the growth of long interconnect delays at HLS level.

A common solution to long critical paths is to insert registers in the datapath in the RTL [Che13], logic synthesis or physical synthesis phase. However, it requires nontrivial efforts in buffering at the RTL level, which calls for a high-level solution. Also, prior work in systolic array applications [TT14, Tuc17, Ruc15, ZCL15] and compilers [LJS89] feature neighbor-to-neighbor interconnect in tightly coupled processing units and eliminate global interconnect. However, classic systolic array requires the application to have a regular data dependency graph, which limits the generality of the architecture. Moreover, neighbor-to-neighbor register buffering introduces logic overhead to each processing unit and incurs non-negligible overhead in the total area.

To address the above-mentioned challenges in low-level buffer insertion, generality and non-negligible area overhead, we propose the Latte microarchitecture. In data paths of the design, Latte features pipelined transfer controllers (PTC), each of which connects to only a set of PEs to reduce critical path. Intrinsically, Latte is applicable to general applications as long as the patterns occur. In addition, to improve the resource efficiency, we also explore the design choices of Latte in the number of PTC inserted, and offer performance/area-driven solutions. We implement Latte in HLS and automate the transformation for PTC insertion, which eases the programming efforts. In summary, Chapter 3 makes the following contributions:

- Identifying four common collective communication and computation patterns—scatter, gather, broadcast and reduce in HLS that cause long critical paths in scaled-out designs.
- Latte microarchitecture featuring customizable pipelined transfer controllers to reduce critical path.
- An end-to-end automation framework that realizes our HLS-based Latte implementation.

Our experiments on a variety of applications show that the Latte-optimized design improves timing of the baseline HLS design by 1.50× with 3.2% LUT overhead on average, and 2.66× with 2.7% overhead at maximum.

This work is published in *Proceeding of 2018 International Symposium on Field-Programmable Custom Computing Machines* [CWY18a].

1.2 Node-Level Performance and Cost Modeling

1.2.1 Computation Resources

Recent trends have shown that FPGA-powered computation intensive nodes are launched in many public clouds, e.g., Amazon Elastic Compute Cloud (EC2), Huawei, Baidu, Alibaba, etc. Unfortunately, even though we have a well- designed, high-performance FPGA accelerator, the system-level speedup could be moderate due to inefficient CPU-FPGA communication and data transfer. As a result, prior work applied several approaches to improve the system efficiency by overlapping communication routines between CPU host and FPGA [CWY18b], caching reused

data [CHW, HWY16], and orchestrating multiple CPU cores and FPGA based on dataflow execution models [CFH18].

However, very few prior work investigated out-of-pocket cost, one of the most important issues in public cloud while using FPGAs. To illustrate the cost issue in public clouds with FPGAs, we conduct case studies that adopt FPGA accelerators in widely used genome variant calling programs in a Genome Analysis Toolkit (GATK) [MHB10]: HaplotypeCaller (HTC) [Ins19] and Mutect2 [CLC13]. As one of the most significant tools in computational genome analysis, GATK has helped make progress in advancing precision medicine. In particular, HTC and Mutect2 are two of the most time-consuming applications in GATK that aim to find germline variants for pair-end sequence reads (HTC) and tumor sequence reads (Mutect2). Both applications include a high-complexity algorithm called Pair Hidden Markov Model (PairHMM) [DEK98] that can be an accelerator candidate on FPGA. Unfortunately, while Mutect2 costs only 0.73× on an `f1.2xlarge` AWS EC2 instance with an FPGA when compared to the cost on an `m4.2xlarge` general-purpose CPU instance, HTC costs 2.56×, which incurs extra cost overhead even though the execution time is reduced. As a result, a key question is being raised: *How does using FPGA accelerators impact an application's out-of-pocket cost in public cloud services?*

Table 1.1: Price comparison of CPU and FPGA instances on public clouds

	CPU Instance	**CPU-FPGA Instance**
AWS EC2 [Ama19b]	8 vCPU, $0.4/hr	8 vCPU + 1 Xilinx VU9P, $1.65/hr (4.125×)
Huawei Cloud [Hua19]	32 vCPU, $1.64/hr	32 vCPU + 1 Xilinx VU9P, $2.83/hr (1.725×)

Since FPGA instances are priced higher than general-purpose CPU instances

in most cloud vendors, applying FPGA accelerators has to bring a high enough application-level speedup to achieve cost saving. Taking AWS EC2 as an example, Table 1.1 shows that its FPGA instance is priced at $1.65/hour, which is **4.125**× over the price of a CPU instance with the same type and number of virtual CPU cores (vCPU). In this case, if adopting FPGA in AWS does not achieve 4.125× application-level speedup, this solution is not as cost-efficient as pure CPU solutions. In fact, our baseline CPU-FPGA system, which lets all eight CPU cores send all PairHMM tasks to the FPGA, only demonstrates 1.6× application-level speedup for HTC, causing 2.56× costs over the CPU-only system. Note that since the proportion of the PairHMM kernel in the whole application is 39%, the optimal application-level speedup of HTC is only 1.64× by Amdahl's law.

In summary, Chapter 4 makes the following contribution: We analyze out-of-pocket cost in using node-level CPU-FPGA platforms on the public cloud, summarize two cases when computation throughput of CPU and FPGA does not match, and explain how these two cases bring extra cost over pure CPU solutions.

1.2.2 Storage Resources

Within the past decade, there has been great success in programming frameworks that support efficient development and deployment of large-scale applications in datacenters. Examples include the pioneering MapReduce framework [DG08a] initially proposed by Google, the open-source Hadoop MapReduce framework [Whi12a], and the more recent Apache Spark framework that improves the performance of Hadoop by up to 100x through in-memory cluster computing [ZCD12a]. Due to its high performance efficiency, Spark has attracted increased attention from both academia and industry.

In such big data computing frameworks, I/O used to play an important role in system performance, and it attracted a significant amount of research [KC14, ORR15, WK15, Awa16]. For example, Kambatla et al. reports in [KC14] that SSDs can deliver up to 70% performance improvement for Hadoop MapReduce workloads. On the other hand, a recent work [ORR15] from the Apache Spark community claims that eliminating I/O accesses in Spark SQL processing workloads can reduce job completion time by a median of at most 19%. Thus, I/O tends to no longer be the bottleneck for the Spark in-memory computing framework. Such studies present quite different implications of the I/O impact based on their application domains and hardware resources; this often confuses users. As a result, the following key question arises: *How does I/O impact the big data application performance running on top of in-memory cluster computing frameworks like Apache Spark?*

Having a quantitative understanding of a complex distributed system like Spark is not trivial. Unfortunately, previous studies in modeling Spark performance [WK15, VYF16, GRB17] usually overlook the impact of I/O in their models. To better answer the above question, we first measure the performance impact of I/O on Spark by conducting an in-depth case study on the Spark-based production-quality Genome Analysis ToolKit (GATK4) [Ins17b]. GATK4 is one of the most important tools in computational genomics, and it has great potential for providing personalized medicine [GMP10]. Since it involves various types of Spark operations, it is typical and complex enough for our motivation study. A brief introduction to Apache Spark and the GATK4 application will be presented in Section 5.2.

Different than [ORR15], we observe that I/O can still play a heavy role—even in the in-memory computing framework of Spark. In addition to the HDFS read and write for the input and output data which introduce I/O access, we make three

other observations; these are analyzed in Section 5.3. First, to avoid the time-consuming recomputation, certain RDD (resilient distributed dataset) operations like groupByKey will perform shuffle write/read to write/read intermediate data to/from storage between different Spark stages. Second, there is usually some non-cached intermediate data (RDD) as it is too large to be totally put into memory. For example, for GATK4, to cache a 30x coverage whole genome intermediate data, it requires at least 3.2TB total CPU memory. And even more for higher coverage genome inputs. Third, effective I/O bandwidth differs under various data request sizes for different I/O devices including HDD and SSD. For example, bandwidth difference between HDD and SSD for HDFS read operation in GATK4 is 3.7x, however, 32x for shuffle read operation. In that way, performance under HDD and SSD varies a lot for some spark stages while not much for others.

To gain more insights into how I/O impacts the performance of Spark applications, we propose a generic I/O-aware analytical model to reason the underlying behavior of different Spark RDD operations and model their performance in Section 5.4. In this model, we analytically combine all the following factors together, which often have been overlooked in past studies.

1. We incorporate the effective I/O bandwidths under different data block sizes during different RDD accesses, including HDFS read/write, shuffle read/write, and persist read/write.
2. We incorporate the I/O bandwidth contention from different CPU cores, and quantify the break point (i.e., number of CPU cores) after which the I/O bandwidth is saturated.
3. We also incorporate the overlap between the CPU computation and I/O accesses

from different data partitions, where we assume a simple but effective pipeline execution model.

To validate the accuracy of our proposed I/O-aware analytical model, we choose a set of representative Spark applications, including GATK4 and five other typical applications of iterative algorithms and shuffle-heavy algorithms. As detailed in Section 5.5, experimental results show that our model achieves a performance prediction error rate within 10%, and well explains the runtime behavior of different stages in those applications.

This **quantitative** model enables us to better understand the performance of Spark applications for further optimizations. Moreover, our model can also be applied in the public cloud. In Section 5.6 we conduct a case study to use our model to optimize the cost of genome sequencing in Google Cloud, where the cost of execution can be modeled as $Cost = f\ (CoreNum,\ DiskSize_{HDFS},\ DiskSize_{Spark\_Local},\ Time)$. Using the basic application profile and platform configuration $(CoreNum,\ DiskSize_{HDFS},\ DiskSize_{Spark\_Local})$, our model can derive the application execution *Time*. Therefore, we can explore the platform configuration space to find the optimal one with the minimum cost. Experimental results demonstrate that we can save 38% to 57% cost compared to recommended default configurations from Spark [Spa17c] and Cloudera [Clo13].

Although we only discuss the above usages, our model can be utilized for other purposes as well. For example, in a shared cluster environment with a job scheduler, our performance prediction model can allow the scheduler to know ahead the approximating job execution time and thus enable better job scheduling with less job waiting time.

In summary, Chapter 5 makes the following contributions:

1. A quantitative performance analysis on the Spark-based production-quality genome analysis toolkit.
2. An accurate I/O-aware performance analytic model for a broad set of Spark applications.
3. A model-driven cost optimization study for genome sequencing in the public Google Cloud.

In addition, we open source our toolset Doppio[1] incorporating the I/O-aware model to the community, which can be accessed at [cds17].

This work is published in *Proceedings of 2018 IEEE International Symposium on Performance Analysis of Systems and Software* [ZRF18a].

1.3 Cluster-Level Performance and Cost Modeling

For node-level straightforward CPU-FPGA integration, the mismatch of computation throughput of CPU and FPGA might bring extra cost over pure CPU solutions. For example, in the Amazon EC2 f1.2x FPGA instance, the PairHMM accelerator on FPGA is capable of achieving 40× speedup over a single-core CPU, and eight vCPU cores cannot fully utilize the computation resource in the accelerator, which results in 2.56× costs over a CPU-only system. In fact, we could borrow more vCPU cores from other pure CPU instances via network to fully utilize the accelerator. In this case, we could achieve cost saving by accelerating many HTC tasks on a system with one FPGA and multiple CPU instances.

[1]Doppio is the anagram for I/O-aware Performance analysis, moDeling and OPtimization for in-memory computing framework.

Built upon this idea, in Chapter 6, we design and implement a framework called Mocha to guarantee the cost saving for arbitrary applications with FPGA accelerators in the public cloud. Mocha first profiles the given application and identifies the performance bottleneck (CPU or FPGA). For the CPU bottleneck and applications such as HTC Mocha improve FPGA utilization by sharing one FPGA among multiple nodes through network. For the FPGA bottleneck, applications like Mutect2, Mocha orchestrate CPU cores to schedule originally offloaded tasks back to CPU cores (partial task offloading) instead of sending tasks to FPGA. As a result, *Mocha could improve the overall system resource utilization and thus reduce the application cost for any applications, no matter how small proportioned the kernel is as long as the FPGA kernel speedup is higher than the cost ratio.*

To demonstrate the cost efficiency improvement from using Mocha, we give concrete and solid accelerator integration case studies on HTC and Mutect2 in GATK in Section 6.4. As detailed in Section 6.4.2, experiment results show that, on different public cloud platforms, when compared to straightforward CPU-FPGA integration with our PairHMM accelerator, Mocha framework saves the cost of HTC by 2.82x and cost of Mutect2 by 1.06x on AWS. On Huawei Cloud, Mocha framework saves cost by 1.22x for HTC and 1.52x for Mutect2. We make makes the following contributions:

- We propose performance and cost modeling that balances the throughput of CPU cores (by partially offloading kernel tasks) and FPGA (by sharing FPGA among multiple nodes) to achieve full computation resource utilization (Section 6.1).
- We propose a framework called Mocha that enables FPGA sharing among multiple nodes through network and partial task offloading policy for CPUs in

order to fully utilize FPGA and CPUs for any applications. Mocha guarantees that the cost efficiency of a CPU-FPGA solution is higher than a pure CPU solution as long as the FPGA kernel speedup is higher than the cost ratio (Section 6.2, 6.3).

- We design and implement a cutting-edge PairHMM accelerator on Xilinx FPGAs. By adopting an effective architecture with various optimization techniques, our accelerator almost exhausts the floating-point computing power of both AWS F1 and Huawei FP1 FPGAs (Section 6.4).

- We present performance and cost efficiency of our accelerator and provide model-driven cost optimization case studies for Genome Variant Calling applications HTC and Mutect2 in two public cloud platforms—Amazon EC2 and Huawei Cloud. By using Mocha, our CPU-FPGA solution achieves the state-of-the-art performance while saving cost by 2.82x for HTC, 1.06x for Mutect2 on AWS, and 1.22x, 1.52x respectively on Huawei Cloud. (Section 6.4.2).

1.4 Applications: Modeling and Optimization in Public Cloud and Private Cloud

Finally, we show how to utilize modeling for cluster-level CPU-FPGA integration together with storage resource to have optimal scheduling in public cloud and private cloud. We use applications in whole-genome sequence pipeline as studied applications.

1.4.1 Public Cloud

In the public cloud where different instances have a different number of CPU cores, CPU types, disks and prices, out-of-pocket cost is the primary concern. In Chapter 7 we formulate how to select instances and schedule genome pipeline stages to achieve the least cost given certain deadline constraints as a MILP (mixed integer linear programming) problem. In this chapter, we make the following contributions:

- We present an MILP formula of optimal cost scheduling under deadline constraints, while taking into consideration different CPU instances, disk size and type, and execution time of genome pipeline stages on the studied instances (Section 7.1).
- We evaluate the modeling on the Amazon EC2 cluster (Section 7.2). We first profile the execution time of genome pipeline stages in different instances (Section 7.2.1), and solve the optimal cost for a single genome (Section 7.2.2) given deadline constraints from loose to tight. We show in Section 7.2.3 that composable instances enabled by Mocha framework save the cost by 20% in GATK pipeline. We also consider, in Section 7.2.4, that when there are multiple genomes, how does optimal average cost per genome change.

1.4.2 Private Cloud

In the private cloud, once the infrastructure is built, hardware (CPU cores, storage disks) is given as fixed. The goal is to minimize the latency and improve the throughput. In Chapter 8, we formulate how to schedule multiple genomes to achieve the least latency as a MILP problem. In that chapter, we make the following

contributions:

- We present the MILP formula of optimal latency scheduling while taking into consideration different CPU cores and storage space in a server (Section 8.1).
- We evaluate the modeling in our local UCLA CDSC cluster (Section 8.2). After examining the optimal scheduling results for cases when number of genome is small (Section 8.2.1), we propose three heuristics in Section 8.2.2 for cases the when number of genome increases.
- We verify the modeling by running experiments on NA12878-Garvan [Sta16] and comparing the difference in Section 8.2.3. The experiment results show that our modeling achieves less than 4% error rate. When the number of genomes is large, we propose two methods in Section 8.3 to construct the scheduling by reusing base case configurations.

Chapter 9 concludes the dissertation.

CHAPTER 2

Energy Efficiency of Full Pipelining

The power and utilization walls have led to a drastically growing interest in customizable accelerator designs on FPGAs. The customized pipeline design has been one of the most important optimizations and widely used to improve the performance of FPGA accelerators. In particular, prior studies have focused on minimizing the pipeline initiation interval (II) so as to maximize the throughput of accelerators. However, the impact of the pipeline II on the energy efficiency of accelerator designs remains unclear. In this chapter, we propose a set of high-level yet accurate analytical models to investigate the impact of the pipeline II on the energy consumption of FPGA accelerators designed in high-level synthesis (HLS). Using a matrix-multiply accelerator, we show that matrix multiplies with II>1 can sometimes reduce dynamic energy below II=1 due to interconnect savings, but II=1 always achieves energy close to the minimum. We also identify sources of inefficient mapping in the commercial HLS tool flow.

2.1 Related Work in Energy Modeling

FPGA energy models have been widely used to provide guidance in design space explorations. Recent studies in [DeH15] developed analytical models to characterize

energy consumption of designs ranging from a sequential design (processor) to a spatial design (FPGA), using Rent's rule [LR71] as a modeling tool. Earlier works [PWY05, LLH05, RA11] employed models to provide in-depth analysis of FPGA power decomposition and the impact of look-up table (LUT) size, cluster size, and segment lengths on power consumption. Recent work introduced FPGA memory models to analyze the effect of the memory architecture (including block size, banking, physical spacing) and parallelism on an application's energy efficiency [KLD15]. While these works present detailed energy models, they do not directly address the microarchitectural structure that results from tuning the II in HLS designs. To the best of our knowledge, this work is the first to model the impact of the pipeline II on energy consumption of FPGA accelerators from a high-level perspective.

2.2 Matrix-Multiplication Kernel

2.2.1 Baseline Kernel

```
1 void matrix_multiply(float a[N][N], float b[N][N], float c[N][N]) {
2   int i, j, k;
3   k_loop: for(k = 0; k < N; k++) {
4     i_loop: for(i = 0; i < N; i++) {
5       #pragma HLS PIPELINE II = II_i
6       j_loop: for(j = 0; j < N; j++) {
7         #pragma HLS UNROLL
8         c[i][j] += a[i][k] * b[k][j];
9 } } } }
```

Figure 2.1: Pseudo code of square matrix multiplication.

To make it easier to understand, we use square $N \times N$ matrix-multiplication as an

example to demonstrate the energy model for mapping applications onto a commodity FPGA. We first develop the HLS code as shown in Figure 2.1. To pipeline the *i_loop* with a specific II (e.g., II_i = 1), we apply the HLS PIPELINE pragma below the *i_loop* and set different II_i value in the pragma. Consequently, the inner-most *j_loop* inside the *i_loop* will be unrolled automatically (HLS UNROLL pragma is shown to illustrate this unroll but not needed explicitly). In the *j_loop*, one row of c matrix, $c[i][0]$, $c[i][1]$, ..., $c[i][N-1]$ are updated using $a[i][k]$ and one row of b matrix, $b[k][0]$, $b[k][1]$, ..., $b[k][N-1]$. The PIPELINE II of *i_loop* (i.e., II_i, for simplicity, we will directly use II for II_i in the rest of the paper) determines the throughput, resource utilization and energy to execute this matrix multiplication kernel. If PIPELINE II is set to 1, one row of c matrix (i.e., one iteration of the *i_loop*) is updated every cycle and it needs the following resources:

1. N distinct sets of PEs, where each set of PEs includes a floating point multiplier and a floating point adder.
2. Simultaneous memory (FPGA on-chip BRAM) accesses to each element within a row of b matrix and c matrix every cycle. This requires full memory partition along the column direction for b and c matrix. In this way, each column of b or c matrix are stored in an independent memory bank.

To get a deep understanding of the pipeline resource consumption when II changes under a given N, we analyze the microarchitectures of the matrix multiplication example in two extreme cases: II = 1 and II = N. For illustration purposes, we set N to 6 in our analysis.

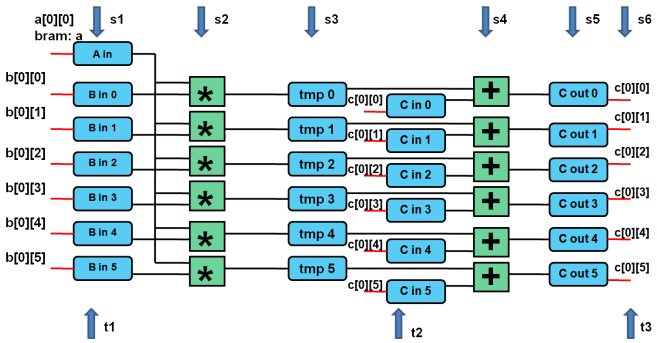


Figure 2.2: Snapshot for N = 6, II = 1, k = 0, i = 0

2.2.1.1 Analysis of II = 1

We first analyze the computation pipeline for the *i_loop* when II = 1 and N = 6, by taking a snapshot of one *i_loop* execution when k = 0 and i = 0. As demonstrated in Fig. 2.2, to finish one *i_loop* iteration, there are 6 logical pipeline stages as explained below.

1. $a[0][0]$ is read from memory into input register *Ain*. Meanwhile, all the elements in the first row of b matrix $b[0][0]$, $b[0][1]$, ..., $b[0][5]$ are read from memory into input registers *Bin0* to *Bin5*.
2. *Ain* is broadcast to all the multipliers while each *Bin* register goes to a distinct multiplier. And all 6 pipelined floating-point multipliers will do the computation concurrently. Note that each pipeline stage here is a logical stage and the physical multiplier in this logical stage is also piplined to achieve II = 1.
3. Results from multipliers are written into temporary result registers *tmp0* to *tmp5*. Meanwhile, all the elements in the first row of c matrix $c[0][0]$, $c[0][1]$, ..., $c[0][5]$ are read from memory into input registers *Cin0* to *Cin5*.

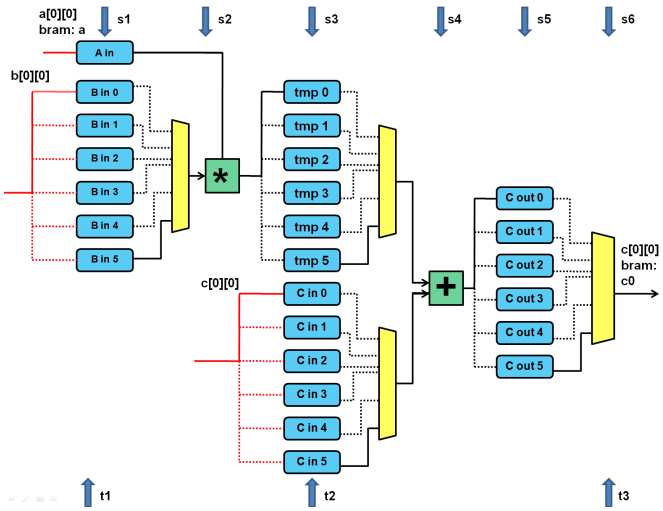


Figure 2.3: Snapshot for N = 6, II = 6, k = 0, i = 0, j = 0

4. Each *tmp* register and *Cin* register go to a distinct adder. And all 6 pipelined floating-point adders will do the computation concurrently.

5. Results from adders are written into output registers *Cout0* to *Cout5*.

6. All *Cout* registers are written into distinct c memory banks simultaneously.

In Fig. 2.2, the red lines show the wires from memory ports to the input registers or output registers to memory ports. $t1$, $t2$, $t3$ are time markers when the input are read from memory into registers and results are written back to memory. Since PIPELINE II in the *i_loop* is 1, the *i_loop* takes new inputs every clock cycle. That is, for the next iteration when k = 0 and i = 1: at $t1 + 1$, the row of b matrix $b[1][0]$, $b[1][1]$, ..., $b[1][5]$ are read into input registers; at $t2 + 1$, the next row of c matrix $c[1][0]$, $c[1][1]$, ..., $c[1][5]$ are read into input registers and the corresponding results are written back to memory at $t3 + 1$.

2.2.1.2 Analysis of II = N

Now we analyze another extreme pipeline for the *i_loop* when II = N, i.e., II = N = 6. In this case, every N (II = N) cycles, the pipeline processes one iteration of the *i_loop*; that is, every cycle, the pipeline processes one iteration of the *j_loop*. We illustrate the pipeline execution by taking a snapshot of one *j_loop* execution when k = 0, i = 0 and j = 0, as shown in Fig. 2.3. Though the architecture generated by Vivado HLS has only one multiplier and adder, the register resource remains unchanged. Here, there are N (6) *Bin* registers (similar for *tmp*, *Cin* and *Cout* registers) in the architecture. And one *j_loop* execution is like the following:

1. Every cycle one b element is read from memory into one input register like *Bin0*, ..., *Bin5*. The solid line marks the input register that has been written with valid value $b[0][0]$ and dash lines mark the registers that have not been changed. While for one a element, it is read from memory for every II (6) cycles and is reused for a row of b matrix $b[0][0]$, $b[0][1]$, ..., $b[0][5]$.
2. There is only one multiplier. One value from *Bin0* to *Bin5* will be selected using an II-to-1 MUX and used as the input to the multiplier. *Ain* is the other input to the multiplier.
3. One *tmp* register will be written with result from the multiplier, and one *Cin* input register will be changed with the new input $c[0][0]$.
4. There is only one adder. It will select the correct input values from two II-to-1 MUXes.
5. One *Cout* output register takes the result from the adder.
6. One *Cout* output register will be selected from an II-to-1 MUX to be written back

to the c memory.

For the next iteration: at $t1 + 1$, the second element in the b matrix, $b[0][1]$, is read into its input register; at $t2 + 1$, the second element in the c matrix, $c[0][1]$, is read into its input register and the corresponding result is written back to memory at $t3 + 1$. It takes II cycles (here II is N) in total for one b or c matrix row to finish.

Here, we identify the inefficiency in the generated architecture. As II increases from 1 to 6, the *Bin*, *tmp*, *Cin* and *Cout* registers are not shared as floating point multipliers and adders. As II is larger than 1, there are II-to-1 MUXes needed to select the correct input, which will incur extra energy in MUXes compared to II = 1. In addition, 1-to-II fan-out and II-to-1 fan-in interconnect will further increase the energy.

2.2.2 Optimized Kernel

```
1 void matrix_multiply(float a[N][N], float b[N][N], float c[N][N]) {
2   int i, j, k, p;
3   k_loop: for(k = 0; k < N; k++) {
4     i_loop: for(i = 0; i < N; i++) {
5     //i_loop PIPELINE II = II_i
6       p_loop: for(p = 0; p < N; p += N/II_i) {
7       #pramga HLS PIPELINE II = 1
8         j_loop: for(j = 0; j < N/II_i; j++) {
9         #pragma HLS UNROLL
10        c[i][p+j] += a[i][k] * b[k][p+j];
11 } } } } }
```

Figure 2.4: Optimized pseudo code of square matrix multiplication.

To fix the inefficient register sharing problem by Vivado HLS tool, we optimize the HLS code by adding *p_loop* in *i_loop*. As shown in Figure 2.4, to pipeline the *i_loop*

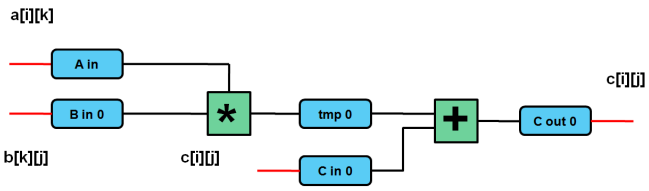


Figure 2.5: Architecture for optimized code, N = 6, II = N = 6

with a specific II (e.g., II_i = 1), we change the increment value of the p index in the *p_loop* and apply the HLS PIPELINE II = 1 pragma. In the *p_loop*, every cycle, $\frac{N}{II\_i}$ elements within one row of the c matrix, $c[i][p]$, $c[i][p+1]$, ... , $c[i][p+\frac{N}{II\_i}-1]$ are updated using $a[i][k]$ and $\frac{N}{II\_i}$ elements within one row of b matrix, $b[k][p]$, $b[k][p+1]$, ... , $b[i][p+\frac{N}{II\_i}-1]$.

The architecture generated by the optimized code will share the registers perfectly. For example, when N = 6, II = 6, the generated architecture is shown in Fig. 2.5. Compared to architecture generated by baseline code shown in Fig. 2.3, there are no extra MUXes, registers or interconnects. The energy model for different II in Section 2.3 and Section 2.4 are based on the architecture generated by the optimized code. Section 2.5 discusses the baseline model in more details.

After optimization, the resources and cycles to finish the matrix multiplication kernel can be generalized in terms of problem size N and PIPELINE II of the *i_loop*:

1. There are $\frac{N}{II}$ multiplier(s) and $\frac{N}{II}$ adder(s), and each computes II elements within one row. The number of B or C input/output registers and temporary registers between multipliers and adders is $\frac{N}{II}$.
2. There are $\frac{N}{II}$ independent memory bank(s) for the b matrix, each with II column(s). It is the same for the c matrix. Only one memory bank is needed for the a matrix

Table 2.1: HLS reported resource usage for multiplier and adder under different IIs, N=24

II	**1**	**2**	**3**	**4**	**6**	**8**	**12**	**24**
DSP	120	60	40	30	20	15	10	5
FF	8520	4260	2840	2130	1420	1065	710	355
LUT	8376	4188	2792	2094	1396	1047	698	349

since $a[i][k]$ is shared within one *i_loop* iteration.

3. The number of cycles to finish the kernel is $N^2 \times II$.

2.3 Energy Model

2.3.1 Computation Energy

The computation energy includes arithmetic energy (multiply-add operations) and register energy for holding the inputs, outputs, and temporaries. We can perfectly share these PEs and registers by factor of II, making the total PE and register energy consumption:

$$E_{compute} \propto \frac{N}{II} \times \left(N^2 \times II\right) = N^3 \tag{2.1}$$

For Xilinx 7 Series FPGAs, each multiply-add needs three DSP48E [Xilc] for the floating-point multiplier and two DSP48E for the adder along with a fixed number of LUTs and FFs. Table 2.1 shows the resource usage for multipliers and adders decreasing as $\frac{1}{II}$ since PEs are perfectly shared.

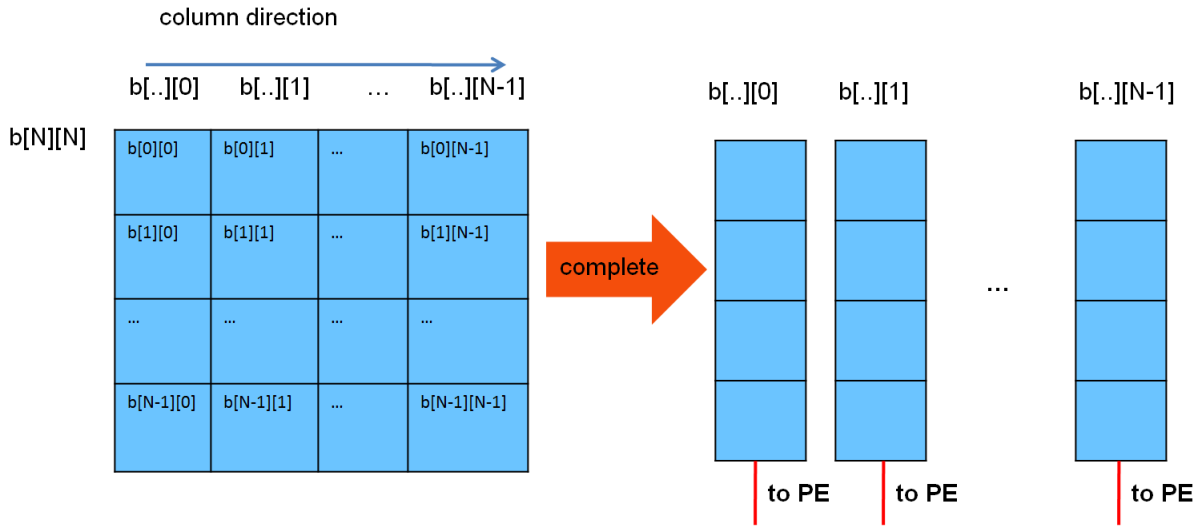


Figure 2.6: Complete partition of b matrix in column direction to enable fully pipelining, i.e., II = 1

2.3.2 Memory Energy

In order to fully pipeline the matrix multiplication, each PE needs to access each column of the b and c matrix simultaneously. HLS provides comprehensive partition pragmas [Xila] to easily partition an array into individual memory banks. For example, when II = 1, we use the complete partition pragma to partition b along the column direction as shown in Fig. 2.6. Each column, $b[..][N]$, becomes an individual memory block.

As II increases, the number of simultaneous accesses to the b matrix decreases, which means more columns can be placed in the same memory bank with size of $N \times II$. In this design, cyclic partition pragma is applied to the b and c matricies to automatically split the memory along the column direction in $\frac{N}{II}$ equally sized blocks interleaving the original array as shown in Fig. 2.7.

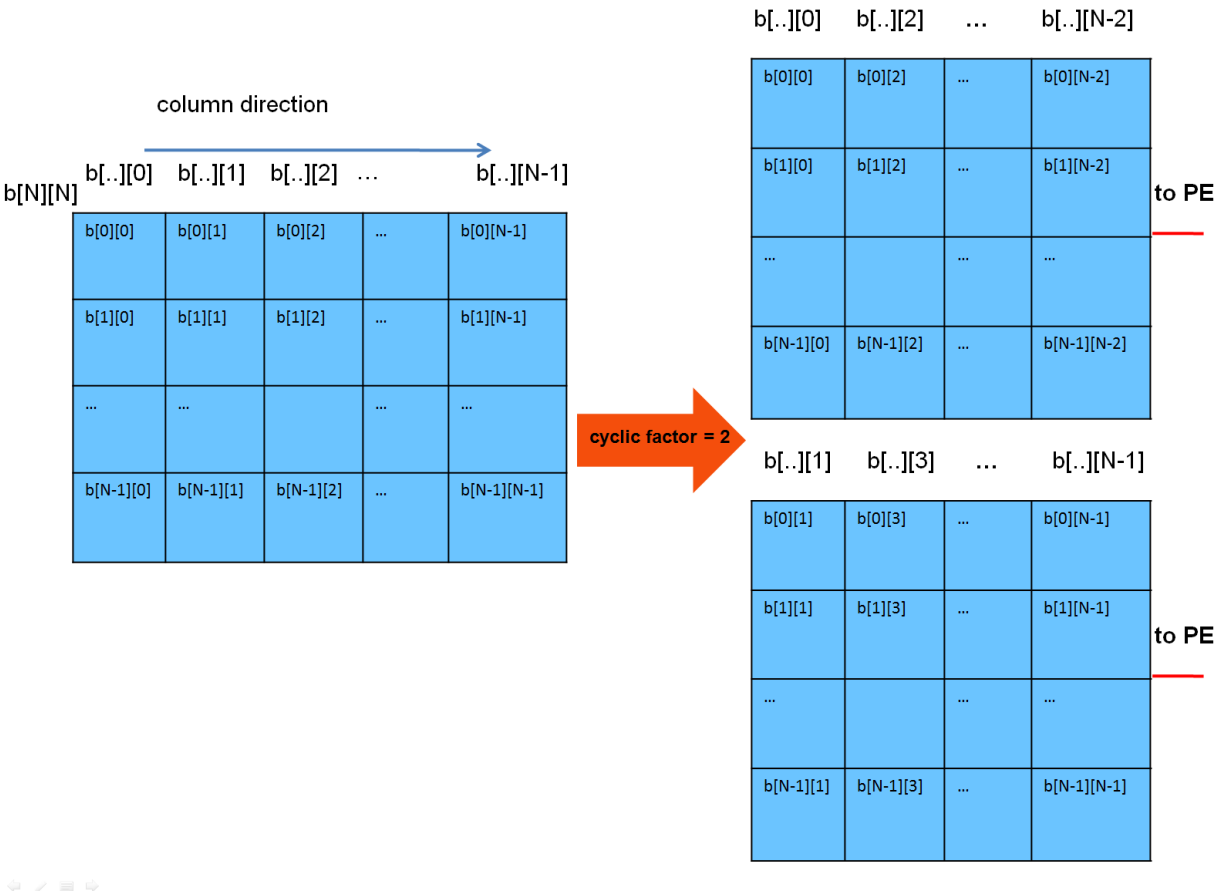


Figure 2.7: Cyclic partition of b matrix in column direction when cyclic factor is 2, i.e., $II = N/2$

In general, there are $\frac{N}{II}$ b or c banks; within each, II columns of data are stored. We need to consider the total memory energy when accessing these banks. In total, there are N^3 b memory reads, N^3 c memory reads and writes, and N^2 a memory reads, which we could safely ignore when N is large enough. Each memory access is from a logic memory bank with size of $N \times II$. On Xilinx 7 series FPGAs, all the logic memory banks are constructed using the embedded BRAM18K memory banks

on the chip [Xild].

If we activate a single BRAM for each read within a bank, the total energy reading from BRAMs is constant at:

$$E_{mem} \propto \frac{N}{II} \times \left(N^2 \times II\right) = N^3 \tag{2.2}$$

When the logical memory bank size is larger than physical BRAM18K bank size, it needs to be constructed using multiple BRAM18K banks, and the area of each logical memory bank increases. This impacts the wiring as we see in the next section.

2.3.3 Interconnect Wire Energy

The wire energy can be decomposed into wires within PEs and wires connecting PEs and memory. Wiring within the PE is fixed and will scale with the compute energy.

$$E_{wire.in.pe} \propto \frac{N}{II} \times \left(N^2 \times II\right) = N^3 \tag{2.3}$$

Wire transferring broadcast data. In this matrix-multiply algorithm, broadcast wires must transfer $a[i][k]$ from the memory bank storing the a matrix to the multipliers as shown in Fig. 2.8. The BRAM blocks storing the a matrix are close to the input register, Ain, and close to one multiplier. This broadcast should take energy proportional to the total area of all the PEs it is feeding.[1] As II increases, the total PE area scales as $\frac{N}{II}$, until we can no longer fit $N \times II$ elements of the b matrix into

[1] [Lei80] shows the H-tree layout has linear layout area, which implies linear wirelength in the area, which in turn implies linear energy.

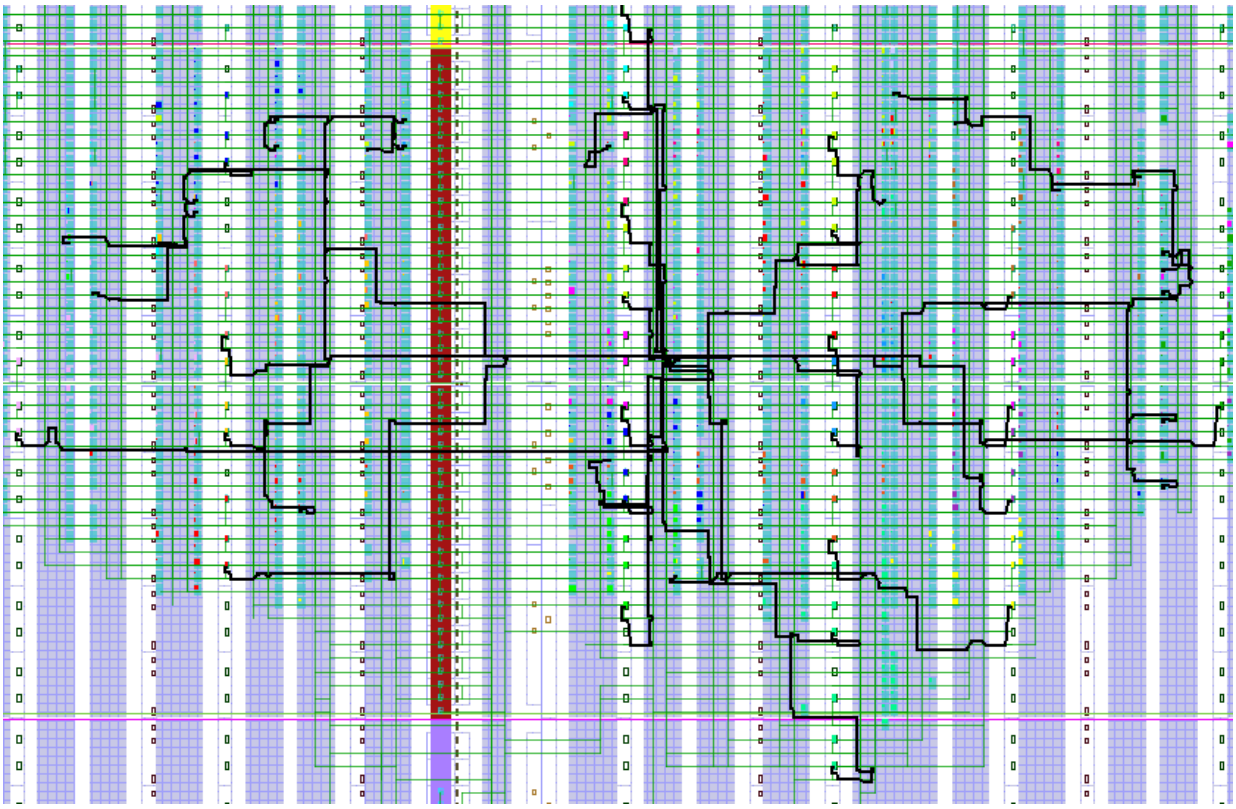


Figure 2.8: Routing of broadcasting a[i][k] to all 24 multipliers, N = 24, II = 1

a single BRAM. Thus the total energy for broadcasting $a[i][k]$ is:

$$E_{wire.share.A} \propto \frac{N}{II} \times N^2 = \frac{N^3}{II} \tag{2.4}$$

After $N \times II > BRAM18K$, this scaling changes in interesting ways. At this point, the total layout area for all the PEs becomes dictated by BRAMs not DSPs, and the area does not change with II. If we must broadcast to all the BRAMs, this means the broadcast energy does not shrink with II.

$$E_{wire.share.A} \propto N^2 \times N^2 = N^4 \tag{2.5}$$

However, we really only need to broadcast to a few BRAMs within a PE, allowing the broadcast energy to continue to shrink with increasing II. Current synthesis tools do not exploit this opportunity.

Wire transferring private data. When the logical memory bank size $N \times II$ is smaller than size of the physical BRAM18K bank (18432 bits, 576 floating-point numbers), the wiring between the private c and b matrix memory banks and the PE logic is constant, so the total energy for wiring also scales proportionally, independent of II:

$$E_{wire.priv.B,C} \propto \frac{N}{II} \times \left(N^2 \times II\right) = N^3 \tag{2.6}$$

When the logical memory bank size is larger than physical BRAM18K bank size, the wiring between the private b and c memory banks and the PE logic also grows as the square root of the memory capacity or $\sqrt{N \times II}$. Thus, the total energy routing memory is

$$\begin{aligned} E_{wire.priv.B,C} &\propto \frac{N}{II} \times \left(N^2 \times II\right) \times \sqrt{N \times II} \\ &\propto N^{3.5} II^{0.5} \end{aligned} \tag{2.7}$$

2.3.4 Leakage

During the computation, the FPGA will also leak energy proportional to the time for the computation and the resources that leak during the computation. If we put nothing else on the FPGA and use a fixed size FPGA that does not offer any power gating for unused components, leakage increases with runtime and hence II

(P_{FPGA_leak} is the leakage power for the FPGA chip):

$$E_{leak} \propto N^2 \times II \times P_{FPGA\_leak} \tag{2.8}$$

However, if we use a design with perfect power gating of unused components, the leakage should scale with the utilized logic. For the case where $N \times II < BRAM18K$:

$$E_{leak} \propto \frac{N}{II} \times N^2 \times II = N^3 \tag{2.9}$$

If we exploit the smaller resources of the $II > 1$ designs to use a smaller FPGA, we can get some of the effects of Eq. 2.9. Similarly, if we exploit the smaller resource utilization of the $II > 1$ to put additional logic onto the FPGA that fills the resources unused by the matrix-multiply, the leakage attributable to the multiply should scale closer to Eq. 2.9.

2.3.5 Total Energy

Putting all the energy components together and assuming perfect power gating (Eq. 2.9), we have total energy as the follows:

$$\begin{aligned} E_{total} &= E_{compute} + E_{memory} + E_{wire} + E_{leak} \\ &= \begin{cases} N^3 \left(c1 + \frac{c2}{II}\right), \\ \text{if } N \times II \leq \text{BRAM18K} \\ N^3 \left(c3 + c4 \times N + c5 \times II^{0.5}\right), \\ \text{if } N \times II > \text{BRAM18K} \end{cases} \end{aligned} \tag{2.10}$$

2.4 Results

For small N, when the design is not memory dominated, we can expect to see decreasing energy until $N \times II = BRAM18K$ driven by broadcast wiring energy. Beyond that, we expect to see energy increase with II due to memory and wiring energy between BRAMs.

We mapped the HLS designs to an Virtix-7 XC7VX485T chip using Vivado 2015.1.5. We simulated each mapped design in Vivado with random a and b matricies. We then used the Switching Activity Interchange format (SAIF) file generated from post-implementation simulation to estimate the energy required by the mapped designs. From the mapped designs, we used linear regression fit to determine the constants $c1$–$c5$ in Eq. 2.10.

Fig. 2.9 shows how the energy components scale with II from the Vivado mapped designs along with the total energy model from our fit model. We can see the mostly flat DSP and logic energies that match the analytic description. We also see that the interconnect energy and the overall energy drop with increasing to $II = 16$ where $N \times II = BRAM18K$. We see the interconnect energy grow after that as expected. However, we also see that BRAM energy, rather than remaining flat, increases with II after $II = 16$. Here, Vivado mapped designs are unnecessarily activating all of the BRAMs, not just the BRAM that holds the data needed on each cycle. This makes the total design energy unreasonably high for large II. It should be possible to avoid activating the unused BRAMs as illustrated in [TBN07, KLD15]. Making the perfect power gating assumption, we see that energy is minimized at $II = 8$. However, the effect is small and the benefit over $II = 1$ is less than 3%. If we get less than perfect power gating, this effect will easily be dominated by an increase in leakage energy

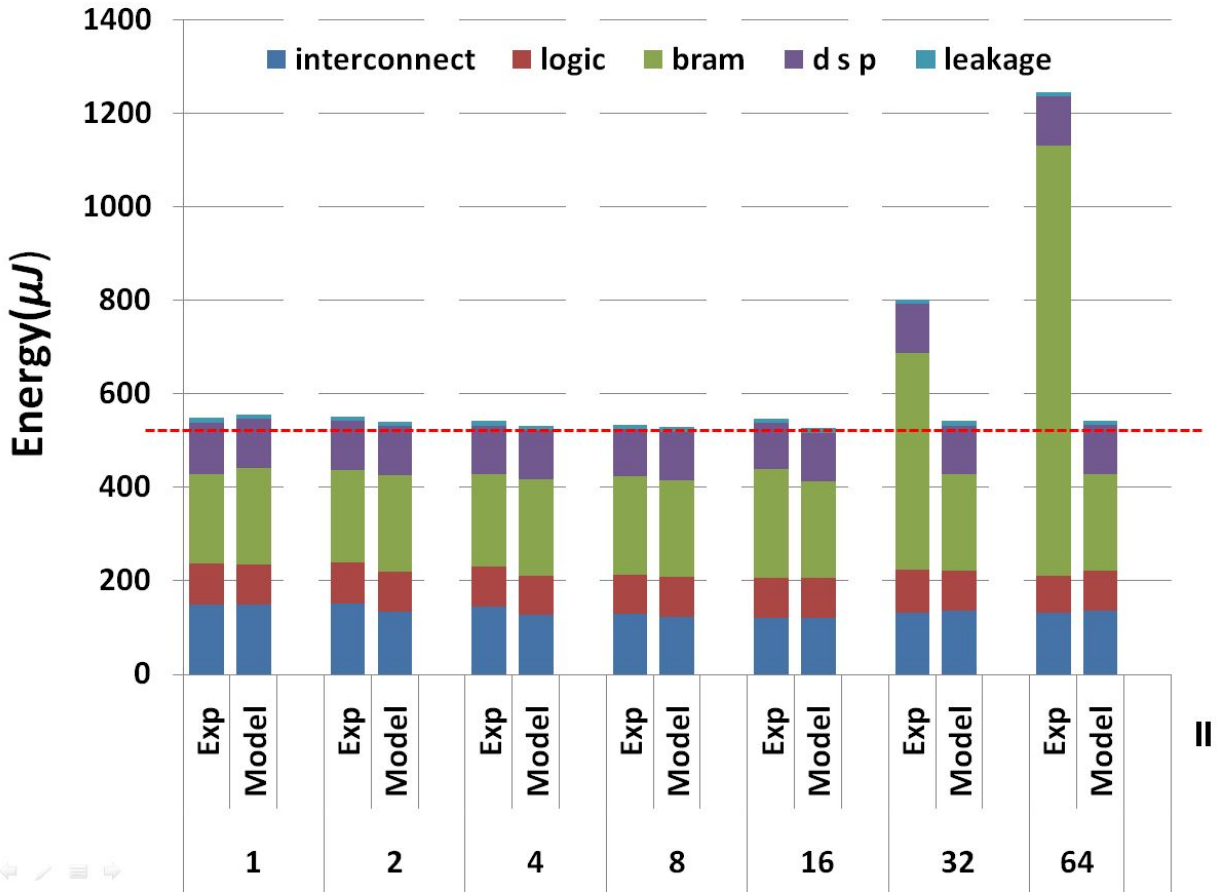


Figure 2.9: Energy Scaling with II for $N = 64$ Matrix Multiply

with II.

Fig. 2.10 shows how energy scales with N, including how this effects the optimal II and our model fit. The II for the minimum energy point decreases with N since larger N means the single BRAM capacity is reached at a lower II. Since all energy components scale as N^3 for the region where $N \times II \leq BRAM18K$, the energy proportions remain the same as N grows.

Note that all the PEs are generating the same addresses for their local b and c

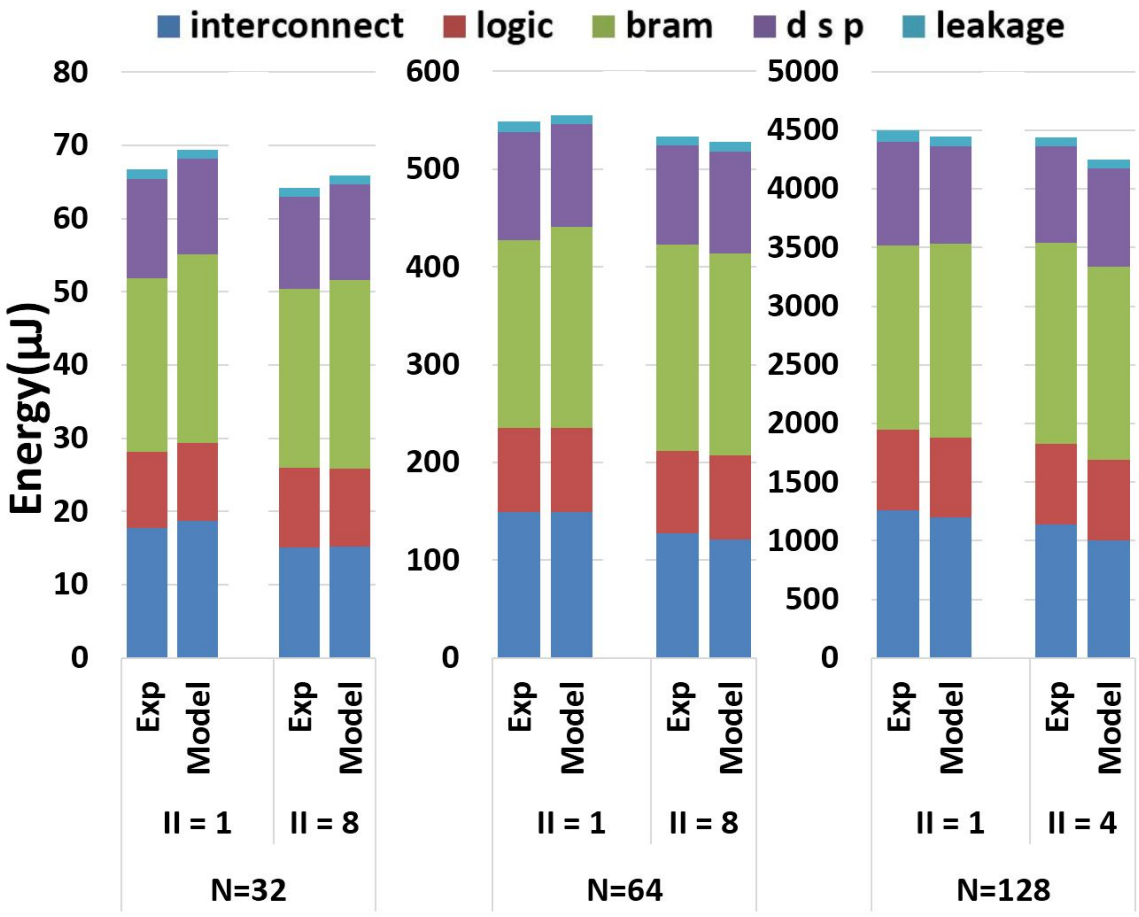


Figure 2.10: Scaling with N for Matrix Multiply

memories. Consequently, the design can use a single address generator for all the BRAMs. However, for some values of N and II, Vivado HLS will not share the address generators, resulting in much larger logic energy at those design points. Also, as we pointed out earlier, without adding the p loop in Listing 1, the HLS tool fails to share the registers properly.

2.5 Discussion for Baseline Kernel

2.5.1 Model of Energy Overhead

Before optimizing the code, there are inefficiency in sharing registers for baseline kernel when $II > 1$. Energy overhead includes energy on MUXes and 1-to-II fan-out and II-to-1 fan-in interconnect.

2.5.1.1 Energy Overhead on MUX

When pipeline II is not 1, each PE is shared and there are input MUXes to select the correct input from different input registers in each cycle as shown in Fig 2.3. The number of II-to-1 MUXes depends on the number of PEs $\frac{N}{II}$, and each MUX switches every cycle in $N^2 \times II$ cycles. When II is small, it is cheap to use LUT resource to implement the II-to-1 MUX. The power of II-to-1 MUX implemented using LUT is proportional to the number of input. Thus the energy spent on MUXes is

$$E_{mux} \propto \left(\frac{N}{II} \times N^2 \times II\right) \times II = N^3 \times II \tag{2.11}$$

As II increases, energy on MUXes increases, which leads to an increase in computation (logic part) energy.

2.5.1.2 Energy Overhead on Interconnect

When II increases, II input registers needs to be routed to the same PE through II-to-1 MUX and II output registers needs to be routed through the same c memory port. For example, as shown in Fig. 2.3, when II = N, there is only one b memory

port and it needs to be routed to different input registers *Bin0* to *Bin5* through 1-to-II fan-out wires. And the multiplier needs to select from one input registers through II-to-1 MUX. Similarly, there are fan-in and fan-out wires associated with the adder. As II increases, the number of registers (area) associated with one PE increases as II. The average length of each wire[2] scales as $\sqrt{II}$ and there are N wires as number of *Bin*, *tmp*,*Cin* and *Cout* registers remain the same. In every II cycles, only one of II nets are valid and are switching. So each wire switches N^2 times in total $N^2 \times II$ cycles. Thus, the total energy for wiring $b[k][j]$, intermediate results and $c[i][j]$ through 1-to-II fan-out and II-to-1 fan-in interconnect is

$$E_{wire} \propto \left(N \times \sqrt{II}\right) \times N^2 = N^3 \times \sqrt{II} \tag{2.12}$$

2.5.2 Results

Fig. 2.11 shows the energy components scale with II from both measurement and model that takes overhead into account for baseline kernel. We see the logic energy constantly increases as II increases, which matches with the overhead from MUXes. The interconnect energy also increases, which imply that the energy overhead in fan-in and fan-out interconnect diminish the energy saving in transferring broadcast data. In this way, $II = 1$ will always achieve the minimum energy consumption.

The optimization process showed in baseline kernel and optimized kernel is important in designing customized accelerator in terms of energy saving. Understanding the underlying architecture generated by HLS tool will help us understand the source of inefficiency and improve the design by code rewriting.

[2]we assume the average wire length scales with $\sqrt{area}$

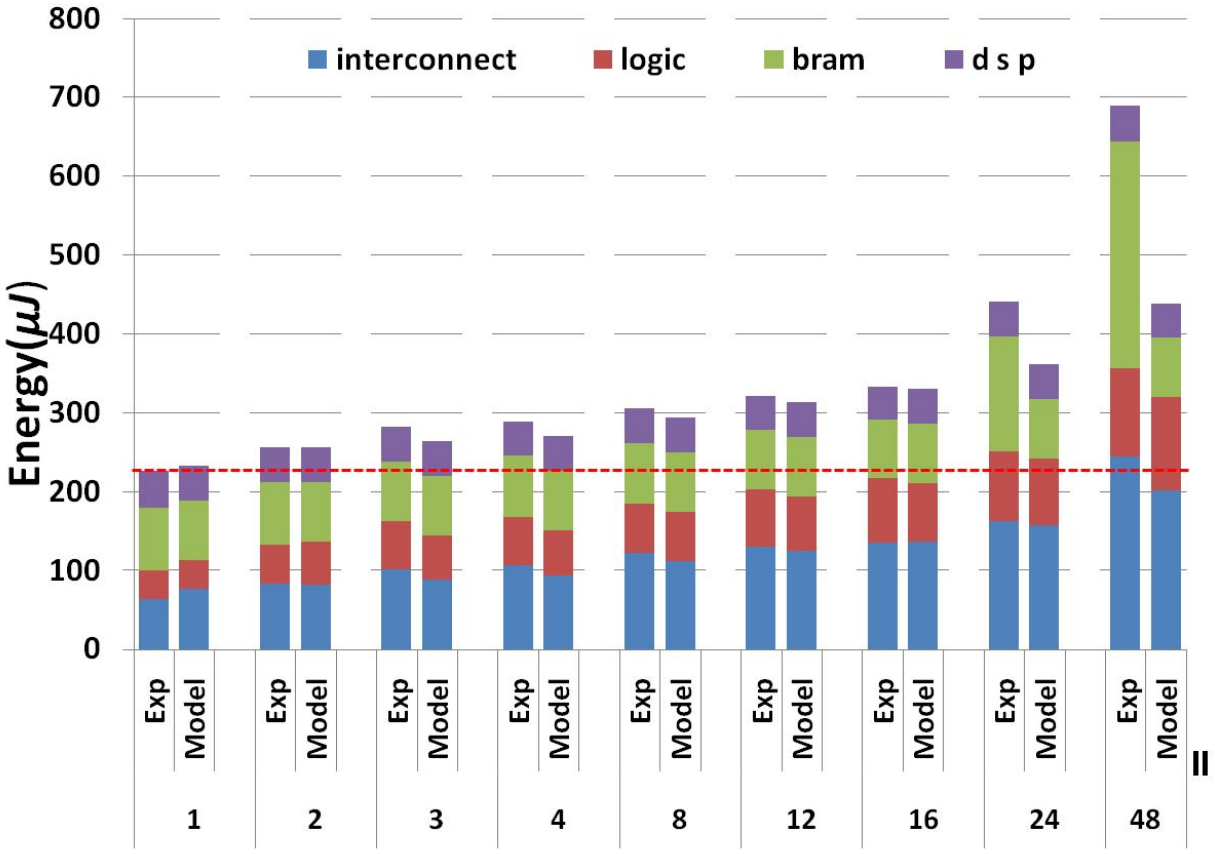


Figure 2.11: Energy Scaling with II for $N = 48$ Matrix Multiply Baseline Kernel

2.6 Conclusion and Future Work

Interconnect energy within our matrix-multiply kernel is minimized for an II that is typically greater than one. With efficient power gating or alternate use of chip resources, this can lead to minimum total energy at a point other than the fully pipelined, $II = 1$ point. Nonetheless, the effect is small and the fully pipelined design often uses the least energy in practice, both due to leakage, inefficient coding style and other discrete and non-ideal scaling effects.

The energy modeling framework illustrated here should be adaptable to other kernels. However, there is good reason to believe that kernels will differ in how they scale in key areas. We expect interconnect energy to scale differently for other tasks or even implementations of the same task. For example, using the systolic-array implementation of matrix multiply [JCP05], one may see different scaling. Our matrix-multiply kernel had near perfect sharing of logic as II increased, which will not be the case for less regular tasks. Consequently, it will be useful to characterize how these components scale for other tasks and develop a suitably parameterized energy model that can be adapted to various tasks characteristics. Ultimately, we hope model generation can be automated and provide high-level guidance for designers. As illustrated here, these models may also help to identify inefficiencies in current mapping tools that should be addressed to achieve energy efficient designs.

CHAPTER 3

Latte: Locality Aware Transformation for High-Level Synthesis

In this chapter we classify the timing degradation problems using four common collective communication and computation patterns in chip-level HLS-based accelerator design: scatter, gather, broadcast and reduce. These widely used patterns scale poorly in one-to-all or all-to-one data movements between off-chip communication interface and on-chip storage, or inside the computation logic. Therefore, we propose the Latte microarchitecture featuring pipelined transfer controllers (PTC) along data paths in these patterns. Furthermore, we implement an automated framework to apply our Latte implementation in HLS with minimal user efforts. Our experiments show that Latte-optimized designs greatly improve the timing of baseline HLS designs by 1.50x with only 3.2% LUT overhead on average, and 2.66x with 2.7% overhead at maximum.

3.1 Motivation and Challenges

In this section we use a common practice accelerator design template shown in Fig. 3.1 to illustrate the low operating frequency in scaled-out designs generated by HLS tools. The `app` defines an accelerator that has input buffer `local_in` and output buffer

`local_out`. In each iteration, it reads in `BUF_IN_SIZE` data (line 13) from off-chip to on-chip buffers, processes in `NumPE` kernels (line 14, 26), and then writes to off-chip from on-chip buffers (line 15). Here, double buffer optimization (A/B buffers) is applied to overlap off-chip communication and computation. Loop unroll (lines 23-26) and local buffer partitioning (lines 6-9) are applied to enable PE parallel processing.

In the remainder of the section, we summarize the design patterns from the corresponding microarchitecture in Fig. 3.2(a) and analyze the root cause of the critical path.

One-to-all scatter. In Fig. 3.2(a), `buffer_load` function is executed to read in data from DRAM using AXI protocol. As shown in Fig. 3.3, a common way to do this in HLS is either using `memcpy` (line 2) or in a fully pipelined loop (lines 3-6) to enable burst read. We observe that when we increase `NumPE`, the HLS report gives a constant estimated clock period for `buffer_load`, which is not the case in real layout. First, `local_in` is partitioned for parallel PE processing. Each partitioned bank (BRAM or FF) is routed to close to the corresponding PE logic, which results in scattered distribution of local buffers. We show the layout of a scatter pattern in a real application in Fig. 3.4(a). The yellow area highlights on-chip input buffers which span the whole chip. The white arrows show the wires connecting AXI read data port (with high fan-out) to buffers. Since HLS optimistically estimates the function delay without considering wire delay and schedule data from the AXI read port to one BRAM bank every clock cycle, the highlight wire is supposed to switch every clock cycle, and this is one cause of the critical path.

All-to-one gather. Fig. 3.2(a) shows the `buffer_store` module connecting partitioned buffer banks and the AXI write port. In order to select the data from a

```
#define BUF_IN_PER_PE BUF_IN_SIZE/NumPE
#define BUF_OUT_PER_PE BUF_OUT_SIZE/NumPE
void app(int data_size,
int *global_in, int *global_out) {
  // local buffer
  int local_in_A[NumPE][BUF_IN_PER_PE];
  int local_in_B[NumPE][BUF_IN_PER_PE];
  int local_out_A[NumPE][BUF_OUT_PER_PE],
  int local_out_B[NumPE][BUF_OUT_PER_PE];
  for (int i = 0; i < data_size/BUF_IN_SIZE+1; i++) {
    // double buffer
    if (i % 2 == 0) {
      buffer_load(local_in_A, global_in+i*BUF_IN_SIZE);
      buffer_compute(local_in_B, local_out_B);
      buffer_store(global_out+i*BUF_OUT_SIZE, local_out_A);
    }
    else {
      buffer_load(local_in_B, global_in+i*BUF_IN_SIZE);
      buffer_compute(local_in_A, local_out_A);
      buffer_store(global_out+i*BUF_OUT_SIZE, local_out_B);
} } }
void buffer_compute(int** local_in, int** local_out) {
  for (int i=0; i<NumPE; i++) {
  #pragma HLS unroll
    // kernel replication
    PE_kernel(local_in[i], local_out[i]);}
}
```

Figure 3.1: HLS accelerator design template.

particular bank in one cycle, A NumPE-to-1 multiplexer (MUX) is generated. We highlight `buffer_store` in violet and MUX logic in yellow in an accelerator layout shown in Fig. 3.4(b). Similarly, long wires from partitioned storage banks to the AXI port through distributed MUX are the cause of long interconnect delay.

One-to-all broadcast. As distinct PEs span a large area, they incur long wires

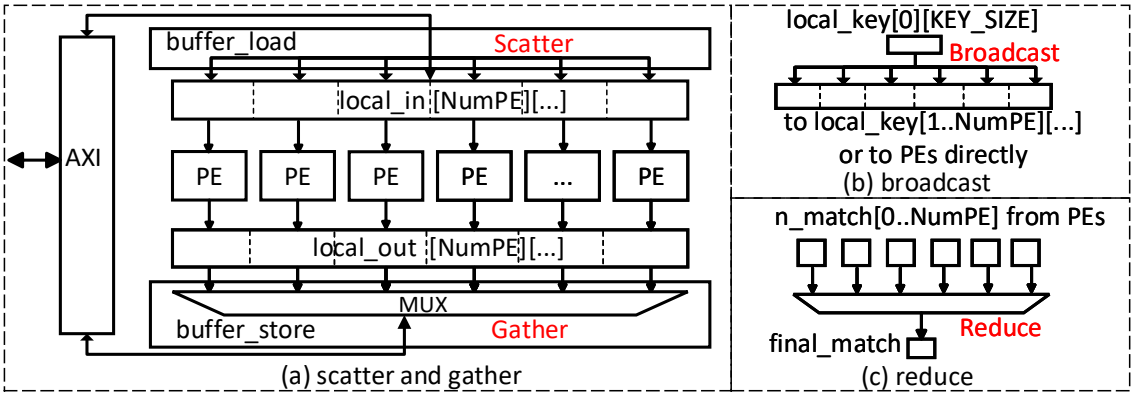


Figure 3.2: Accelerator microarchitecture.

to broadcast data to computation logic directly (`bc_in_compute`), e.g., matrix A is broadcast to multiplier-accumulators in matrix-multiplier [Zho16]) or to local copies of shared data within each PE (`bc_by_copy`), e.g., Advanced Encryption Standard (AES) [DR13] broadcasts a shared key to all processing elements that perform encryption tasks independently.

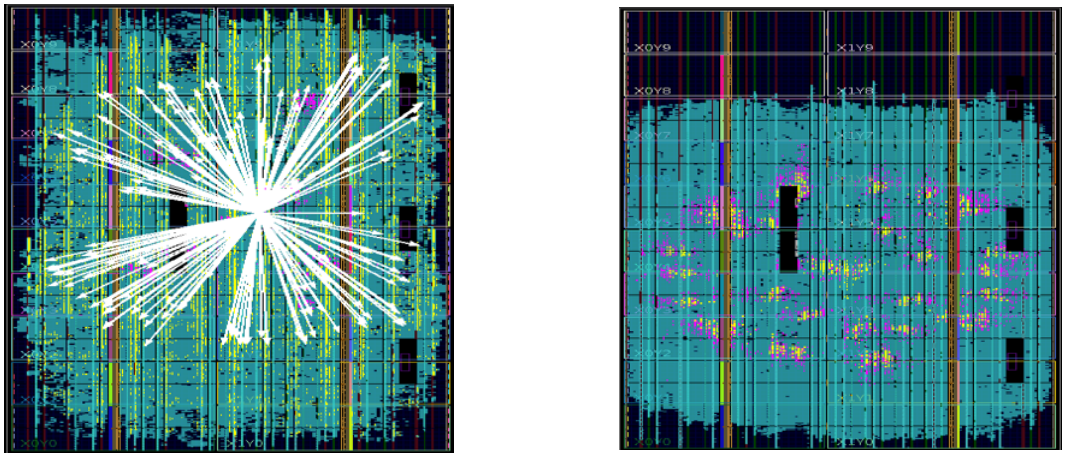
All-to-one reduce. Reduce is a common operation that returns a single value by combining an array of input. One example is the string matching application KMP to count the number of a certain word found in a string, where different string matching engines need to accumulate their results to get the final count.

We show architecture of broadcast and reduce in Fig. 3.2(b)(c) and baseline code in Fig. 3.5. Layout of broadcast wires are similar to those in scatter and reduce as in gather patterns.

The four patterns are common and appear in most accelerator designs. As shown in Table 3.1, we have implemented several accelerators from a variety of domains of applications and reported location of the critical path in the baseline designs. Except

```
1 void buffer_load(int local_in[NumPE][], int* global_in) {
2   // memcpy(local_in, global_in, BUF_IN_SIZE);// burst read
3   for(int i = 0; i < NumPE; i++)
4     for(int j = 0; j < BUF_IN_PER_PE; j++) { // for each PE
5     #pragma HLS pipeline II = 1
6     local_in[i][j] = global_in[i*BUF_IN_PER_PE + j];}
7 }
8 void buffer_store(int* global_out, int local_out[NumPE][]) {
9   memcpy(global_out, local_out, BUF_OUT_SIZE);
10  // for loop (similar to buffer_load, not shown)
11 }
```

Figure 3.3: HLS baseline buffer load and store.



(a) Scatter pattern

(b) Gather pattern

Figure 3.4: Layout of accelerator architecture.

NW and VITERBI, where critical paths lie in the computation PEs, all the other designs have critical paths that result from the four patterns.

```
// broadcast_in_compute omit here due to space limit
// broadcast_by_copy defined
void bc_by_copy(int local_key[NumPE][], int* global_key) {
  memcpy(local_key[0], global_key, KEY_SIZE);// to 1st copy
  for(int j = 0; j < KEY_SIZE; j++){
  #pragma HLS pipeline II = 1
    for(int i = 1; i < NumPE; i++){
    #pragma HLS unroll
      // 1st copy to the rest
      local_key[i][j] = local_key[0][j];}}
}
// each element in int* n_match is from a PE
void reduce(int &final_match, int n_match[NumPE]) {
  for(int i = 0; i < NumPE; i++){
  #pragma HLS pipeline II = 1
    final_match += n_match[i];}
}// other reduction operations are similar
```

Figure 3.5: HLS baseline broadcast and reduce.

Table 3.1: Benchmarks and *Achilles's heel* patterns in baseline designs.

Benchmark	Domain	Scatter	Gather	Broadcast	Reduce
AES	Encryption	✓	✓	✓	
FFT	Signal	✓	✓		
GEMM	Algebra	✓	✓	✓	
KMP	String	✓		✓	✓
NW	Bioinfo.	✓	✓		
SPMV	Algebra	✓	✓	✓	
STENCIL	Image	✓	✓	✓	
VITERBI	DP	✓	✓		

Checkmark ✓ represents the design has the pattern.
A star represents that a critical path lies in the pattern.
For broadcast, GEMM uses bc_in_compute while others use bc_by_copy.

3.2 Latte Microarchitecture

In order to reduce the wire delay in the critical paths in the patterns while keeping the computation throughput, i.e., not changing `NumPE`, we introduce the pipelined transfer controller (PTC), the main component of the Latte microarchitecture in the data path.

3.2.1 Pipelined Transfer Controller (PTC)

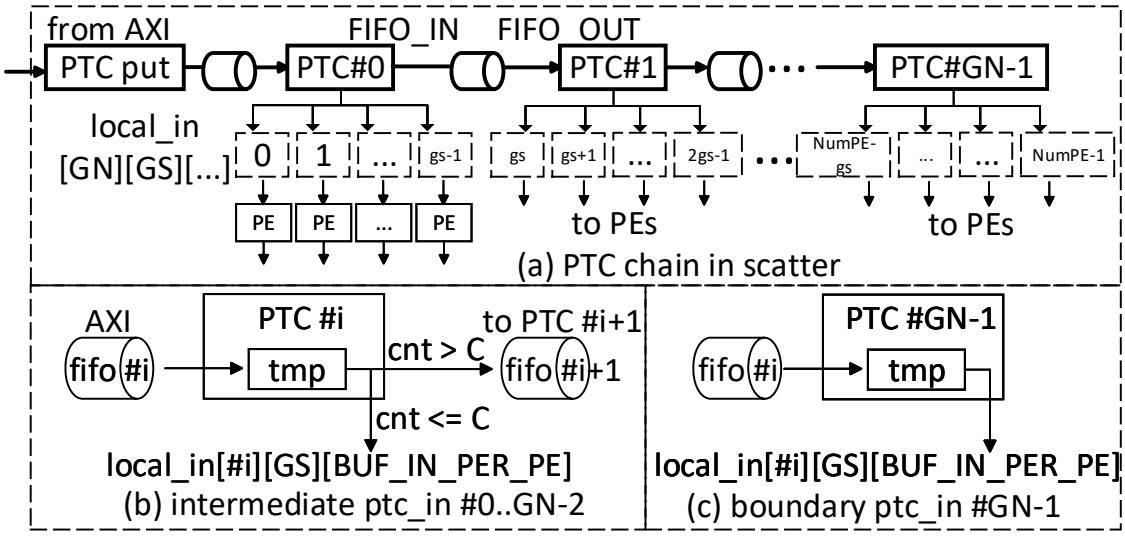


Figure 3.6: Microarchitecture of PTC in scatter.

Fig. 3.6(a) shows the microarchitecture of PTC chains in a scatter pattern. PTCs are chained in a linear fashion through FIFOs, and each PTC connects to a local set of buffers in PEs to constrain the wire delay. We denote the local set size as group size `GS` and number of sets as group number `GN`. The corresponding HLS implementation is also presented in Fig 3.7. To access `local_in` from different sets in

parallel, we first redefine it as `local_in[GN][GS][BUF_IN_PER_SIZE]` (line 2). PTCs are chained using FIFOs (`hls::stream`), and a dataflow pipeline (line 5) is applied to enable function pipeline in the PTC modules defined below (lines 7-10). There are three types of PTCs: `ptc_put`, intermediate and boundary `ptc_in`. In `ptc_put` (lines 13-17), it reads in data from AXI in a fully pipelined loop and writes to the first FIFO. Intermediate `ptc_in` reads in data from the previous PTC through FIFO. It first writes to local set of PE buffers and then writes the rest to the next FIFO (lines 18-33), as shown in Fig 3.6(b). Similarly, Fig 3.6(c) shows boundary `ptc_in`, where it reads data from the last FIFO and writes all the data to a local set of PE buffers.

In addition, we show the microarchitecture of the PTC chain in gather pattern in Fig. 3.8(a). Similarly, there are three types of PTC: boundary `ptc_out` (Fig. 3.8(b)), intermediate `ptc_out` (Fig. 3.8(c)) and `ptc_get`. The modules are similar to those in scatter with a difference in the opposite data transfer direction.

The microarchitectures of PTC broadcast and reduce patterns are similar to those for scatter and gather, which we leave out due to the space limitation. PTC is somewhat similar to the idea of multi-cycle communication in the MCAS HLS system [CFH04].

We finally show the microarchitecture of PTC in broadcast and reduce pattern in Fig. 3.9(a), and (b), respectively. PTC in broadcast is similar to that of scatter. On the other hand, PTC in broadcast forwards data that read from the previous FIFO to both FIFO in the next and also local set of buffers (`bc_by_copy`) or PEs (`bc_in_compute`) directly. PTC in reduce is similar to that of gather. The difference is that PTC in reduce has a reduction operation within each PTC to do local reduction of result from the previous FIFO and local set of buffers.

```
#include <hls_stream.h>
int local_in[GN][GS][BUF_IN_PER_PE]; // redef.
void PTC_load(
  int local_in[GN][GS][], int* global_in) {
#pragma HLS dataflow
  hls::stream<int> fifo[GN];// FIFOs, in Fig. 7a
  ptc_put(global_in, fifo[0]);
  for(int i = 0; i < GN-2; i++){
    ptc_in(fifo[i], fifo[i+1], local_in[i], GN-1-i);}
  ptc_in(fifo[GN-1], local_in[GN-1]);
}
void ptc_put(int* global_in, stream<int> &fifo){
  for (int i=0; i<NumPE; i++)
    for(int j = 0; j < BUF_IN_PER_PE; j++){
#pragma HLS pipeline
      fifo << global_buf[i*BUF_IN_PER_PE+j];}
}
void ptc_in( // #0..GN-2 ptc_in, in Fig. 7b
stream<int>&fifo_in, stream<int> &fifo_out,
int local_set[GS][BUF_IN_PER_PE], int todo){
  int i, j, k; int tmp;
  for(i= 0; i < GS; i++){ // to local first
    for(j = 0; j < BUF_IN_PER_PE; j++){
      tmp = fifo_in.read();
      local_set[i][j] = tmp;
  } }
  for(k=0; k < todo; k++) // to next ptc
  for(i= 0; i < GS; i++){
    for(j = 0; j < BUF_IN_PER_PE; j++){
      tmp = fifo_in.read();
      fifo_out.write(tmp);
  } }
}
```

Figure 3.7: Code snippet of HLS implementation for PTC in scatter.

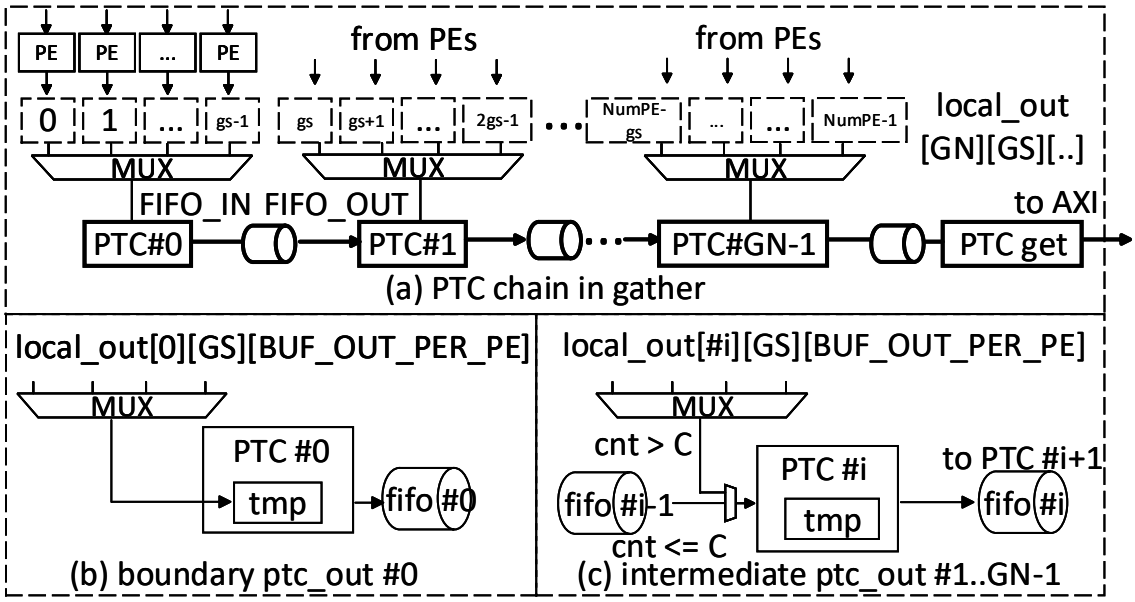


Figure 3.8: Microarchitecture of PTC in gather.

It is possible to manually implement the Latte microarchitecture in HLS. However, the implementation expands over 260 lines of code (LOC), which is 10× more than the baseline code shown in Fig. 3.3 and Fig. 3.5. To relieve the burden of manual programming effort in implementing Latte, we provide an automation framework that reduces the 260-LOC implementation to simply a few directives.

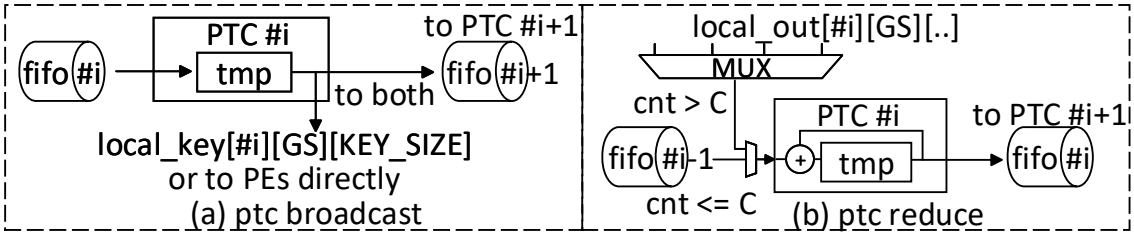


Figure 3.9: Microarchitecture of PTC in broadcast and reduce.

3.2.2 Automation Framework

We implement a semiautomatic framework to make use of Latte by having a user-written HLS kernel with simple Latte pragmas. The Latte pragma indicates the on-chip buffer with the pattern to be optimized. For example, Fig. 3.10 presents an example of using a Latte pragma to enable scatter pattern with PTCs for the on-chip buffer from Fig. 3.1.

```
1 #pragma latte scatter var="local_in_B"
2 int local_in_B[NumPE][BUF_IN_PER_PE];
```

Figure 3.10: An example of Latte pragma.

After parsing the kernel code with pragmas, we perform code analysis by leveraging the ROSE compiler infrastructure [rosrg] to identify the kernel structure, array types and sizes. Subsequently, we apply predefined HLS function templates of Latte by performing source-to-source code transformation. Corresponding optimization, such as memory partitioning, memory coalesce [CWY17], and so forth, are applied as well. We implement a distributed runtime system that launches multiple Amazon EC2 [amac2] instances for exploring the PTC group size with Vivado HLS [Xila] in parallel to determine the best design configuration. Note that since we only search the group size that is a divisor of the PE number, the design space is small enough to be fully explored.

3.3 Experimental Evaluation

We use Alpha Data ADM-PCIE-7V3 [Alpdf] as an evaluation FPGA board (Virtex-7 XC7VX690T) and Xilinx Vivado HLS, SDAccel 2017.2 [Xila] for synthesis. For each

benchmark listed in Table 3.1, we implement the baseline design and scale out N times until fully utilizing the on-chip resource or failing to route. We then obtain the baseline frequency as F and baseline area as A. Then for N of an application, we choose GS as divisors of N. For each GS, Latte optimizations are applied on all existing patterns, and frequency is reported as F_{GS}, area as A_{GS}. Thus, the performance ratio of the Latte optimized design and baseline is expressed as F_{GS}/F, performance-to-area (P2A) ratio as $\frac{F_{GS}/A_{GS}}{F/A} = \frac{F_{GS}}{F}/\frac{A_{GS}}{A}$ (in terms of latency, each PTC introduces one extra cycle, which is negligible compared to the cycle number of the original design). Latte enables design space exploration for both ratios as shown in Fig. 3.11 for `GEMM`. As can be seen, `GEMM` achieves the optimal performance when GS is four, which has 227 MHz operating frequency with 35% LUT overhead. In addition, P2A optimal design is identified when GS is 16, achieving 207 MHz with only 8% area overhead. On the other hand, we can observe the performance degradation when GS decreases from four to one. The reason is that the critical path has been moved from data transfer to PEs when GS is four, and further reducing the data transfer wire delay will not improve the performance. This illustrates the motivation for selecting a suitable GS instead of always setting GS to one. Smaller GS means more PTS, and it only introduces more area overhead. Thus, neighbor-to-neighbor PTCs design is not necessarily the Perf. optimal nor Perf./Area optimal.

In addition, we report the resource utilization and operating frequency for baseline designs under N PEs (`ori.`) and the corresponding Latte designs with optimal P2A GS (`latte`) in Table 3.2. The Latte optimized design improves timing over baseline HLS design by 1.50× with 3.2% LUT, 5.1% FF overhead on average. For `FFT`, it even achieves 2.66× with only 2.7% LUT and 5.1% FF overhead. Even for designs such as `NW` and `VITERBI` where critical paths in baseline lie in PEs, the Latte optimized design

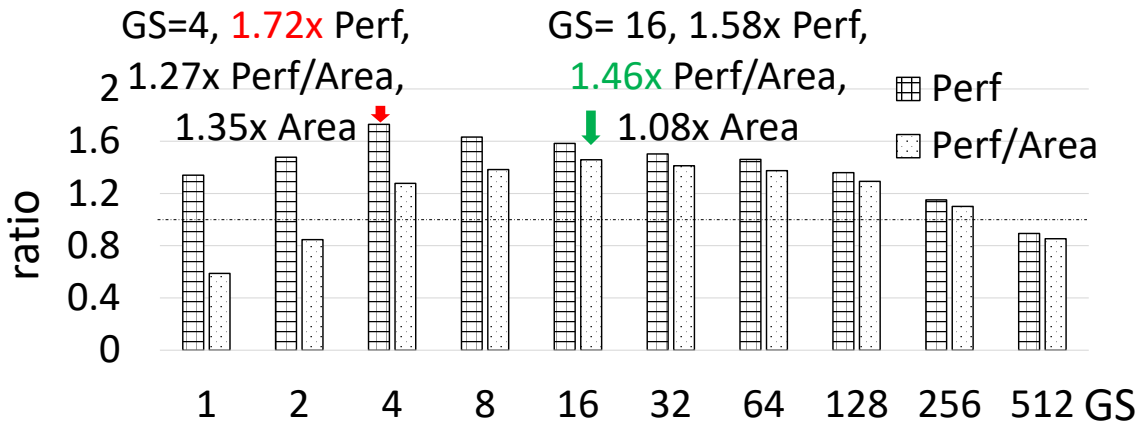


Figure 3.11: Performance and P2A ratio in GEMM with 512 PEs.

is still beneficial. A possible reason is that the Latte design helps the placement of PEs, which helps routing within PEs. In summary, the average frequency has been improved from 120 MHz to 181 MHz.

Finally, the overall frequency to area in Latte designs are shown in Fig. 3.12. It achieves 200 MHz on 61% chip area, and 174 MHz on 90%, which helps greatly in frequency degradation. We also show the layout of PTCs in gather pattern in FFT with 64 PEs and 16 PTCs in Fig 3.13, where PTCs are connected in linear fashion and scale much better.

3.4 Conclusion

In this chapter we summarize four common collective communication and computation patterns (i.e., scatter, gather, broadcast and reduce) in HLS that generate long interconnects in scaled-out design and result in degraded frequency. To achieve a high frequency, we propose the Latte microarchitecture which features pipeline

Table 3.2: Baseline design vs Latte optimized design.

Bench.	type	N / GS	LUT	FF	DSP	BRAM	Freq.
AES	ori.	320 /	50.4%	17.3%	0.1%	76.3%	127
	latte	/ 32	1.017	1.009	1	1	165, 1.30x
FFT	ori.	64 /	50.5%	23.2%	88.9%	78.5%	57
	latte	/ 4	1.027	1.056	1	1	152, 2.66x
GEMM	ori.	512 /	37.8%	29.6%	71.1%	69.7%	131
	latte	/ 16	1.044	0.962	1	1	207, 1.58x
KMP	ori.	96 /	5.0%	3.0%	0.2%	52.3%	126
	latte	/ 24	1.045	1.174	1	1	195, 1.54x
NW	ori.	160 /	65.1%	50.7%	0.0%	78.2%	174
	latte	/ 80	0.995	0.997	1	1	177, 1.02x
SPMV	ori.	48 /	19.1%	11.9%	18.9%	93.2%	160
	latte	/ 6	1.029	1.037	1	1	192, 1.20x
STENCIL	ori.	64 /	12.9%	10.9%	48.1%	87.1%	141
	latte	/ 16	1.094	1.139	1	1	188, 1.33x
VITERBI	ori.	192 /	72.0%	25.7%	10.8%	39.3%	155
	latte	/ 12	1.008	1.031	1	1	168, 1.08x
Average	ori.	/	NA	NA	NA	NA	120
	latte	/	1.032	1.051	1	1	181, 1.50x

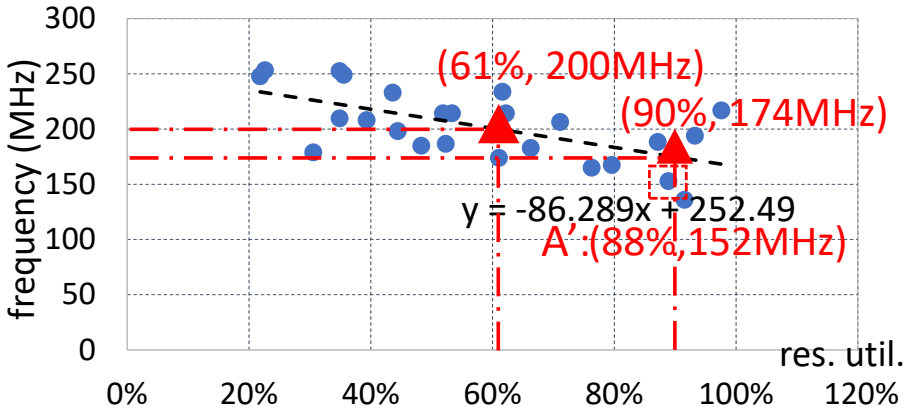


Figure 3.12: Freq. degradation much less severe in Latte optimized design.

transfer controllers in the four patterns to reduce wire delay. We also implement an automated framework to realize HLS-based Latte implementation with a only few

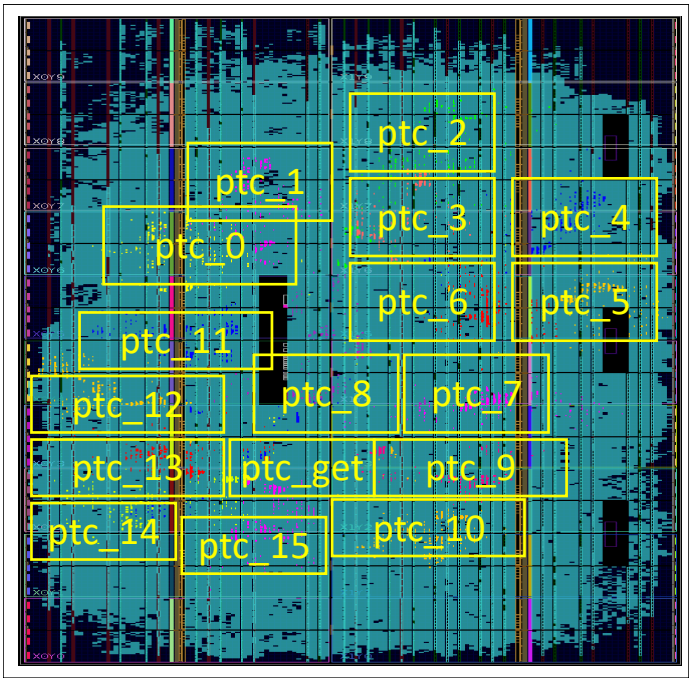


Figure 3.13: Layout for PTC chains in FFT, N=64, GS=4, GN=16.

lines of user-provided derivatives. Experiments show that the Latte optimized design improves frequency from 120 MHz to 181 MHz with 3.2% LUT, 5.1% FF overhead on average.

CHAPTER 4

Node-Level Performance and Cost Modeling: Computation Resources

In this chapter we first give a brief introduction to the driving applications discussed in subsequent chapters—that is, the whole genome sequencing pipeline explored in Section 4.1. Then, in Section 4.2 we analyze and reason the cost inefficiency of commonly adopted CPU-FPGA system integration considering two scenarios that either CPU or FPGA is the performance bottleneck. In Section 4.3 we combine the analysis result and generalize the performance and cost model to further conclude optimization opportunities that motivate the proposed framework in Chapter 6.

4.1 Whole Genome Sequencing Pipeline

In this chapter and the following chapters we use the whole genome sequencing pipeline as our driving application. Next-generation sequencing (NGS) technologies [BT13] have revolutionized genome research. Currently, according to the Broad Institute, an entire human genome with 30x coverage can be sequenced within a day, and the Broad Institute generates more than 43000 30x whole human genomes every year [Insa]. The advancement of high-throughput sequencing techniques establishes the need for faster subsequent genome alignment and analysis tools. We introduce

the main tools in GATK best practices [ACH13, Insb].

Alignment (BWA-MEM). BWA-MEM (Burrows-Wheeler Aligner) is the pre-processing step of the genome processing pipeline. It takes raw sequence data in FASTQ [CFG09] format, maps the genome data to the reference human genome and outputs analysis-ready binary alignment/map formats (BAM files [BM13]). In BWA-MEM, about 30%-50% of computation time is spent on a dynamic programming algorithm kernel, the Smith-Waterman (S-W) algorithm. The Smith-Waterman algorithm has quadratic time complex, and it is suitable for FPGA acceleration [Com18].

Performing base-quality score recalibration (GATK). Raw reads from DNA sequencers are error-prone, and each read has a possibility of misinterpreted nucleotide bases. For each base, a Phred quality score is reported, and it characterizes the confidence of the base accuracy [Ins17a]. Quality scores affect the downstream of analyses such as the genome-wide association study of cancer or precision medicine. The base quality score recalibration (BQSR) aims to detect and correct patterns of cosystematic biases by generating a model using the confidence scores reported from the DNA sequencer.

Performing data compression (Samtools). Samtools provides compression tools to convert the sequence alignment/map (SAM) to BAM files [BM13, LHW09]. In compression, the Deflate algorithm, which is the core of many lossless compression standards, is a good candidate for FPGA acceleration [QDF18].

Calling variants (GATK). After the data is properly processed as an analysis-ready BAM file, we use HaplotypeCaller (HTC) [Ins19] to find germline variants for pair-end sequence reads and Mutect2 [CLC13] for tumor sequence reads to identify the variation relative to the reference genome. Both applications include a high-

complexity algorithm called Pair Hidden Markov Models (PairHMM) [DEK98] that is suitable for the FPGA accelerator [Com18]. PairHMM has high time-complexity and heavy floating-point operations. In the current GATK implementation where Intel AVX intrinsics are used in the kernel, PairHMM typically dominates 39% and 89% of overall execution time in HTC and Mutect2 respectively .

4.2 Analysis for Straightforward CPU-FPGA Integration

Starting from Google's MapReduce [DG08b] in 2004, most modern widely used big data analytic systems such as Apache Hadoop [Whi12b] and Spark [ZCD12b] embrace dataflow and parallel pattern programming models. For example, programmers can use a parallel pattern *map* to specify that all tasks are completely independent and can be executed in parallel using multiple CPU cores. We first analyze the performance and cost for an application that consists of only *map* tasks on a single-node multiple-core CPU platform, where the execution timeline is shown in Figure 4.1a. For illustration purpose, we use the same input size and execution time of each task. We consider the following factors in our analysis:

- M is the total number of tasks.
- P is the total number of CPU cores in a single node.
- t is the time of each task on a CPU core.
- r is the proportion of kernel that can be offloaded to a FPGA accelerator.
- S is the FPGA accelerator speedup compared to a single-core CPU. It includes the CPU-to-FPGA communication overhead.

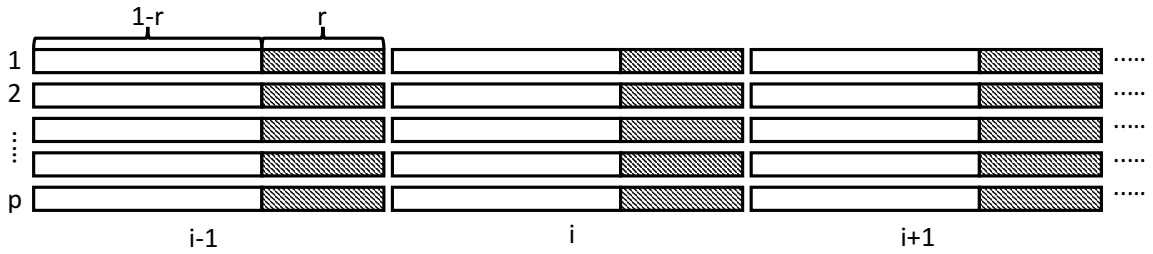


Figure 4.1a: CPU-Only System

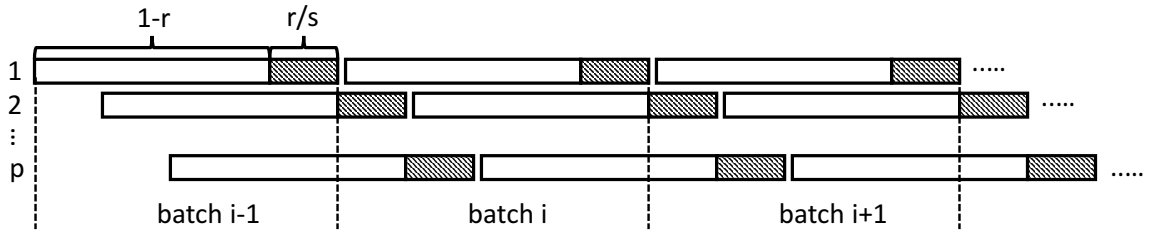


Figure 4.1b: CPU-FPGA System Case A, CPU is bottleneck

- c is the cost per unit time of a CPU core.
- CR is the cost ratio of FPGA compared to a single-core CPU. For example, on AWS `f1.2xlarge` instance, $CR = (\$1.65/\$0.4*8\text{-}8) = 25$.
- $\widetilde{P}$ is the Matching Core Number. When $P = \widetilde{P}$, CPU and FPGA have the same throughput.

As shown in Figure 4.1a, in each batch, P CPU cores are executed in parallel. In total, there are $\frac{M}{P}$ batches of tasks. The total runtime T_0 and cost C_0 is:

$$\begin{aligned} T_0 &= \frac{M}{P} \times t, \\ C_0 &= T_0 \times P \times c = M \times t \times c \end{aligned} \tag{4.1}$$

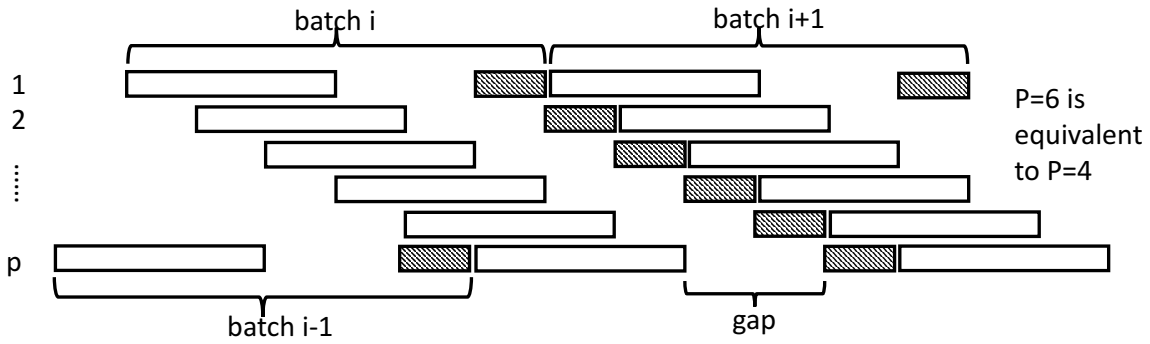


Figure 4.1c: CPU-FPGA System Case B, FPGA is bottleneck

Since the parallel programming model adopted by many modern big data analytic systems provides clear semantic information, it is an opportunity for system developers to offload as many time-consuming computational tasks as possible to FPGA accelerators to improve the performance. As a result, intuitive CPU-FPGA runtime systems [HWY16] usually ask all CPU workers to send all their tasks to the FPGA. Many of the systems include an accelerator task queue to deal with tasks requested from different workers. In this scenario, workers have to stay idle before their request can be fulfilled on the FPGA accelerator. In this subsection, we analyze the performance and cost of such systems.

Depending on r, S, and P, there are two cases when the computation throughput of CPU and FPGA are not balanced. Here we gradually increase P to illustrate these two cases.

Case A, $P<\widetilde{P}$: FPGA is underutilized. Even if we can ask multiple CPU cores to send tasks to an FPGA, the offloaded tasks still need to be fulfilled sequentially on the FPGA. As shown in Figure 4.1b, cores 1 and 2 to core P offload tasks on FPGA in a pipeline fashion. After core 1 finishes batch i, it starts the non-accelerated part

of batch *i+1* on a new data partition.[1] And when it is about to request accelerator in batch *i+1*, all the tasks in the previous batch have already finished execution on the FPGA. Therefore, its kernel acceleration request can be fulfilled without waiting. Thus, core 1 has no idling cycles, as well as the other cores. In this case, for any CPU core, it only needs to wait $\frac{r}{S} \times t$ when the FPGA is working on a kernel part and there are no other idling cycles. Runtime for each batch task is $t \times (1 - r + \frac{r}{S})$. And total runtime T_{1a} is :

$$T_{1a} = \frac{M}{P} \times t \times (1 - r + \frac{r}{S}), \tag{4.2}$$

For a platform with P CPU cores and one FPGA, total cost per unit time is $(P + CR) \times c$; total cost to run the application C_{1a} is:

$$\begin{aligned} C_{1a} &= T_{1a} \times (P + CR) \times c \\ &= M \times t \times c \times (1 - r + \frac{r}{S}) \times (1 + \frac{CR}{P}) \end{aligned} \tag{4.3}$$

For FPGA, the utilization is not 100% so there are idling cycles between offloaded tasks. Comparing C_{1a} in Equation 4.3 to C_0 in Equation 4.1, we can have *Cost Efficiency Index* $I = \frac{C_{1a}}{C_0} = (1 - r + \frac{r}{S})(1 + \frac{CR}{P})$.

When we gradually increase P to $\widetilde{P}$, the FPGA reaches 100% utilization. This happens when total runtime of offloaded tasks from $\widetilde{P}$ cores equals the runtime of a

[1]Here we plot the accelerated part at the end of the task, which simplifies the illustration. In real applications, the accelerated kernel can be in anywhere in a task.

single batch task time, that is, $\widetilde{P} \times \frac{r}{S} \times t = (1 - r + \frac{r}{S}) \times t$. We call $\widetilde{P}$ a *Matching Core Number*, and it can be calculated as $\widetilde{P} = \frac{(1-r)\times S}{r} + 1$.

Case B, $P > \widetilde{P}$: FPGA becomes a bottleneck and CPU has idling cycles. When P is larger than $\widetilde{P}$, the FPGA is fully utilized and CPU cores have to wait more cycles in addition to $\frac{r}{S} \times t$. As shown in Figure 4.1c, after core 1 finishes the non-accelerated part of batch i, it sends requests to the accelerator task queue. Since the offloaded task from core P in batch i-1 has not finished yet, core 1 needs to wait extra gap cycles before its task can be executed on the FPGA. In this case, launching more CPU cores than $\widetilde{P}$ does not further improve the application performance. Application runtime now equals the total runtime for FPGA to finish kernel execution from all M tasks, as $T_{1b} = M \times \frac{r \times t}{S}$. Equivalently, the application runtime is equal to the total runtime for CPU cores to finish $\frac{M}{\widetilde{P}}$ batches of tasks, which is $T_{1b} = \frac{M}{\widetilde{P}} \times (1 - r + \frac{r}{S}) \times t$. As $\widetilde{P} = (\frac{(1-r)\times S}{r} + 1)$, we can rewrite $T_{1b} = \frac{M}{\widetilde{P}} \times (1 - r + \frac{r}{S}) \times t = \frac{M}{\widetilde{P}} \times (1 - r)(1 + \frac{r}{(1-r)\times S}) \times t = \frac{M}{\widetilde{P}} \times (1 - r)(1 + \frac{1}{\widetilde{P}-1}) \times t = \frac{M}{\widetilde{P}-1} \times (1 - r) \times t$. Using basic algebra rules, we have

$$\begin{aligned} T_{1b} &= M \times \frac{r}{S} \times t = \frac{M}{\widetilde{P} - 1} \times (1 - r) \times t, \\ &= M \times t \times \frac{r + (1 - r)}{S + \widetilde{P} - 1} = \frac{M}{\widetilde{P} - 1 + S} \times t \end{aligned} \tag{4.4}$$

Actually, $T_{1b} = \frac{M}{\widetilde{P}-1+S} \times t$ is quite intuitive to understand, as there are in total M tasks, each with time t, and in the system there is a fully utilized FPGA that works as S CPU cores, and $\widetilde{P}$ equivalent CPU cores. The minus 1 CPU core accounts for the penalty of CPU time that is spent on waiting for the FPGA kernel to be finished.

As we have T_{1b}, the total cost is:

$$C_{1b} = T_{1b} \times (P + CR) \times c = M \times t \times c \times (\frac{P + CR}{\widetilde{P} - 1 + S}) \tag{4.5}$$

Comparing C_{1b} in Equation 4.5 to C_0 in Equation 4.1, *Cost Efficiency Index* $I = \frac{C_{1b}}{C_0} = \frac{P+CR}{\widetilde{P}-1+S} = \frac{r(P+CR)}{S}$.

4.3 Generic Model

We summarize the above two cases and derive a generic model for the *Cost Efficiency Index* of a straightforward CPU-FPGA system compared to a CPU-only system:

$$I = \begin{cases} (1 - r + \frac{r}{S})(1 + \frac{CR}{P}) & \text{if } P \leq \widetilde{P} = (\frac{(1-r) \times S}{r} + 1) \\ \frac{P+CR}{\widetilde{P}-1+S} = \frac{r(P+CR)}{S} & \text{if } P > \widetilde{P} = (\frac{(1-r) \times S}{r} + 1) \end{cases} \tag{4.6}$$

Table 4.1: Analysis of Cost Efficiency Index I for HTC and Mutect2 on Amazon EC2 f1.2xlarge, S = 40, P = 8, CR = 25.

Application	r	S	$\widetilde{P}$	Case A or B	I
HTC	39%	40	64	A (8 < 64)	2.56×
Mutect2	89%	40	6	B (8 > 6)	0.73×

As shown in Table 4.1, HTC and Mutect2 both use the PairHMM kernel. But the kernel proportions are different. Typically, pairHMM kernel in HTC takes only 39%, while taking 89% in Mutect2. We get these two numbers by profiling the portions of PairHMM in all the datasets shown in Table 6.5. We implement a PairHMM kernel on an FPGA board on the Amazon EC2 `f1.2xlarge` instance; it has 40× speedup

Table 4.2: Analysis of Cost Efficiency Index I for HTC and Mutect2 on Huawei Cloud fp.1c, S = 43, P = 32, CR = 23.

Application	r	S	$\widetilde{P}$	Case A or B	I
HTC	39%	43	68	A (32 < 68)	1.06×
Mutect2	89%	43	6	B (32 > 6)	1.14×

than a single CPU core. To match the computation throughput of CPU cores with the PairHMM kernel in HTC, we need $\widetilde{P} = \frac{(1-39\%)}{39\%} \times 40 + 1 = 63.6$ cores, which is much larger than the number of CPU cores ($P = 8$) on that AWS `f1.2xlarge`. In this case, the FPGA board in HTC is severely underutilized. As shown in Table 4.1, *Cost Efficiency Index I* in HTC using FPGA is about 2.56× than using pure CPUs. On the other hand, for Mutect2, we only need six cores ($\widetilde{P} = 6$), which means that equivalently there are only six working CPU cores, and two other cores are idling.

Similarly, Table 4.2 shows I on the Huawei Cloud `fp.1c` instance that has 32 CPUs and CR = \$2.83/\$1.64*32-32 = 23. For HTC on this platform, when there are 32 CPU cores, FPGA utilization is better than that on the Amazon `f1.2xlarge` instance. However, it is still not fully utilized and I is still larger than 1. For Mutect2, *Matching Core Number* $\widetilde{P}$ is 6, which leaves 26 CPU cores idling. Thus, I is higher and now it is 1.14× larger than 1.

4.4 General Discussions

4.4.1 Cost Modeling Analysis

We apply the analytic model to two real-world platforms, e.g., Amazon AWS F1 where P is 8 and CR, cost ratio is 25, and Huawei Cloud fp.1c where P is 32 and

CR, FPGA cost ratio CR is 23. Figure 4.2 shows *Cost Efficiency Index* I versus different accelerator kernel ratio r and speedup S. Depending on P and $\widetilde{P}$, there are two regions: region A is when $P < \widetilde{P}$ and FPGA has idling cycles; region B is when $P > \widetilde{P}$ and FPGA is fully utilized. Within the two regions, depending on whether I is larger than 1 or not, we further divide the region A into regions A1 and A2 and the region B into regions B1 and B2.

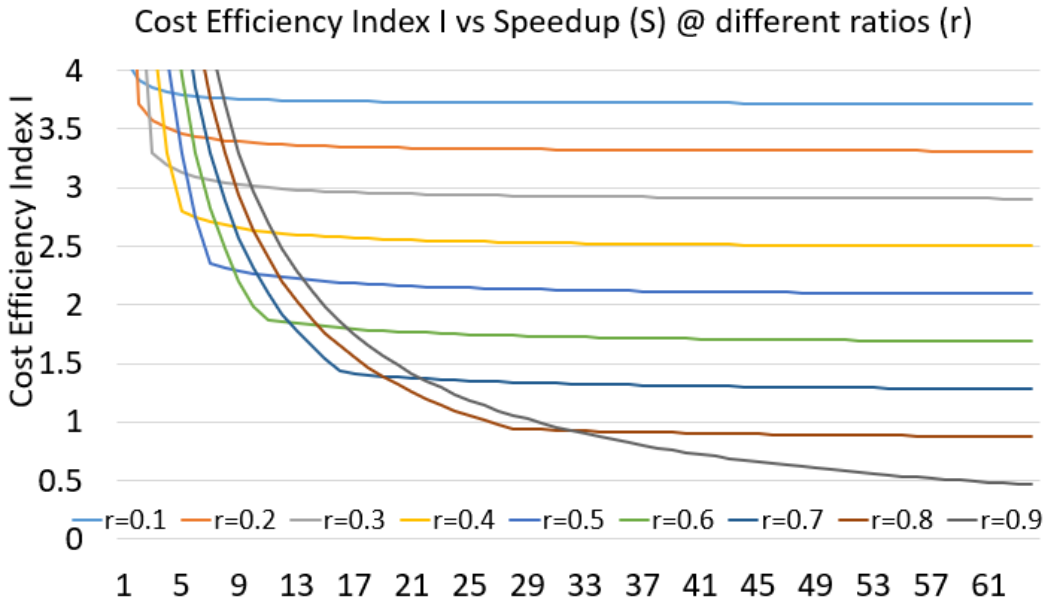


Figure 4.2: Cost Efficiency Index (I) vs. kernel ratio (r) and accelerator speedup (S) when CR = 25 on AWS EC2 F1.

Region A1: Cost inefficiency in region A where FPGA has idling cycles and $P < \widetilde{P}$. CPU cores are all allocated to the non-accelerated part of the program. And maximum computation throughput of FPGA is faster than that of CPUs. In other words, CPUs become the system bottleneck, and FPGA has idling cycles. Depending on the kernel ratio r and speedup S, for example, when r is too small and

S is large, the FPGA can be idling most of the time, which wastes the extra CR dollars per hour that users have paid. Idling of FPGA resource in this case is the root cause of cost inefficiency.

Region A2: Cost efficiency in region A where FPGA has idling cycles and $P < \widetilde{P}$. All CPU cores are allocated to work on the non-accelerated part of the program. FPGA always waits for CPUs in this case. However, the cost benefit from FPGA acceleration is larger than the waste on the idling cycle, and the overall cost efficiency is still improved.

Region B1: Cost inefficiency in region B where FPGA is fully utilized and $P > \widetilde{P}$. FPGA accelerator speedup S is smaller than CR in this region. In other words, FPGA costs the same as CR CPU cores, but it does not bring as much speedup as CR cores. This is the root cause of cost inefficiency.

Region B2: Cost efficiency in region B where FPGA is fully utilized and $P > \widetilde{P}$. FPGA accelerator speedup S is larger than CR in this region. Both CPUs and FPGAs are busy all the time. These facts explain why the CPU-FPGA platform achieves a better cost efficiency than the CPU-only solution.

4.4.2 Insights and Optimization

For accelerator design, from Equation 4.6, we offer the following insights:

1. To achieve higher cost efficiency for the CPU-FPGA platform, speedup S should be larger than CR. This can also be seen in Figure 4.2, when $S < CR = 25$, *Cost Efficiency Index* I is always larger than 1.
2. To achieve a higher cost efficiency for the CPU-FPGA platform, ratio r should be

larger than $1 - \frac{P}{P+CR}$. This can also be seen in Figure 4.2. When $r < 0.758$, *Cost Efficiency Index I* is always larger than 1.

3. If an application falls into region A1, we should use more CPUs to improve the performance instead of using FPGAs. This usually happens when the accelerated kernel consumes a small fraction of the total application and FPGA kernel speed-up is huge.

4.4.3 Different Cost Ratios (CR)

We show *Cost Efficiency Index I* on different kernel ratio (r) and accelerator speedup (S) when CR are different in Figure 4.3 (CR = 20), 4.4 (CR = 15) and 4.5 (CR = 10) respectively. When CR decreases, there are more regions that could achieve *Cost Efficiency Index I* smaller than 1. For example, in GATK pipeline, the FPGA accelerator developed by Falcon Computing Solutions [Com18] for Smith-Waterman in BWA achieves speedup (S) as 16. For AWS and Huawei Cloud, using FPGA accelerator can be more cost-efficient than CPU solutions if FPGA instances are priced lower than current price such that CR is smaller than 16.

4.5 Conclusion

To sum it up, for straightforward CPU-FPGA integration, *I* depends on *r, S, P, CR* and is not guaranteed to be smaller than 1, which means the out-of-pocket cost is not optimized because either the CPU or FPGA is underutilized. This motivates us to implement a more efficient CPU-FPGA integration framework to improve the utilization of FPGA (Case A) or CPU (Case B) in order to reduce the overall

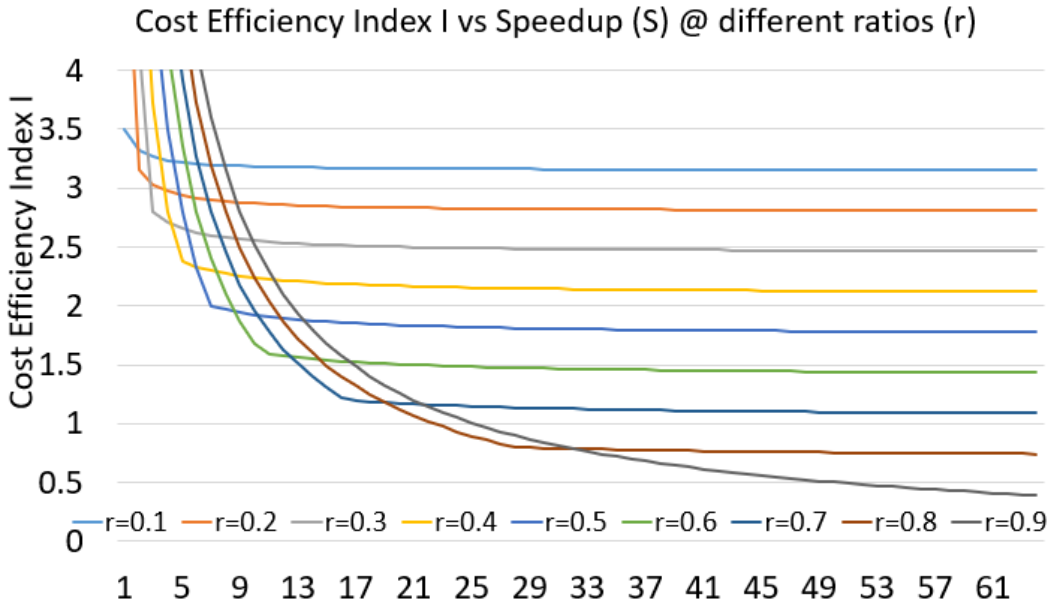


Figure 4.3: Cost Efficiency Index (I) vs. kernel ratio (r) and accelerator speedup (S) when CR = 20.

application cost and achieve a higher cost efficiency than CPU solutions—that is, a lower I that is smaller than 1 in Chapter 6.

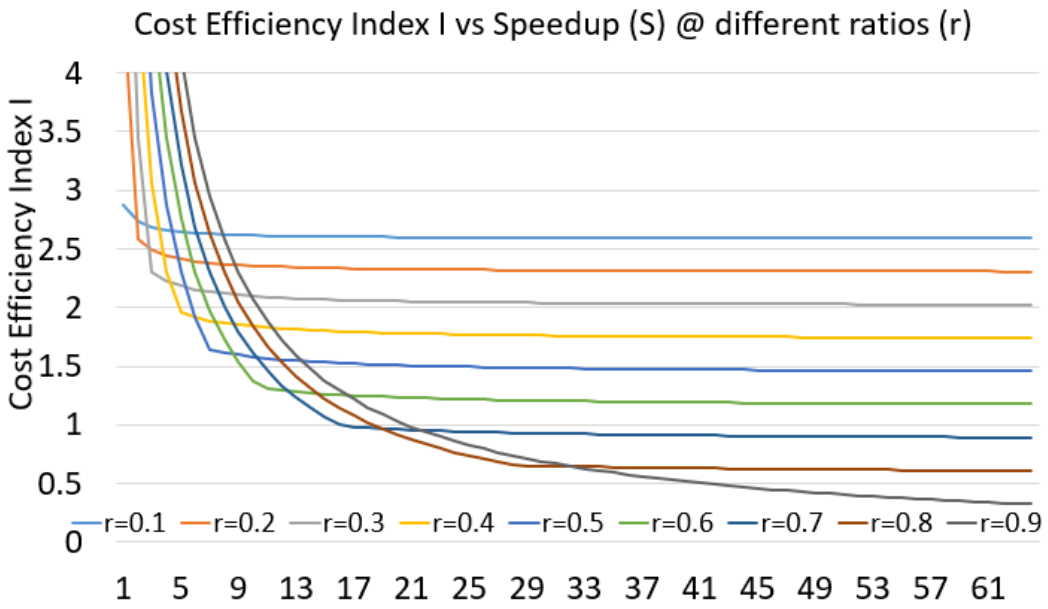


Figure 4.4: Cost Efficiency Index (I) vs. kernel ratio (r) and accelerator speedup (S) when CR = 15.

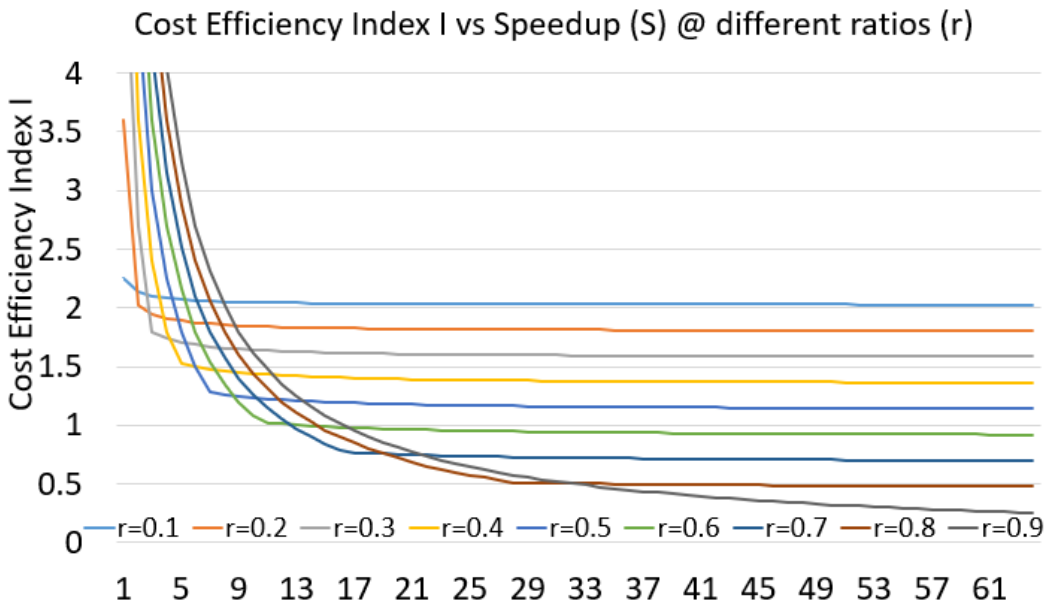


Figure 4.5: Cost Efficiency Index (I) vs. kernel ratio (r) and accelerator speedup (S) when CR = 10.

CHAPTER 5

Node-Level Performance and Cost Modeling: Storage Resources

5.1 Introduction

In conventional Hadoop MapReduce applications, I/O used to play a heavy role in the overall system performance. More recently, a study from the Apache Spark community—state-of-the-art in-memory cluster computing framework—reports that I/O is no longer the bottleneck and has a marginal performance impact on applications like SQL processing. However, we observe that simply replacing HDDs with SSDs in a Spark cluster can have over 10x performance improvement for certain stages in large-scale production-quality genome processing. Therefore, one key question arises: *How does I/O quantitatively impact the performance of today's big data applications developed using in-memory cluster computing frameworks like Apache Spark?*

In this chapter we select an important yet complex application—the Spark-based Genome Analysis ToolKit (GATK4)—to guide our modeling. We first use different combinations of HDDs and SSDs to measure the I/O impact on GATK4 and change the CPU core number to discover the relation between computation and I/O access. Combining with Spark underlying implementations, we further

analyze the inherent cause of the above observations and build our model based on the analysis. Although building upon GATK4, our model maintains generality to other applications. Experimental results show that we can achieve an performance prediction error rate within 10% for typical Spark applications of both iterative and shuffle-heavy algorithms. Finally, we further extend our model to a broader area - that of optimal configuration selection in the public cloud. In Google Cloud, our model enables us to save 38% to 57% cost for genome sequencing compared with its recommended default configura- tions. Currently, more and more companies are adopting cloud computing for specific workloads. Our proposed model can have a huge impact on their choices, while also enabling them to significantly reduce their costs.

5.2 Background

5.2.1 Apache Spark

Apache Spark [ZCD12a] is a widely used in-memory large-scale data processing framework. Spark exposes a programming model to big data application developers based on resilient distributed datasets (RDDs). The RDD abstraction provides a series of *transformations* (e.g., map, filter) and *actions* (e.g., collect, count) that enable lazy evaluation of data partitions over a cluster of nodes. By caching RDDs in memory and thus reducing I/O accesses, Spark often achieves significant performance improvement over the Hadoop MapReduce [Whi12a] framework.

A typical Spark application launches its driver program on a master node and then distributes its tasks (data partitions) into a cluster of slave/worker nodes for

parallel processing. Each slave node will read its data partitions from a distributed file system (e.g., HDFS [SKR10]) and perform parallel computations. In addition, each slave node in Spark has its local storage directory (spark.local.dir) to store data, including RDDs, that persist on disk specified by a user program, or intermediate data [Spa17a, Spa17b] preserved by the Spark framework. In the following we will interchangeably use the term Spark Local to refer to this local directory.

5.2.2 Genome Analysis ToolKit (GATK4)

Building a performance model for a complex distributed system like Spark is not trivial. Therefore, we first analyze a realistic and complex Spark application—GATK4—and use its analyzing result to guide our modeling.

GATK4 is a Spark-based production-quality genome analysis toolkit widely used in computational genomics. It includes three major stages: 1) MarkDuplicate (MD), which groups reads (small DNA fragments from the biochemical sequencing machine) by alignment information and marking duplicate reads; 2) BaseRecalibrator (BR), which builds a statistical model on how to update the quality scores of the aligned reads; and 3) SaveAsNewAPIHadoopFile (SF), which updates the quality scores and saves the analysis-ready reads into storage. In addition to the genome reads in the BAM file [LHW09, NKS12], GATK4 also takes two other input files: 1) the VCF file that contains all known genome variations, and 2) the reference genome file to which all the genome reads are aligned. The output of GATK4 can be used to conduct genome-wide association studies (GWAS, also known as population studies) to discover unknown genome variations. The main data flow of GATK4 in the Spark perspective is shown in Fig. 5.1.

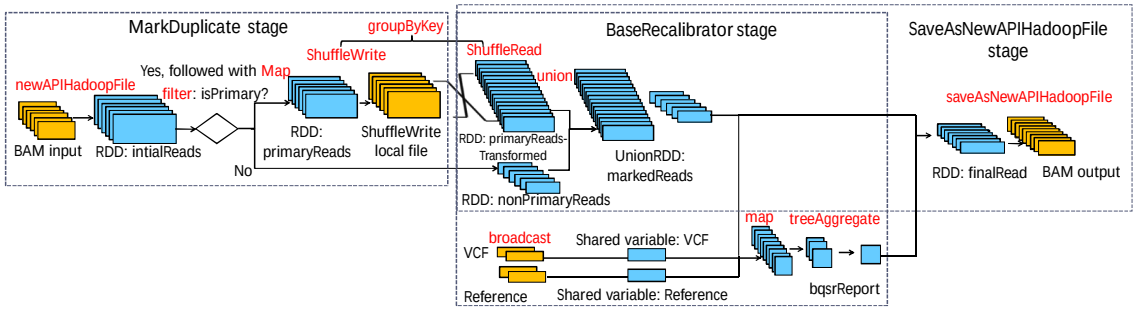


Figure 5.1: The Spark RDD flow of GATK4 pipeline.

5.2.3 Experiment Setup

Table 5.1 describes the system software and hardware configuration for each slave node. Table 5.2 describes the default Spark and HDFS configuration in our cluster setting unless otherwise specified. Although we use fixed Spark configurations here, out proposed model can work for other configurations as well. We will revisit this in Section 5.4.

Table 5.1: Software and hardware configuration

System	Linux kernel	2.6.32-642
	CPU	2×Intel Xeon CPU E5-2699 v3 = 36 cores
	RAM size	128 GB
	Network	10Gb/s
HDD	Model	Western Digital 4000FYYZ-01UL1B2
	RPM	7200
	Capacity	4 TB
	max_sectors_kb	512 KB
SSD	Model	SAMSUNG MZ7LM240HCGR-0E003
	Capacity	240 GB
	max_sectors_kb	512 KB

As an example, we process a single whole human genome with 30x coverage sampled from a patient with breast cancer (HCC1954 [GKV98]). This genome contains 500 million read pairs, each read with around 101 nucleotides. This input genome in a compressed BAM file is around 122GB, and the output analysis-ready

Table 5.2: Spark and HDFS configuration

Spark	version	spark-1.6.2
	SPARK_WORKER_CORES	36
	SPARK_WORKER_MEMORY	90 GB
HDFS	version	hadoop-2.6.0
	dfs.blocksize	128 MB
	dfs.replication	2
JAVA	version	jdk 1.8.0_73

genome file (also in compressed BAM format) is around 166GB. To measure the impact of I/O on the GATK4 performance, we test its performance under four different HDD/SSD configurations for each node, as summarized below.

Table 5.3: Hybrid configurations of HDDs and SSDs

Configuration	**1**	**2**	**3**	**4**
HDFS	1 SSD	1 HDD	1 SSD	1 HDD
Local (spark.local.dir)	1 SSD	1 SSD	1 HDD	1 HDD

We leave the description of five other typical Spark applications in Section 5.5.

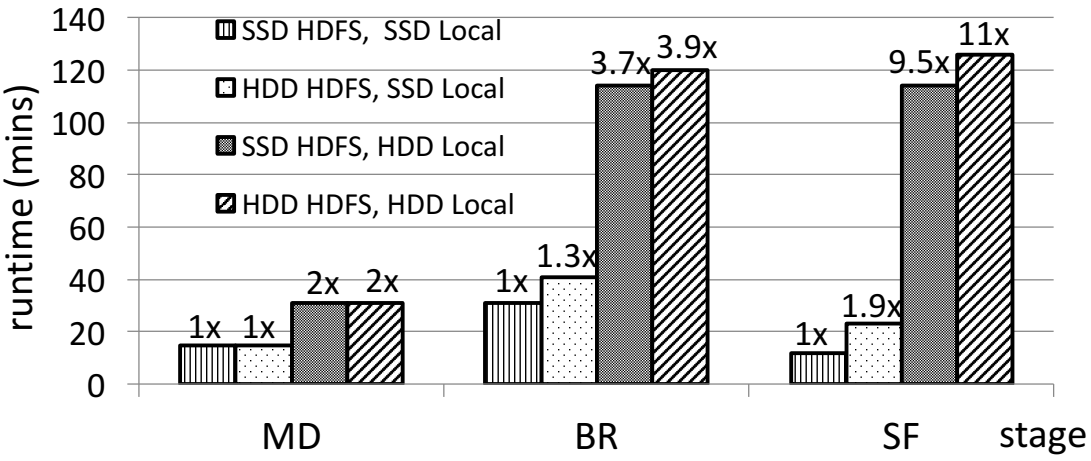


Figure 5.2: Runtime for different stages in GATK4 using 500M read pairs input.

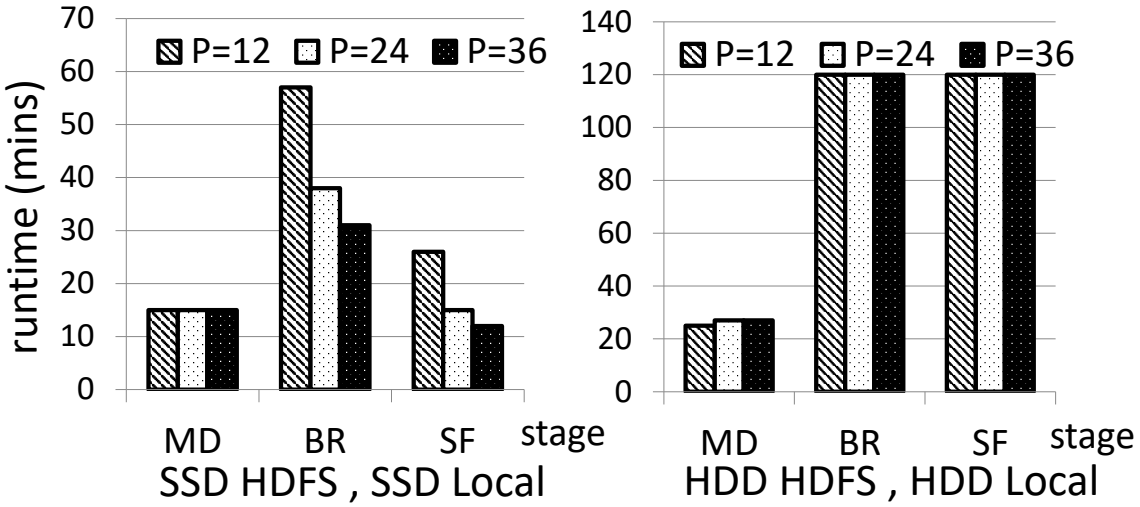


Figure 5.3: Runtime for 2HDD and 2SSD cases when the number of CPU cores per node P = 12, 24, 36.

5.3 GATK4 Performance Analysis

Using the Spark-based GATK4 as a motivational example, we first measure the impact of I/O on its performance under different execution stages with different RDD operations, and different numbers of CPU cores. After that, we summarize the observations that motivate us to build an I/O-aware performance analytical model for Spark applications. A **four-node** small cluster (one for master) is used in this example.

5.3.1 GATK4 Performance Profile

We break down the I/O impact into different GATK4 stages using 500M read pairs input and 36 Spark executor cores. As shown in Fig. 5.2, we find that the performance impact varies considerably in different stages with different RDD operations. The IO data sizes for different stages are shown in Table 5.4.

Table 5.4: I/O data size (GB) in different GATK4 stages

I/O (GB)	HDFS read	Shuffle write	Shuffle read	HDFS write
MD	122	334	0	0
BR	122	0	334	0
SF	122	0	334	166

1. Changing the HDFS folder from HDD to SSD has no performance gain for the MD stage (though MD has the same I/O data access size as the BR stage according to Table 5.4), up to a 30% and 90% performance gain for the BR and SF stages, respectively.
2. The sensitiveness to Spark Local bandwidth varies in the different stages. It is interesting that the major time-consuming stage changes from BR to SF and BR when switching Spark Local from HDD to SSD.
3. Spark Local is much more IO-sensitive than HDFS.

Finally, we evaluate the I/O impact on different stages under different numbers of CPU cores, using 500M read pairs input. As shown in Fig. 5.3, when the number of cores (P) increases from 12 to 36, the runtime for the BR and SF stages decreases in the 2SSD configuration but stays the same in the 2HDD configuration. For the MD stage, the runtime almost stays the same when P changes in the 2SSD as well as 2HDD configurations. We observe that when there are more cores, SSDs *might* achieve a larger performance gain than HDDs.

As observed above, I/O still plays a heavy role in the GATK4 performance. As we will demonstrate in Section 5.5.2, even in typical Spark iterative algorithms where intermediate data is cached in memory, I/O can play an important role with a small iteration number (less than 100). The performance gap between HDD and SSD can be as large as 2x.

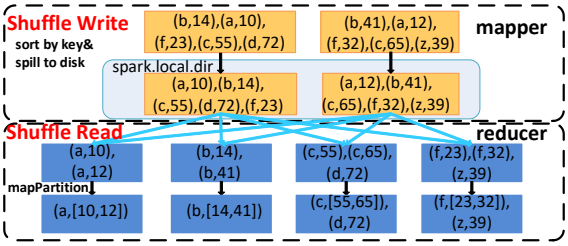
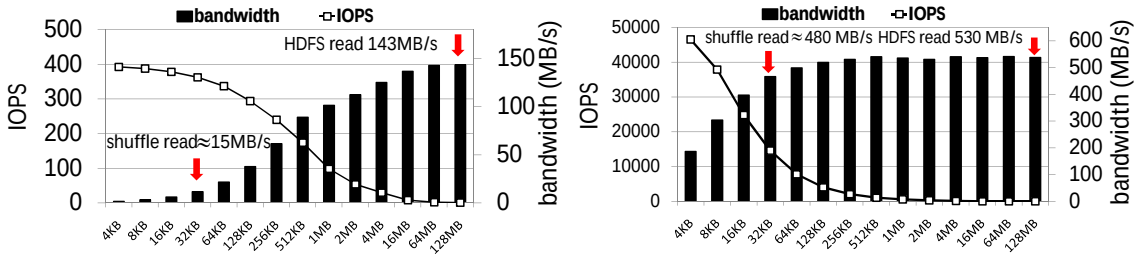


Figure 5.4: An illustration of the groupByKey operation.



(a) HDD (b) SSD

Figure 5.5: Read bandwidths and IOPs for HDD and SSD on different block sizes.

5.3.2 I/O-intensive Operations

After a detailed analysis, we find there are two other major I/O-intensive operations in GATK4, in addition to the HDFS read and write of the input and output file. These two operations are discussed in detail in the following.

5.3.2.1 Shuffle-Heavy RDD Operations

Certain RDD operations in Spark, such as groupByKey() and repartition(), involve very time-consuming shuffle operations. During shuffling, mappers write all the intermediate data into the Spark Local. Later, reducers read those intermediate results from the mappers' local storage [Spa17a, Spa17b]. Such RDD operations involve a significant amount of I/O accesses.

Fig. 5.4 presents an overview of the groupByKey() operation used in Spark, which

has two phases: ShuffleWrite and ShuffleRead. To redistribute and group the data based on its key, the ShuffleWrite phase generates *map* tasks to sort the data based on its key and spills the output from the *mapper* side onto Spark local storage after data serialization and compression. The ShuffleRead phase generates *reduce* tasks to collect and aggregate data from all the *mappers*. In this phase, data is read from the Spark local storage and network, and then decompressed and deserialized. ShuffleRead is the phase where both disk and network are involved, and data redistribution among different data partitions occurs. In GATK4, the MD and BR stages are separated by the ShuffleWrite and ShuffleRead phases as shown in Fig. 5.1. Though shuffle-related operations also involve network communication, the 10Gbps network usually is not the bottleneck of Spark applications [Cut15]. Hence, we mainly focus on the analysis and modeling of the I/O part.

5.3.2.2 Large RDDs NOT Cacheable in Memory

Another major source of I/O access comes from multiple references to large RDDs that consume a large amount of memory and cannot be cached in memory. One example is the UnionRDD markedReads in GATK4, which is used by both the BR and SF, as shown in Fig. 5.1. Since this RDD is not cached in the memory, each time the program uses and performs actions on it, it will be 1) read from the persist storage (Spark Local) if there is a persist write for this RDD, or otherwise 2) recomputed using input data from either the HDFS or Spark Local.

To explain why this UnionRDD cannot be cached in memory in GATK4, we change the original GATK4 to cache it for a small input (compressed, serialized data), and then measure its runtime memory consumption (decompressed, deserialized data).

Based on this information, we find that caching this single UnionRDD for the whole genome with 122GB input requires around 870GB memory space. Assuming that around 40% of the entire Spark executor memory is used as storage memory to cache this UnionRDD, the total Spark memory required would be around 2.18TB. Each server that we use has 128GB RAM, and if we allocate 90GB for the Spark executor, we need 25 slave nodes in total, which is quite expensive for practical usage. Therefore, such large RDDs can not be fully cached in memory and need to be persisted in disk or simply be rebuilt from input anytime that a program uses them. Both of these approaches will incur a large amount of I/O access.

5.3.3 Effective I/O Bandwidth under Various Data Request Sizes

As previously analyzed, the big performance gap of different storage configurations is mainly caused by switching the Spark Local from HDD to SSD. We find that the effective I/O bandwidth in the shuffle read stage has up to a 32x gap for SSD and HDD, which is much larger than the 3.7x gap of their peak bandwidths. In this section we use 36 Spark executor cores and the whole genome input unless otherwise specified.

After a detailed analysis, we find that unlike the HDFS read/write that usually involves large data block accesses (e.g., 128MB), shuffle read incurs many small block size I/O accesses, where HDDs have a much lower effective bandwidth than SSDs. As a result, replacing a HDD with a SSD for the Spark Local for BR and SF stages can achieve up to 3.7x and 9.5x performance gains, respectively.

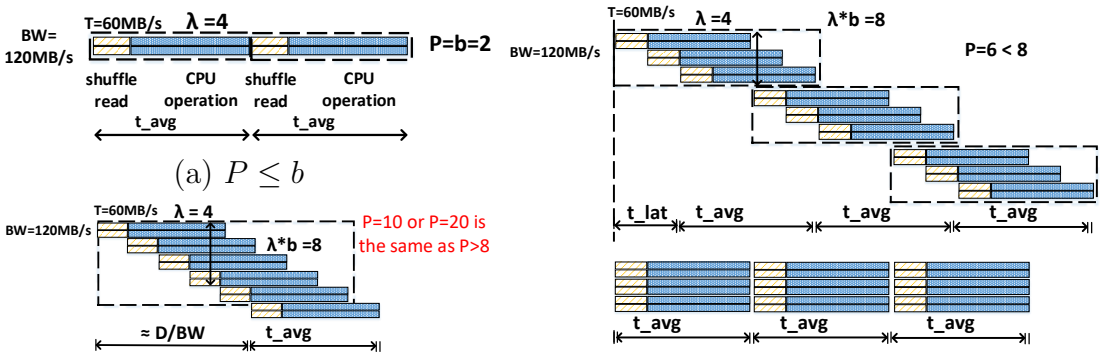
5.3.3.1 Effective I/O Bandwidth on HDD and SSD

We use fio [SAA11] to test the input/output operations-per-second (IOPS) and effective bandwidth on different read block sizes for HDD and SSD to simulate the shuffle read operation and HDFS read operation in Spark. As shown in Fig. 5.5a and 5.5b, when block size is 30KB, the average bandwidth is 15MB/s for HDD and 480MB/s for SSD, which means a 32x bandwidth gap. For such a small data access size, shuffle read I/O in HDD becomes the bottleneck of the system, while shuffle read I/O in SSD does not. The bandwidth gap between HDD and SSD is higher when the block size is smaller. When the block size is 4KB, the gap can be as high as 181x. When the block size is 128MB (default in HDFS block size), the gap is only around 3.7x.

5.3.3.2 Why Shuffle Read Accesses Small Data Blocks

Now we analyze why shuffle read involves numerous small data block accesses. As shown in Fig. 5.4, assume the mapper side has M partitions. There are M local output files that are indexed with all the reducer IDs. On the reducer side, assume there are R tasks. Each reducer reads shuffle data indexed with its own reducer ID from M separate files in the mapper side, and dynamically merges the data. To fit the data read by each reducer in memory for use in later RDD operations (e.g., unionRDD in GATK4), there is usually a fixed data size for each reducer (e.g., each reducer in GATK4 reads 27MB shuffle data). Since the fixed amount of data in each reducer comes from M mapper files, when M is large it will incur a large amount of small block size I/O accesses.

In GATK4, the number of mappers M is determined by the partition number of



(a) $P \leq b$

(b) $b < P \leq \lambda \times b$

(c) $P > \lambda \times b$

Figure 5.6: Execution model for (a) $P \leq b$, no I/O contention; (b) $b < P \leq \lambda \times b$, I/O contention is hidden by the CPU computation; and (c) $P > \lambda \times b$, I/O becomes a bottleneck, and increasing the number of CPU cores P does not help.

the RDD initialReads as shown in Fig. 5.1, i.e., the number of HDFS blocks of the input HDFS file. When the input is a whole genome, M = 122GB*1024(MB/GB) / 128(MB/HDFS block) = 973. The number of reducers R is set so that each reducer task reads in 27MB shuffle data as tuned in the original GATK4. On average, each reducer reads around 27MB / 973 = 30KB data from each mapper. We also use iostat to measure the average I/O request size (in sectors, 512B per sector) in the BR and SG; the average request size is 60, which corresponds to the 30KB (≈512B×60) block size.

5.3.3.3 Shuffle Performance Analysis

According to Fig. 5.5a and 5.5b, the HDD small block access bandwidth is only 15MB/s, which also matches the result of iostat. Hence, the time needed for shuffle read (334GB as in Table 5.4) is 334 * 1024(MB) / 3 / 15(MB/sec) / 60(sec/min) = 126mins, which perfectly matches the execution time of both BR and SF shown in Fig. 5.2. This further indicates that all computation time is overlapped by HDD

access. One may notice in Table 5.4 and Fig. 5.4, although the shuffle data size of MD is exactly the same as BR and SF, the execution time of MD is much shorter. That is because the block size of shuffle write is much larger than shuffle read—about 365MB in this case since mappers write data in sorted chunks (as described in Section 5.3.2.1).

5.4 I/O-Aware Spark Analytical Model

To reason and quantify the underlying behavior of Spark tasks with different RDD operations, we propose an I/O-aware analytical model that considers the effective I/O bandwidths under different data request sizes and different numbers of CPU cores, and the overlap between the CPU computation and I/O access. For illustration purposes, we use Spark shuffle read as an example to explain our model, which works similarly for other I/O or computation-intensive RDD operations.

5.4.1 Model Variable Definition

Shown in Fig. 5.6a, we define the following variables used in our model.

1. T is the I/O (here, shuffle read) throughput per core when there is no I/O bandwidth contention. Usually we can test this T under the SSD configuration using a single core for the Spark executor. Assume $T = 60$MB/s for illustration purposes, which might not be the real measured throughput.
2. Similarly, t_avg is the average execution time of a single task (for a single data partition).
3. λ is the average time ratio of the entire task execution to the I/O access. Assume

$\lambda = 4$ in this example.

4. BW is the effective I/O bandwidth under the average data request size in the I/O operation. Assume BW = 120MB/s in our example.
5. $b = \frac{BW}{T}$ is the break point for the number of CPU cores per node, after which the CPU cores will contend for the limited I/O bandwidth. In the example shown in Fig. 5.6a, $b = 2$.
6. D is the entire data access size.
7. P is the number of actual launched executor cores per node.
8. N is the number of nodes.
9. M is the number of tasks (data partitions).

5.4.2 Different Execution Phases

When we gradually increase P from 1 to the number of maximum executor cores, there are three phases where the runtime model and I/O access are different.

***P*≤*b*: no overlap with I/O and CPU computation**. In this case, I/O is not a bottleneck as the number of executor cores does not exceed the bandwidth break point. As shown in Fig. 5.6a, after a batch of P tasks finish their execution, another batch of P tasks are launched. Here we only show two batches of tasks, and there is no overlap with I/O access (shuffle read) and CPU computation. Therefore, the estimated runtime formula is $\frac{M}{N*P} \times t\_avg$.

***b*<*P*≤*λ*×*b*: overlap with I/O and CPU computation within a batch**. As shown in Fig. 5.6b (right column of Fig. 5.6), P tasks are launched in a batch. Since b tasks already saturate the I/O bandwidth, the next b tasks start I/O operations

after the first b tasks finish their I/O operation.[1] When $P \leq \lambda \times b$, the first b tasks in the second batch start the I/O operation (and also include some computation-like decompression) right after the first b tasks in the first batch finish, i.e., when the CPU cores are available. Here we show three batches of tasks. After an initial latency t_lat, each batch of tasks finishes in t_avg. Therefore, the estimated runtime formula is $\frac{M}{N*P} \times t\_avg + t\_lat$.

We find that in this case, the estimated runtime formula is almost the same as that of the $P \leq b$ case, since the CPU computation can hide the I/O access. *We conclude that the performance of parallel part (that is the left part of above formula) scales with the number of CPU cores P as long as $P \leq \lambda \times b$, where I/O does not hit the bandwidth break point or is hidden by the CPU computation.* We can define $\boldsymbol{B=\lambda \times b}$, as the turning point where I/O starts to become a bottleneck.

$P>\lambda \times b$, *i.e.*, $P>B$: I/O becomes a bottleneck. As shown in Fig. 5.6c, when P increases further, there is more overlapping of I/O and CPU operations. Since $\lambda \times b$ determines the maximum parallelism of CPU tasks, if P is larger than that, it means I/O becomes a bottleneck. In this case, the estimated runtime formula is $\frac{D}{N \times BW} + t\_avg$, which is determined by the entire data access size and effective I/O bandwidth. *That is, further increasing the number of CPU cores P does not help the system performance when $P > B$.*

[1] In the real case, all P tasks are launched at the same time, where Fig. 5.6b draws equivalent sequential I/O access with no I/O contention for easy illustration.

5.4.3 Generic Model

Therefore, our model can be generalized as follows: for each stage i, its runtime t_{stage} is:

$$\begin{aligned} t_{stage} &= \max\left(t_{scale}, t_{read\_limit}, t_{write\_limit}\right), \\ t_{scale} &= \frac{M}{N * P} \times t\_avg + \delta_{scale} \\ t_{read\_limit} &= \frac{D_{read}}{N * BW_{read}} + \delta_{read}, \\ t_{write\_limit} &= \frac{D_{write}}{N * BW_{write}} + \delta_{write} \end{aligned} \tag{5.1}$$

Here t_{scale} is the execution time when none of the I/O read and write is a bottleneck, and its parallel part scales with $N * P$. t_{read_limit} (t_{write_limit}) is the I/O read (write) time when it becomes a bottleneck, i.e., $P > B_{read}$ ($P > B_{write}$). We add a constant δ to each term to accommodate the linear part of the code. The model is built for each stage, and for the entire application, the total runtime is the sum of each stage's runtime, $t_{app} = \sum t_{stage}$.

Note that our model relates to disk bandwidth rather than disk number. Thus, it is general enough to support the multi-disk case. In addition, different hardware platforms or Spark configurations will lead to different t_avg. Therefore, our model can still correctly capture the execution time.

5.5 Model Evaluation Results

To demonstrate how to use our I/O-aware model to explain and predict the runtime behavior of Spark programs, we first apply it to GATK4 and validate the model

accuracy. To better illustrate the generality of our model, we later apply it to typical big data applications from SparkBench [LTW15] and BigDataBench [WZL14]. Our experimental results demonstrate that our model can predict their performance with an error rate less than 10%, and can well explain their behaviors under different configurations. An **eleven-node** (one for master, ten for slave nodes) cluster is used for the experiments in this section.

5.5.1 Applying Model to GATK4

5.5.1.1 MD Stage

Changing the HDFS folder from an HDD to an SSD for the MD stage gives no performance gain when $P = 36$, as shown in Fig. 5.2. The HDFS read operation in MD only occupies a small portion of the task execution time. The time ratio of the entire task execution over I/O access $\lambda = 12$ is already pretty large. Although the break point b is different for HDD and SSD for HDFS read (4.3 and 16, respectively), B in both cases is larger than 36, the maximum number of executor cores per node in our setting.

When using SSD as Spark Local, the runtime is calculated as $t = t_{scale} = \frac{M}{N*P} \times t\_avg + \delta_{scale}$. To be noted here, in Fig. 5.3 MD stage time does not scale for SSDs. This is not because the I/O is the bottleneck, it is because the garbage collection time increases with larger P and dominates the execution time of MD, which is currently not included in our model and will be dealt with in future work. When using HDD as Spark Local, shuffle write becomes the I/O bottleneck, BW_{write} = 100MB/s, $B = 10$, and runtime does not scale for $P = 12$, 24, 36.

5.5.1.2 BR and SF Stage

There are two kinds of tasks in the BR stage. One starts from the RDD nonPrimaryReads that need HDFS read, with a $\lambda = 1.3$; i.e., the CPU computation time is small compared to the I/O access time. However, since most read records are filtered out after the filter() function, as shown in Fig. 5.1, this task only occupies a small portion of the total BR execution time. The other task starts from the shuffle read, and the CPU computation time is long, with a $\lambda = 20$. And this task dominates the BR execution time. Due to space constraints, we will mainly illustrate the modeling for the latter task.

We first explain the case when both the Spark Local and HDFS are set to separate SSDs. For shuffle read, if there is no I/O contention, each core's read throughput T is around 60MB/s. And λ, the time ratio of the entire task execution over shuffle read task time, is 20. The SSD shuffle read bandwidth BW is around 480MB/s. Thus the break point $b = \frac{BW}{T} = 8$. In this way, the BR stage runtime scales with the number of executor cores P up to $B = 160$ cores. This matches well with the results in Fig. 5.3: when P increases from 12 to 24 to 36, the runtime of BR decreases accordingly.

However, when changing the Spark Local to an HDD, where HDD shuffle read bandwidth for 30KB block size is only 15MB/s, even one core suffers the I/O contention in some sense, that is, b=1. Compared to the shuffle read time in SSD, the shuffle read time in HDD in each core is 4x longer, which means that λ here is 5. Thus, $B = 5$; this means the runtime of BR does not further decrease when P is larger than 5. Based on the above analysis, there is no performance gap between the HDD and SSD when P is small. As $P > 5$, it further increases the performance gap between the two storage configurations on BR, as seen in Fig. 5.3. This also explains

how the SF runtime scales with P. Since in SF λ is smaller, the performance gap when changing Spark Local from SSD to HDD starts even earlier than the BR.

5.5.1.3 Model Accuracy Results for GATK4

Fig. 5.7 presents the real measurements compared to model predicted runtime for different stages under different I/O configurations, when there are ten slave nodes, and P= 6, 12, 24. The average error rate is less than 6%, which is true for other omitted cases as well.

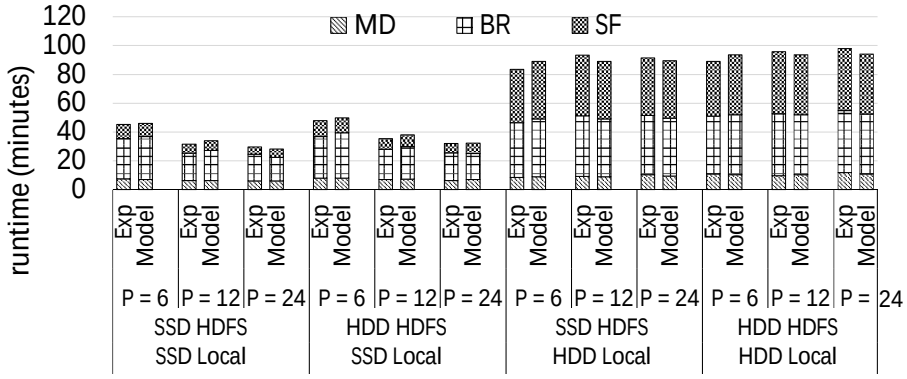


Figure 5.7: Comparison of measured runtime (exp) and model predicted runtime (model) for GATK4.

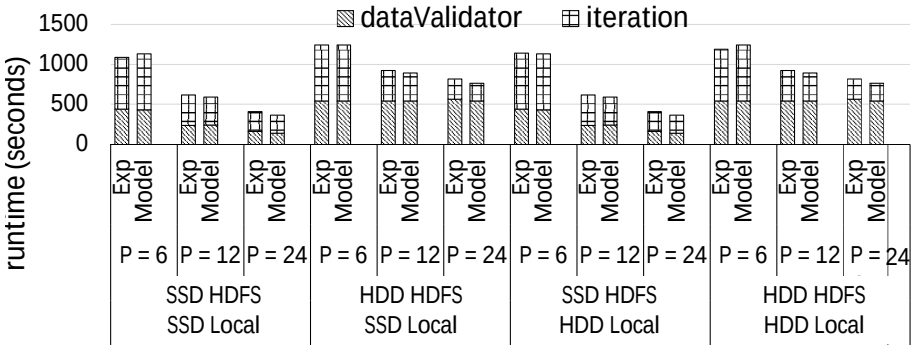


Figure 5.8: Comparison of measured runtime (exp) and model predicted runtime (model) for Logistic Regression (LR), small dataset.

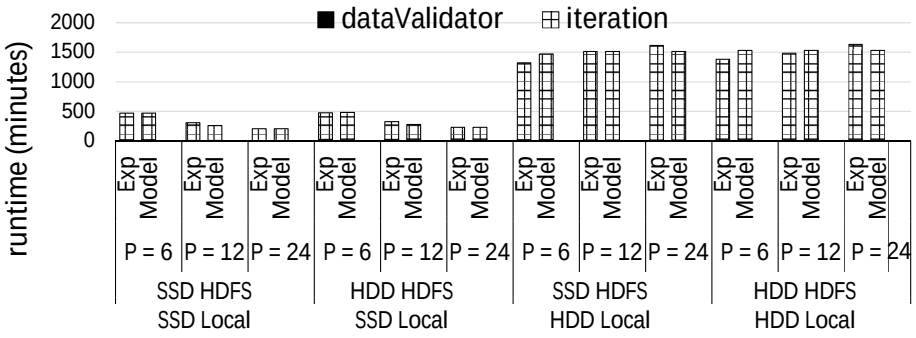


Figure 5.9: Comparison of measured runtime (exp) and model predicted runtime (model) for Logistic Regression (LR), large dataset.

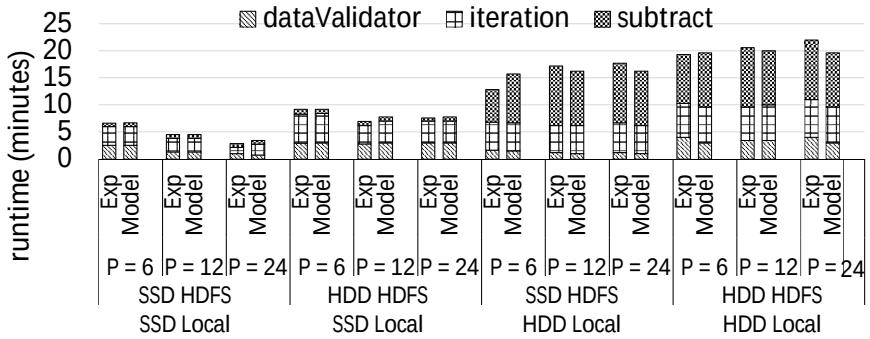


Figure 5.10: Comparison of measured runtime (exp) and model predicted runtime (model) for Support Vector Machine (SVM).

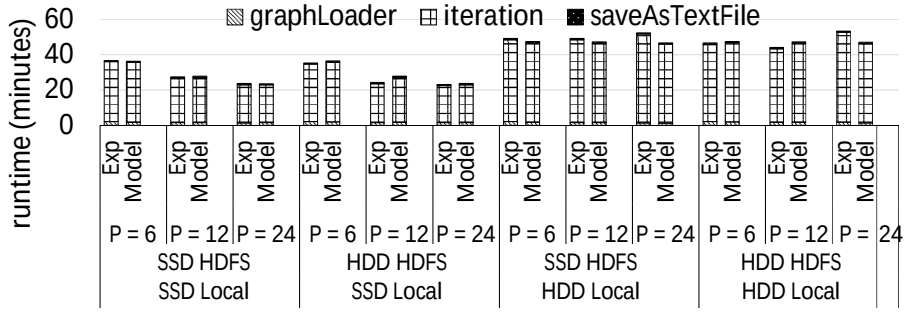


Figure 5.11: Comparison of measured runtime (exp) and model predicted runtime (model) for PageRank (PR).

5.5.2 Generality of Our Model: Other Applications

5.5.2.1 Logistic Regression

Logistic Regression in Spark Mllib [MBY16] is a typical iterative machine learning algorithm. It consists of two stages: dataValidator and iteration. In our experiment, We take two datasets generated by SparkBench: 1,200 (small) and 4,000 (larger) million examples, each with 20 features. The iteration number is set to 50 in this experiment. For the small dataset, the RDD parsedData generated from dataValidator can be cached in memory. For the large dataset, it is too large to be totally cached in memory and will be put in Spark Local. The sizes of RDD parsedData for the small and large datasets are 280GB and 990GB respectively. Results are shown in Fig. 5.8 and Fig. 5.9 with an average error rate of 5.3%.

5.5.2.2 Support Vector Machine

Support Vector Machine [SV99] is another typical iterative machine learning algorithm. It consists of three phases: dataValidator, iteration and subtract. Input dataset has 12 million samples, 1000 features, 1200 partitions. Iteration number is set to 10, and each iteration reads in 82GB in-memory cached RDD generated from dataValidator. The subtract phase incurs shuffle, and total shuffle size is 170GB. Results are shown in Fig. 5.10 with an average error rate of 8.4%.

5.5.2.3 PageRank

PageRank [PBM99] in Spark GraphX [XGF13] is an iterative graph algorithm that ranks the relative importance of webpages. It consists of three phases: graphLoader,

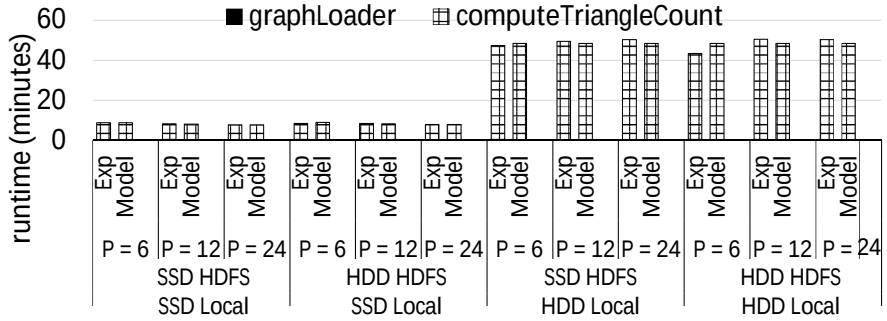


Figure 5.12: Comparison of measured runtime (exp) and model predicted runtime (model) for Triangle Count (TC).

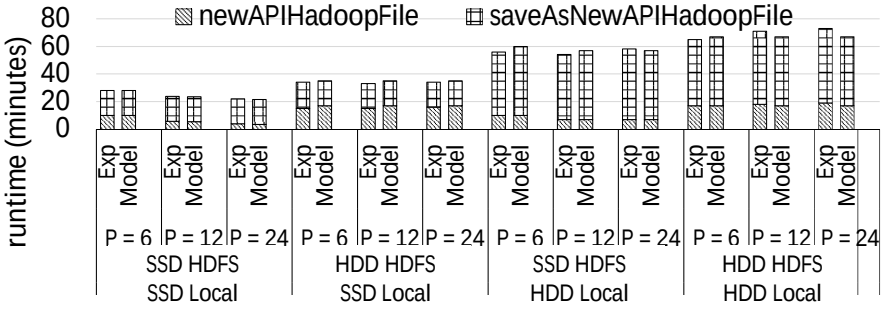


Figure 5.13: Comparison of measured runtime (exp) and model predicted runtime (model) for Terasort (TS).

iteration, and saveAsTextFile. We generate a dataset that has 20 million vertices, 4800 partitions (other data generator parameters are set as default). Iteration number is set to 10, and each iteration reads in cached RDD data from the last iteration and generates new RDD data for the next iteration to compute. The cached RDD total size is as large as 420GB, and it is larger than total executor storage memory space (assume 40% total executor memory is for storage) and persist in Spark Local. Results are shown in Fig. 5.11 with an average error rate of 5.2%.

5.5.2.4 Triangle Count

Triangle Count [SV11, KMP12] in Spark GraphX is a graph algorithm to count three-vertex small graphs within a large graph. It consists of two phases: graphLoader, computeTriangleCount. The computeTriangleCount phase will first repartition the graph to canonicalize [Spa17d] the graph so that there are no self loops or duplicated edges and all edges are oriented, and then compute the triangle count. We generate a dataset that has 1 million vertices, 2400 partitions. ComputeTriangleCount phase incurs 49GB in-memory cached RDD and 396GB total shuffle data. Results are shown in Fig. 5.12 with an average error rate of 3.6%.

5.5.2.5 Terasort

Terasort in Spark is another shuffle-heavy algorithm. There are two stages in Terasort: newAPIHadoopFile (NF) and saveAsNewAPIHadoopFile (SF). In the NF stage, input records are read from HDFS and sorted by ranges, and then the shuffle data is written to Spark Local. In the SF stage, each partition reads in the shuffle data that belongs to its range, sorts the record by key within the range and writes the output to HDFS. We take one example dataset generated by SparkBench: it has 10 billion records, with a total size of 930GB data. Results are shown in Fig. 5.13 with an average error rate of 3.9%.

Summary: For iterative algorithms, when dataset is small and cached in memory, runtime difference between HDD and SSD comes from HDFS read (write), and can be as large as 2x in LR (Fig 5.8). When dataset is large and persist on disk, runtime difference mainly comes from persist read (write) on Spark Local in each iteration, as shown in iteration phases for LR (7.0x in Fig 5.9) and PR (2.2x in Fig 5.11). For

iterative algorithms with shuffle phase and shuffle-heavy algorithms like Terasort, runtime difference between using HDD and SSD as Spark Local can be modeled as shown in subtract phases in SVM (6.2x in Fig 5.10), TC (6.5x in Fig 5.12), and Terasort (2.6x in Fig 5.13). In summary, our model enables users to quantitatively analyze and understand application runtimes on in-memory computing frameworks like Apache Spark.

5.6 Application of The Performance Model—A Case Study for Cost Optimization in Public Cloud

As reported by Broad Institute in 2017, 17 TB of new genome data is generated per day, and in total 45PB of data is managed. Moreover, according to [SLF15], with the advancement of DNA sequencing, it is estimated that 20 exabytes of genome data will be produced every year by 2025. Huge data in genome analysis requires enormous CPU, memory, and I/O resources. Consequently, private institutions may not be able to afford the cost. Public cloud providers, e.g., Google Cloud, Amazon EC2 and Microsoft Azure, offer abundant CPU and associated memory and disk I/O resources that users may request. However, to process 20 exabytes genome data in Google Cloud means about 1.6×10^{10} CPU hours in GATK4, which is about 0.53 billion dollars for CPU cost only. Moreover, users have to pay for the requested I/O resources as well. Cloud providers support different disk I/O options. While SSD offers a much higher bandwidth compared to HDD, it is charged at a much higher price (4.2x in Table 5.5). An important question naturally arises from such observations: *In a public cloud, how does one effectively find the optimal configuration to minimize cost for its required workload?*

This is not a trivial question. While a higher configuration can deliver shorter execution time, the cost per time unit is increased. On the other hand, although adopting a lower configuration guarantees lower cost per time unit, the total execution time rises. Hence, a balance needs to be discovered.

Table 5.5: Disk price in Google Cloud platform

Type	Price (per **GB/month**)
Standard provisioned space	$0.040
SSD provisioned space	$0.170

5.6.1 Cost Modeling for HDDs

With GATK4 as an example, we demonstrate that with our I/O-aware analytic model, users can quickly find the optimal hardware configuration from a large set of configurations in a public cloud to save on cost. In this section, all the results we report are genome sequencing with 500 million read pairs, as described in Section 5.2.3.

In Google Cloud, users can specify the CPU instance with a different number of virtual CPU cores. The disks are also virtualized in Google Cloud. According to their datasheet [Pla17], the virtual disk bandwidth is related to its configured size. Size and type (e.g., HDD or SSD) are the determinant factors of the disk price.

First, we create lookup tables for HDD and SSD persistent disk under different sizes. The read bandwidths for different request sizes of HDD and SSD can be found in [cds17]. Three profiling runs are executed to get the model parameters.

After getting the model, the configuration selection problem is converted to minimize a discrete multivariate function $Cost = f\ (P,\ DiskSize_{HDFS},\ DiskSize_{Spark\_Local},\ Time)$. Here P denotes the core number per node and $Time$ denotes the execution time which can be derived from our model. This optimization problem can be solved by

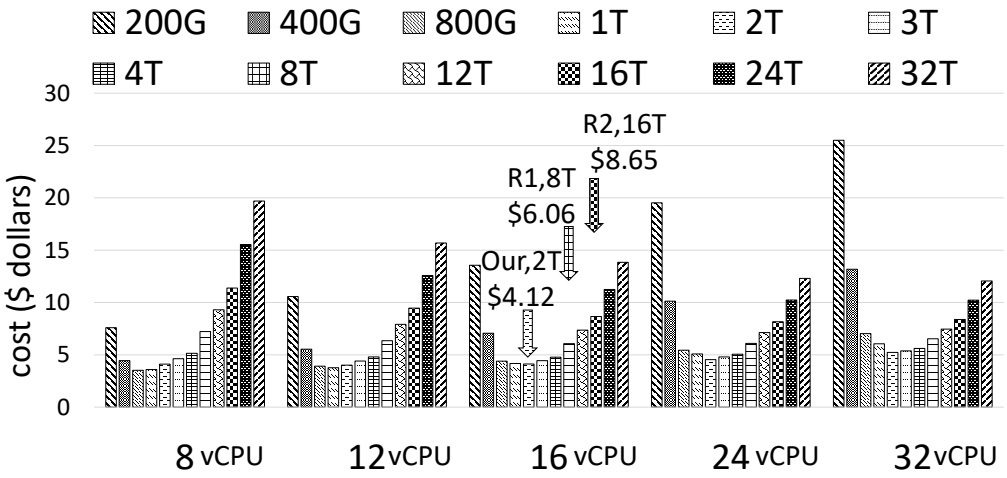
the gradient descent method. The optimal configuration that we get is the $P = 16$, $DiskSize_{HDFS} = 1\text{TB}$, $DiskSize_{Spark\_Local} = 2\text{TB}$.

In order to give readers an idea of the trend of cost function, in Fig. 5.14a and Fig. 5.14b we present the cost results under $DiskSize_{HDFS} = 1\text{TB}$ and $DiskSize_{Spark\_Local} = 2\text{TB}$. There are two recommended hardware configurations for reference: R1 [Spa17c] hardware provisioning from the Apache Spark official website, suggesting 1:2 ratio of disks to CPU cores; R2 [Clo13] hardware provisioning for Hadoop cluster from Cloudera, suggesting two hex-core machines with 12 disks (1TB), with 1:1 ratio of disks to CPU cores. If a 16 vCPU is used as a worker node, the estimated cost for R1 with 8TB disk is \$6.06, and for R2 with 16TB disk it is \$8.65. Interestingly, the estimated cost found by our model is \$4.12, which is 32% and 52% lower than R1 and R2 cost, respectively.

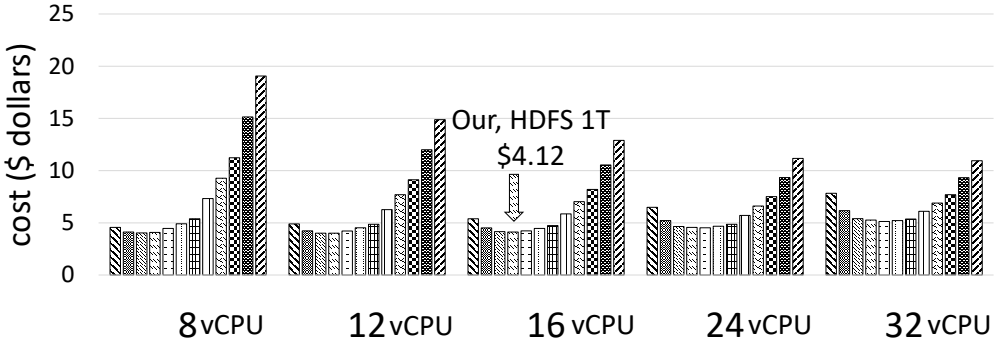
5.6.2 Model Verification on Google Cloud

True, the virtualized environment is much different than the physical one. However, as shown in Section 5.4, all of the model factors are only related to the system performance experienced at the user level, whether or not the underlying runtime is virtualized. That means the abstraction of our model is at user level—which is higher than the underlying system level. Therefore, our model can still work well in the cloud environment, and this is proved by the following experiment results.

Due to our limited Google Cloud credit, we verify our analytic model for using ten slave nodes, each with 16vCPU and 1TB as HDFS, while changing the HDD Local size. The measured runtime and predicted runtime from models are compared in Fig. 5.15. When HDD Local size increases from 200GB to 2TB, runtime decreases.



(a) Cost as Local size changes (HDFS is set to 1TB HDD).



(b) Cost as HDFS size changes (Local is set to 2TB HDD).

Figure 5.14: Cost for using different sizes of HDDs

After 2TB, runtime remains flat as expected. The average error rate is less than 4%.

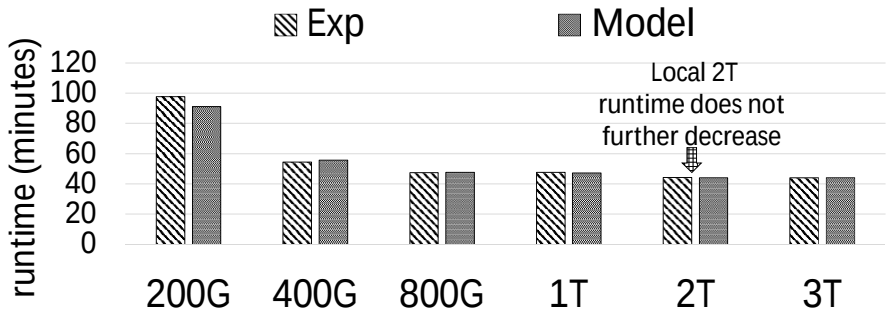


Figure 5.15: 16vCPU: Comparison of measured runtime model (exp) and predicted runtime (model) for GATK4 when using different sizes HDD as Local (HDFS is set to 1TB HDD).

5.6.3 Cost Modeling for SSDs

Fig. 5.16 shows the case where SSD is used for Spark Local, with estimated cost and runtime for different numbers of executor cores P using different sizes SSD as Local (from 20GB to 3.2TB). The optimal cost of using SSD is less ($3.75) which is another 1.1x as compared to $4.12 using HDD as Spark Local. The measured runtime of using 200GB SSD as Spark Local is 43 mins, while the estimated runtime from the model is 45 mins (error rate 4.6%). Last, considering SSD as HDFS does not bring further cost savings. We omit the details due to space constraints.

5.6.4 Modeling Results

In conclusion, using 200GB SSD as Spark Local and 1TB HDD as HDFS achieves the cost-optimal hardware configuration for 16vCPU as a worker node. The cost is $3.75, which is 38% and 57% lower than the cost of suggested configurations in R1 [Spa17c]

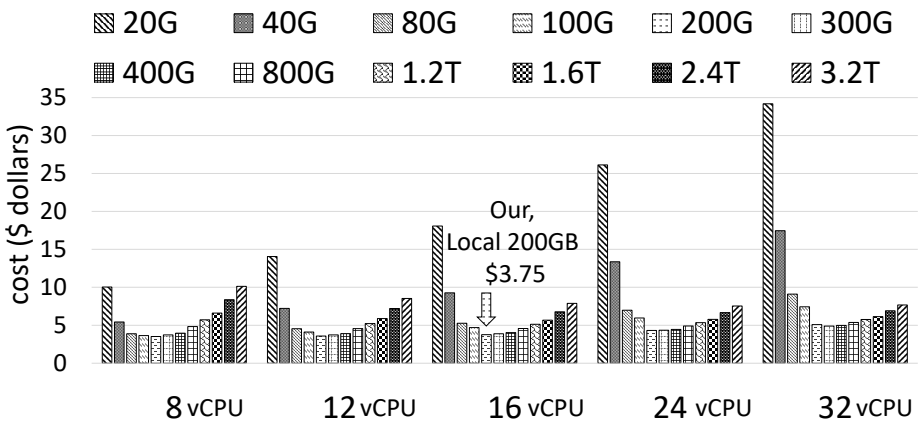


Figure 5.16: Cost for using different sizes SSD as Local (HDFS is set to 1TB HDD). and R2 [Clo13] respectively.

5.7 Related Work

5.7.1 Spark Performance Analysis and Modeling

K. Ousterhout et al. [ORR15] use blocked time analysis to study the impact of network, I/O and stragglers on Spark performance. In their paper, for the SQL workloads and the hardware setup they study, optimization on the network and I/O storage reduces the runtime by at most 2% and 19%, respectively. The conclusion on I/O part can also be explained by our model: 1) Average megabytes transferred to or from disk MB/s per node in their SQL workload is 10 MB/s (98MB/s in GATK4); 2) CPU:Disk Ratio in their cluster is 4:1 (18:1 in our cluster). By applying this number in Equation 5.1, I/O is not bottleneck in their application and cluster setup. On network part, A. Trivedi et al. [TSP16] point out that moving from a 1Gbs to a 10Gbs network reduces the Spark runtime by up to 2.5x, and the network performance still matters. Others study Spark performance from an architecture perspective [ABV16],

NUMA [WXW16, ABV16, CO16], huge page [WXW16], hyperthreading [ABV16, CO16], tuning JVM parameters and OS parameters [CO16], or from an application perspective [LTW15].

E. Gianniti et al. [GRB17] use Fluid Petri Nets to model the performance of MapReduce and Spark applications. It focuses on a scenario of the user-shared cluster, using previous execution information to study the distribution of the task time which is used for future prediction. Yet, our model can work in a much wider scenario, and the methodology we adopt is quite different than their statistical way. CherryPick [ALC17] leverages Bayesian optimization and builds the non-parametric performance model. However, it picks a cost-saving configuration from a limited number (66) of predefined cloud configurations. Studies like Ernest [VYF16] and [WK15] build analytic models to predict the Spark performance on iterative machine learning algorithms when there are more slave/worker nodes. However, in their models, the I/O impact on different data request sizes is not considered; this has a significant impact on performance, especially for the HDD case (as we studied in Section 5.3.3 and Section 5.5.2).

5.7.2 Impact of I/O on Parallel and Distributed Computing

Work in [KC14, AP15] studies how SSD and HDD impact the MapReduce performance. In [KC14] the runtime difference between SSDs and HDDs is compared: the number of SSD and HDD is matched as 1 to 11 on equivalent sequential I/O bandwidth. We do not match the number of disks as in [KC14] because the I/O bandwidth of SSDs and HDDs is not constant and varies significantly with application-requested I/O block size (as explained in Section 5.3.3). Thus, matching on sequential I/O does

not mean matching on random I/O. Other work like [ZZR16, DJ09] studies the SSD performance with its internal mechanism.

Work in [GAB15, HAL16] characterizes I/O impact for HPC clusters and proposes job scheduling algorithms to optimize the system throughput among different applications. Opass [YWZ15] analyzes how remote and imbalanced read accesses impact system performance in distributed file systems and proposes optimization to maximize data locality and access balance. I. S. Choi et al. [CYK15] leverage PCIe SSD to optimize the I/O performance of Spark. Work in [CMC16] discusses how to scale Spark in HPC clusters where a global parallel file system is used instead of local disks. In summary, to the best of our knowledge, we are the first to propose an I/O-aware analytic model to quantitatively analyze and model the impact of I/O on applications running on top of the in-memory computing framework Spark.

5.8 Conclusion

In this chapter we observe that I/O can still play a heavy role—even in the in-memory cluster computing frameworks like Apache Spark. After a quantitative analysis, we find that the performance gap is mainly caused by a large number of (random) intermediate data accesses with small data blocks to the Spark local storage, where SSDs can achieve a much more effective bandwidth than HDDs. Such Spark local storage accesses are often used to avoid recomputation of time-consuming sort in shuffle operations, or persist large RDDs that consume a large amount of memory if cached. More importantly, we propose an I/O-aware analytical model to reason the performance of Spark programs, where it brings together the effective I/O bandwidth under different data access sizes and different numbers of CPU cores, and the overlap

between the CPU computation and I/O accesses which has been often overlooked in past studies. Our proposed model can analytically explain and predict (within a 10% error rate) the runtime behavior of the production-quality genome analysis toolkit GATK4, and typical iterative algorithms that are computation-heavy, as well as typical shuffle-heavy algorithms. Finally, we demonstrate our model's usage by doing cost optimization in Google Cloud, which can quickly find the optimal hardware configuration in large exploration space and help us save 38% and 57% in cost for genome sequencing compared to two other recommended Spark cloud configurations.

CHAPTER 6

Cluster-Level Performance and Cost Modeling

In this chapter we propose the Mocha framework to optimize overall application cost on public clouds. We first present the key approach of Mocha framework in Section 6.1 that balances the throughput of CPU cores (by partially offloading kernel tasks) and FPGA (by sharing FPGA among multiple nodes) to achieve full computation resource utilization. It means that Mocha guarantees the cost efficiency for any applications as long as FPGA speedup S is larger than cost ratio CR.

According to the approach, Figure 6.1 depicts an overview of Mocha framework. By taking the user application and an instance list of the public cloud, Mocha first launches its profiling application to obtain kernel proportion r, kernel speedup S, cost ratio CR of CPU-FPGA instance and number of CPU cores P on CPU instances. The information is used as the input of our cost model to achieve cost efficiency by determining the system configuration (e.g., number and type of instances to be launched, the percentage of total kernel tasks to be offloaded to FPGAs) to generate the Mocha configuration in Section 6.2.

By taking the generated system configuration, Mocha runtime (Section 6.3) creates clients in the user application on each instance that needs to leverage the FPGA accelerator. The runtime, shown in the right part of Figure 6.1, includes client and node accelerator manager (NAM). The client is launched on CPU instances to offload

partial tasks to the instance with an FPGA accelerator via network. NAM is launched on CPU-FPGA instances to receive tasks from multiple clients and schedule tasks on the accelerator.

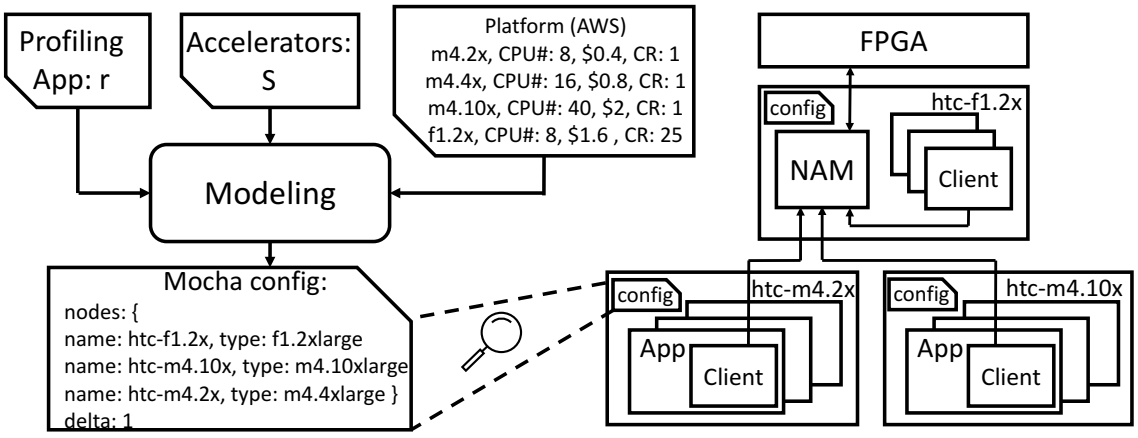


Figure 6.1: Mocha framework overview

6.1 CPU-FPGA Integration and Cost Modeling

In Chapter 4, we summarize two cases when the computation throughput of CPU and FPGA are not equal to each other: Case A happens when number of CPU cores P is smaller than *Matching Core Number* $\widetilde{P}$ and FPGA is underutilized; Case B happens when number of CPU cores P is larger than *Matching Core Number* $\widetilde{P}$ and CPU is underutilized. In this section, we explain methods to either improve FPGA utilization or CPU utilization for Case A and Case B respectively.

FPGA Utilization Improvement: For applications of Case A like HTC, one opportunity to reduce I is to improve FPGA utilization by sharing the FPGA among multiple nodes through the network. In other words, when $\widetilde{P}$ is larger than the maximum number of CPU cores on a single node P_0 in a datacenter, we can

launch more CPU node(s) to request the same FPGA. In Equation 4.6, if we set $P = \widetilde{P} = (\frac{(1-r)\times S}{r} + 1)$, we can achieve optimized cost efficiency I' as:

$$\begin{aligned} I' &= (1 - r + \frac{r}{S})(1 + \frac{CR}{\widetilde{P}}) = (1 - r + \frac{r}{S}) + (1 - r + \frac{r}{S}) \times \frac{CR}{\widetilde{P}} \\ &= (1 - r + \frac{r}{S}) + (\frac{\widetilde{P} \times r}{S}) \times \frac{CR}{\widetilde{P}} = 1 - r + \frac{r \times (CR + 1)}{S} \end{aligned} \tag{6.1}$$

We can see from the Equation that when $S > CR + 1$, I' is guaranteed to be smaller than 1. For HTC on AWS, if we set $\widetilde{P}$ as 64, I' is 0.86 <1, as opposed to I as 2.56× in straightforward integration.

CPU Utilization Improvement: For Mutect2 on Huawei Cloud as shown in Table 4.2, PairHMM kernel dominates 89% of overall execution time so its matching core number $\widetilde{P} = \frac{(1-89\%)}{89\%} \times 43 + 1 = 6$. As there are in total 32 CPU cores on the Huawei FP1 instance, CPU is severely underutilized and I is 1.14× over pure CPU solutions.

For applications of Case B like Mutect2, one opportunity to reduce I is to improve CPU utilization by a partial task offloading policy. As demonstrated by Figure 4.1c, for core 1 in batch i, instead of waiting extra gap cycles on the FPGA resource, core 1 can directly work on the shadow part (though using more time) to avoid waste of CPU resource. Intuitively, the most efficient way to utilize FPGA and CPU in this case is to schedule part of the kernel tasks (M_1) on FPGA and the other tasks (M_2) on CPU, as shown in Figure 6.2. Thus, the overall application runtime T'_{1b} and tasks number M_1, M_2 follow equations as:

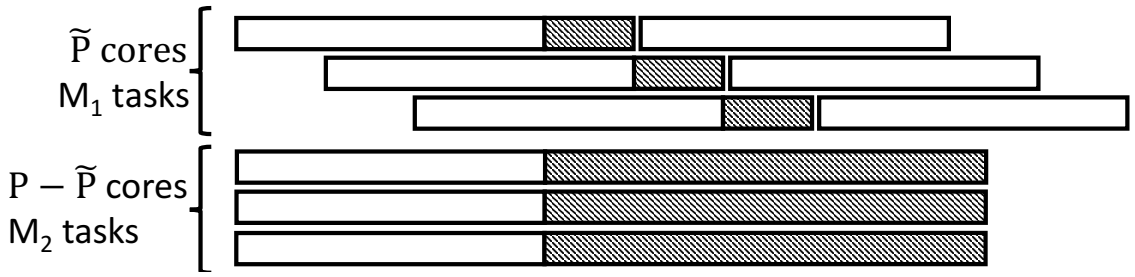


Figure 6.2: Partial task offloading

$$T'_{1b} = \frac{M_1}{\widetilde{P} - 1 + S} \times t = \frac{M_2}{P - \widetilde{P}} \times t, \qquad M_1 + M_2 = M, \tag{6.2}$$

We can rewrite Equation 6.2 to $T'_{1b} = \frac{M1+M2}{P-1+S} \times t = \frac{M}{P-1+S} \times t$. As a result, the optimized cost efficiency $I' = T'_{1b} \times (P + CR) \times c \times /C_0 = \frac{P+CR}{P-1+S}$, and $I' < 1$ when *CR <S -1*. This is achieved when we offload $\delta = \frac{M_1}{M} = \frac{\widetilde{P}-1+S}{P-1+S}$ of total kernel tasks on FPGA and schedule $1 - \delta = \frac{P-\widetilde{P}}{P-1+S}$ of total kernel tasks on CPU. We define δ as offloading task ratio. For Mutect2 on Huawei, if we set $\delta = \frac{6-1+43}{32-1+43}$=0.65, I' is 0.74 <1, as opposed to I as 1.14× in the straightforward integration.

To sum things up, it doesn't matter that $\widetilde{P}$ is larger or smaller than the number of CPU cores in a single node P_0, we have *Optimized Cost Efficiency Index* I' as:

$$I' = \begin{cases} 1 - r + \frac{r \times (CR+1)}{S} & \text{if } \widetilde{P} > P_0, \text{ set } P = \widetilde{P} \text{ on multi-nodes} \\ \frac{P+CR}{P-1+S} = \frac{P_0+CR}{P_0-1+S} & \text{if } \widetilde{P} < P_0, \text{ set } \delta = \dfrac{\widetilde{P} - 1 + S}{P - 1 + S}, \; P = P_0 \end{cases} \tag{6.3}$$

Consequently, *as long as* $S - 1 > CR$, I' *is guaranteed to be smaller than 1 in both cases*. This modeling gives quantitative support of CPU-FPGA integration for Mocha to set up a cluster with appropriate CPU-FPGA nodes and pure CPU nodes to achieve full resource utilization within the cluster.

6.2 Cost Model Implementation

After profiling the application and FPGA accelerator to get r, S, Mocha calculates $\widetilde{P}$ as described in Equation 6.3. Specifically, with r, s and the platform information which lists available CPU instances and number of CPU cores within an instance, we can obtain the proper number and type of CPU nodes we should launch to optimize the cost efficiency. For example, on Amazon EC2, `m4` series instances have `m4.x`, `m4.2x`, `m4.4x`, `m4.10x` and `m4.16x` which have 4, 8, 16, 40, 64 cores respectively. According to Equation 6.3, if $\widetilde{P}$ is larger than P_0, we set a cluster with in total $\widetilde{P}$ cores. For example, for HTC on AWS EC2, $\widetilde{P}$ is 64, which is larger than 8. We first select `f1.2x` instance, and then select other CPU nodes to get the remaining 64-8 = 56 CPU cores. We iteratively pick up the largest possible instance until all the remaining cores are allocated. Thus, we first pick up `m4.10x` which has 40 cores, and update the remaining cores as 56-40 = 16. Then we pick up `m4.4x` and the number of remaining cores reaches zero. As a result, three instances, including `f1.2xlarge`, `m4.10xlarge`, and `m4.4xlarge` with 8, 40, 16 cores, are selected. If $\widetilde{P}$ is smaller than P_0 cores, we use only one CPU-FPGA instance, and set δ, offloading task ratio accordingly. For example, for Mutect2 on AWS EC2, $\widetilde{P}$ is 6, we can simply select `f1.2xlarge` and calculate $\delta = 95\%$ based on Equation 6.3.

According to the determined system configuration, Mocha launches new instances

and broadcasts the necessary information to them. For example, Figure 6.3 shows how we launch instances on Amazon EC2 by using AWS Command Line Interface (CLI). In order to have a low-latency and high-throughput network among multiple nodes in AWS EC2, Mocha first creates an AWS EC2 placement group [Ama19a] and places all instances in the same group. In this case, all instances within a group have a network performance as high as 10 Gb/s (`m4.2x` and `m4.x` have 5 Gb/s network bandwidth as node limit). After all nodes are created, the IP address of each node is updated in the configuration file as (shown in Figure 6.4) and the configuration file is broadcast to each node. On the Huawei Cloud, all general computing instances have a 6 Gb/s network bandwidth.

```
1 #create a placement group called "htc-cluster"
2 aws ec2 create-placement-group --group-name htc-cluster --strategy
      cluster
3 #launch nodes in the placement group
4 aws ec2 run-instances --image-id amiId --count 1 --instance-type
      f1.2xlarge --placement GroupName="htc-cluster"
5 aws ec2 run-instances --image-id amiId --count 1 --instance-type
      m4.10xlarge --placement GroupName="htc-cluster"
```

Figure 6.3: Mocha launches new nodes within a placement group.

6.3 Mocha Runtime

After the cluster has been launched, Mocha runtime starts executing the application. In Mocha runtime, there are two major components: CPU client and node accelerator manager (NAM). The CPU client is launched on all instances to communicate with the NAM for data sharing as well as task offloading to the FPGA accelerator (locally and

```
{
    "nodes": [// nodes details after creation
    {"name": "htc-f1.2x", "type":"f1.2xlarge", "ip": "10.0.0.1"},
    {"name": "htc-m4.10x", "type": "m4.10xlarge"", "ip": "10.0.0.2"},
    {"name": "htc-m4.2x", "type": "m4.4xlarge", "ip": "10.0.0.3"}
    ],
    "nam_ip": "10.0.0.1", // IP of the CPU-FPGA instance
    "delta": 1// percentage of tasks offloaded to FPGA
}
```

Figure 6.4: Configuration file.

remotely). The NAM in Mocha runtime is adapted from the Blaze [HWY16, UCL16] node manager, an open source framework that enables FPGA accelerators as a service (FaaS) by abstracting multiple physical FPGA accelerators as a single logic accelerator. Mocha enhances the NAM by adding a feature that can divide a powerful FPGA accelerator into multiple logic accelerators so that the FPGA can always be fully subscribed.

In the rest of this section, we explain the communication mechanism among the CPU client, NAM, and the accelerator.

The CPU Client: A client is launched in the user application program by taking system configuration broadcast from the Mocha master. In particular, it refers to the δ from the system configuration and δ determines the ratio of offloading the task to FPGA. As shown in Figure 6.5, if δ is not 1, a random number between 0 and 1 is generated and compared to δ. If the random number is larger than δ, kernel computation falls back to use CPU by calling `compute()`. Otherwise, the accelerator client prepares input data and calls `start()` to connect to NAM to send task. After task is finished, the output data is read from the client to `result`.

```
1 std::string nam_ip = "10.0.0.1"; //ip of the CPU-FPGA node
2 // Create a Client with name "PairHMM", 2 input, 1 output
3 Client_ptr client(new Client("PairHMM", 2, 1, nam_ip));
4 if (delta != 1 && ((double)rand()/(double)RAND_MAX > delta)){
5     compute(); // run on CPU
6 }
7 else{ // run on FPGA
8     // client setup input blocks from user program
9     client->setupInput();
10    // client connects to NAM, call fpga and receive output
11    client->start();
12    client->getOutput(result);
13 }
```

Figure 6.5: Code snippet of client in an user program

Communication between CPU client and NAM: As shown in Figure 6.6a, the CPU client first connects to the instance with FPGA accelerator according to the system configuration broadcast by the Mocha master. The client sends message `ACCREQUEST` to NAM to ask for accelerator with accelerator ID "PairHMM." If NAM has loaded the requested accelerator bitstream on FPGA, it sends `ACCGRANT` to acknowledge the client to send metadata and the input data block(s) of the tasks in `ACCDATA`. After all input data blocks are ready in NAM, NAM enqueues the task with input blocks to a task queue. After the task is finished, NAM sends back `ACCFINISH` with the metadata of output data block(s) to the client.

In Figure 6.6a, solid lines are message data with metadata information of accelerator and input/output blocks. The single-headed dashed lines are accelerator data transfer processes between a client and NAM. The tail side in a single-headed dashed line creates `SHARED DataBlock` by using metadata of `OWNED DataBlock` created at

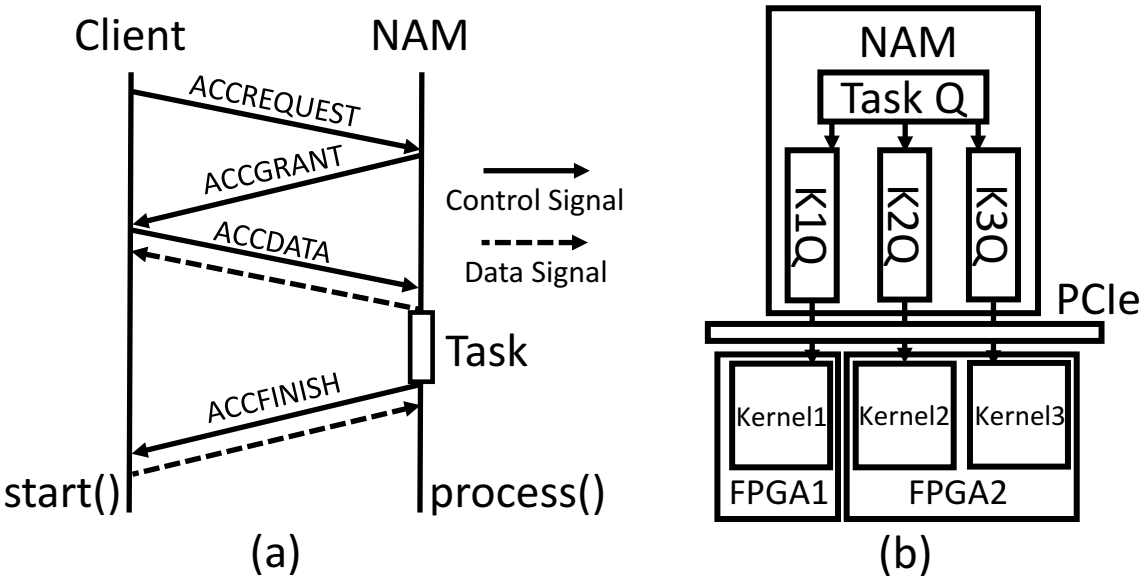


Figure 6.6: (a) The communication protocol between Client and Node Accelerator Manager (NAM), (b) NAM enhanced by Mocha

the arrowhead. The metadata of input/output data blocks is shown in Figure 6.7 in protocol buffers format [Goo19]. When a client and NAM are on the same node, data are shared between directly through memory mapped files. In this case, metadata simply includes the memory mapped file path which lets `SHARED DataBlock` access directly. On the other hand, when a client and NAM are on different nodes, data are shared through the network by using Boost.Asio [Koh16] library. Thus, the IP address and port at which the `OWNED DataBlock` is listening are needed for `SHARED DataBlock` to connect to and read from.

Communication between NAM and accelerator: In the original Blaze, the task queue directly dispatches tasks to platform queues. Each platform queue is associated with a physical FPGA device. This leads to a potential underutilized problem when the task granularity size is too small to saturate the computing power

```
message DataMsg {
  // data size
  optional int32 element_size = 1;
  optional int32 num_elements = 2;
  optional int64 scalar_value = 3;
  // same node: memory-mapped file name
  optional string file_path = 4;
  // diff nodes: ip, port
  optional string data_ip = 5;
  optional int32 data_port = 6;
}
```

Figure 6.7: Protobuf specifying metadata of data blocks.

of a full FPGA. This problem can be addressed by partially reconfiguring a large powerful FPGA as multiple small kernels. In addition, the FPGA accelerator design philosophy prefers multiple smaller kernels rather than one large kernel since it helps improve timing. Mocha enhances the NAM by supporting multiple kernels in a single FPGA. As shown in Figure 6.6b, each kernel queue is associated with an FPGA kernel instead of an FPGA device. The NAM always schedules a task to the kernel queue that has the least number of tasks to balance the load among all the kernels.

6.4 Case Study: Accelerate Genome Variant Calling on Public Clouds

In GATK, HTC and Mutect2 are very time-consuming applications because they both use a dynamic programming (DP) algorithm, PairHMM [DEK98], to calculate the relation probability of two sequences. PairHMM has high time complexity and heavy floating-point operations. It typically dominates 39% and 89% of overall execution time in HTC and Mutect2 respectively in the current GATK implementation where Intel AVX intrinsics are used in the kernel. In this section, the PairHMM accelerators developed by FCS (Falcon Computing Solutions [Com18]) are evaluated on both AWS EC2, and Huawei Cloud and compared with the other FPGA and GPU PairHMM accelerators in Section 6.4.1. The accelerator has a speedup S larger than CR+1, which guarantees improvement of cost efficiency of CPU-FPGA solutions after Mocha framework is used. In Section 6.4.2 we give a detailed evaluation of how Mocha framework improves the overall application performance and cost efficiency of CPU-FPGA solutions for HTC and Mutect2 on AWS EC2 and Huawei Cloud. Compared to a straightforward CPU-FPGA solution in Blaze [HWY16], Mocha saves cost by 2.82x with only 4.9% performance degradation for HTC, and saves cost by 1.05x with 1.05x performance improvement for Mutect2 on Amazon EC2. Similarly, on Huawei Cloud, Mocha saves cost by 1.22x with only 2.8% performance degradation for HTC and saves cost by 1.52x with 1.52x performance improvement for Mutect2. For each application on both platforms, the optimized cost efficiency I' is smaller than 1, which implies that the CPU-FPGA solution is more cost efficient than the pure CPU solution by using a Mocha framework.

6.4.1 Evaluation of PairHMM Accelerator

Table 6.1 shows the resource utilization of FCS PairHMM accelerator on the AWS F1 and Huawei Cloud FP1 FPGAs. Both platforms use Xilinx Ultrascale+ VU9P chips. However, we find that the user resource budgets are different. This is because the AWS F1 platform has a larger static region than the Huawei Cloud FP1 platform. Therefore, we can only fit 184 PEs on AWS F1 but can fit 200 PEs on Huawei Cloud FP1. Although there is still DSP resource idling, we cannot further improve the number of PEs because the frequency degrades significantly due to routing congestion.

Table 6.1: The resource utilization of FCS PairHMM accelerator on AWS F1 and Huawei Cloud FP1 FPGAs. The number on the right side of each cell is the available resource for users excluding the platform static region.

	PE#	LUTs	LUTRAMs	Flip-Flops	BRAMs	URAMs	DSPs	Fmax
AWS F1	184	560k/946k (59%)	31k/562k (6%)	802k/2M (40%)	1082/1.8k (58%)	164/917 (18%)	4612/6831 (68%)	186 MHz
Huawei Cloud FP1	200	607k/978k (62%)	31k/562k (6%)	870k/2.1M (42%)	1174/1.8k (63%)	172/960 (18%)	5012/6830 (73%)	191 MHz

A common metric to evaluate the performance of PairHMM is giga cell updates per second (GCUPs). Equation 6.4 shows how to compute it. Both the average and peak GCUPs of FCS designs are measured by running through all the datasets listed in the second column in Table 6.5 (The first three samples are whole exome sequence, NA12878Garvan is a whole genome sequence, and TCRBOA1 is a tumor sequence).

$$\text{GCUPs} = \frac{\text{read\_length} \times \text{haplotype\_length}}{\text{PairHMM\_time}} \tag{6.4}$$

The design beats the best CPU performance shown in the third row by around 40x. To the best of our knowledge, the PairHMM accelerator demonstrates the best GCUPs on FPGA instances in public clouds.

Table 6.2: Comparison of performance and cost efficiency among state-of-the-arts and FCS PairHMM accelerator.

Implementation							Performance		Cost Efficiency		
Tech	Make	Model	Parallelism	Part#	Freq	Code	Avg GCUPs	Peak GCUPs	*CR*	Avg *S*	$\frac{CR}{S-1}$
CPU [RPA16]	Intel	Broadwell	singe-core AVX	E5-2686 v4	2.3GHz	GKL	0.676	0.699	NA	NA	NA
GPU [WXC17]	Nvidia	Volta	5120 CUDA cores	V100	1.2GHz	CUDA	61.1	195	53 [Ama19b]	90	0.59
FPGA (FCS [Com18])	Xilinx	aws-vu9p-f1	184 PEs	XCVU9P	186MHz	SDAccel	26.9	32	25 [Ama19b]	40	0.64
FPGA (FCS [Com18])	Xilinx	huawei-vu9p-fp1	200 PEs	XCVU9P	191MHz	SDAccel	29.4	35.8	23 [Hua19]	43	0.54

Performance of the best GPU PairHMM accelerator [WXC17] is shown in the fourth row. The authors open sourced the PairHMM GPU implementation [Wan19], and we measured its GCUPs by testing it on the AWS EC2 `p3.2x` instance. The instance has one Nvidia V100 GPU, 8 vCPU, and it is priced at $3.06/hr [Ama19b]. For this GPU, CR = ($3.06/$0.4*8-8) = 53. There exists a 2x gap on GCUPs between FCS design (29.8) and the GPU accelerator(61.1). However, if we consider cost efficiency, as shown in Equation 6.3, when r, P_0 are the same, I' is determined by both CR and S. Here we use $\frac{CR}{S-1}$ as a metric to compare the cost efficiency of different accelerators in a public cloud. As shown in the last column of Table 6.2, among the GPU accelerator, FPGA accelerator on AWS EC2 and Huawei Cloud, the FPGA accelerator on Huawei Cloud achieves the highest cost efficiency.

When multiple nodes are launched in a cluster in the Mocha framework, data blocks are shared through the network. As S is the end-to-end FPGA accelerator speedup, in addition to PCIe communication overhead, network latency also needs to be taken into consideration. For the PairHMM kernel on AWS EC2 and Huawei Cloud, we give one representative task breakdown as shown in Table 6.3. Here, PCIe bandwidth is assumed as 6GB/s and network bandwidth is 6Gb/s.

After including network latency, S changes from 39.8 to 39.7, which means network does not have a lot of effect on S for clients that are not on the same node as NAM. In general, for other applications, a similar analysis is needed to perform to get an

Table 6.3: Time breakdown (secs) of a representative PairHMM task with 3MB input and 40KB output.

CPU task	FPGA kernel	PCIe	network
88	2.29	0.0005	0.004

updated S for remote clients and an updated *Matching Core Number* in the modeling phase.

6.4.2 Evaluation of FCS GATK Acceleration Solution

To demonstrate how Mocha improves the overall application cost for HTC and Mutect2 on different platforms, we run these two applications on AWS EC2 and Huawei Cloud respectively for the dataset as specified in Table 6.5. We have two baselines: pure CPU solution and straightforward CPU-FPGA integration solution in Blaze. For each dataset, we measure the time by running it 10 times and taking the average latency.

As shown in detail in Section 6.2, on each platform, for each application, Mocha generates system configuration file which specifies the number and type of CPU nodes we should launch to fully utilize CPU and FPGA. To do a fair comparison of Mocha with two baselines, we launch instances for each baseline with *same number of CPU cores* as $\widetilde{P}$ in Mocha modeling. We summarize instances for pure CPU, Blaze and Mocha in Table 6.4.

Table 6.4: Mocha system configuration for HTC and Mutect2 on AWS EC2 and Huawei Cloud. For example, eight `f1.2x:8` means we launch eight `f1.2x` instances, each with 8 CPU cores.

Application	AWS EC2				Huawei Cloud			
	P	pure CPU	Blaze [HWY16]	Mocha	P	pure CPU	Blaze [HWY16]	Mocha
HTC	64	`m4.16x` :64	eight `f1.2x`: 8	`f1.2x`: 8, `m4.10x`: 48, `m4.2x`: 8	64	`s2.16x`: 64	two `fp.1c`: 32	`fp.1c`:32, `s2.8x`:32
Mutect2	6	`f1.2x`:8	`f1.2x`: 8	`f1.2x`: 8	6	`fp.1c`: 32	`fp.1c`: 32	`fp.1c`: 32

Table 6.5: Comparison of performance and cost of pure CPU solution, Blaze and Mocha.

Application	SampleID	AWS EC2						Huawei Cloud					
		pure CPU		Blaze [HWY16]		Mocha		pure CPU		Blaze [HWY16]		Mocha	
		Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
HTC	NA12878 [Ill19]	578	$0.51	362	$1.33	386	$0.48	577	$0.55	361	$0.58	378	$0.49
		1x	1x	1.60x	2.61x	1.50x	0.94x	1x	1x	1.60x	1.07x	1.53x	0.89x
HTC	NA12891 [Ill19]	592	$0.53	381	$1.40	404	$0.50	591	$0.57	369	$0.60	373	$0.48
		1x	1x	1.55x	2.64x	1.47x	0.93x	1x	1x	1.60x	1.06x	1.58x	0.84x
HTC	NA12892 [Ill19]	549	$0.49	352	$1.29	374	$0.46	542	$0.52	349	$0.57	356	$0.46
		1x	1x	1.56x	2.63x	1.47x	0.94x	1x	1x	1.55x	1.09x	1.52x	0.88x
HTC	NA12878Garvan [Sta16]	2767	$2.46	1731	$6.35	1767	$2.18	2709	$2.61	1710	$2.79	1778	$2.30
		1x	1x	1.60x	2.58x	1.57x	0.89x	1x	1x	1.58x	1.07x	1.52x	0.88x
HTC	**Average**	**1x**	**1x**	**1.58x**	**2.62x**	**1.50x**	**0.93x**	**1x**	**1x**	**1.58x**	**1.07x**	**1.54x**	**0.87x**
Mutect2	**TCRBOA1 [BCM18]**	16784	$1.86	3047	$1.40	2885	$1.32	4196	1$.90	2807	$2.21	1850	$1.45
		1x	**1x**	**5.51x**	**0.75x**	**5.82x**	**0.7x**	**1x**	**1x**	**1.49x**	**1.16x**	**2.27x**	**0.76x**

Table 6.5 gives a comparison of performance in seconds and cost in dollars of pure CPU solution, Blaze, and Mocha on four sequences for HTC and one sequence for Mutect2 in AWS and Huawei Cloud. We also give normalized performance improvement and application cost of Blaze and Mocha as compared to the pure CPU solution. The average normalized value of HTC and Mutect2 are shown in bold font to highlight the performance and cost difference of pure CPU, Blaze and Mocha. As shown in the row of averaged normlized value for HTC, Blaze incurs 2.62x more extra cost than pure CPU. This corresponds to I in Table 4.1, where inefficiency comes from FPGA underutilization. Mocha improves cost efficiency by borrowing cores from CPU nodes to send tasks to a single `f1.2x` instance to do acceleration. As a result, Mocha can spend less dollars per hour than Blaze, while achieving 1.5x performance improvement than pure CPU, which has a 5% degradation compared to 1.58x fpr Blaze. Equation 6.3 gives an estimated $I' = 0.86$, and experimental result gives 0.93. In other words, our modeling accuracy is as high as 92%. For Mutect2, according to Table 4.1, $\widetilde{P}$ is 6, very close to 8 CPU cores on `f1.2x`, which implies a narrow optimization space for Mocha. In this case, by partially offloading kernel tasks from FPGA to CPU, Mocha further improves the performance compared to Blaze

by $\frac{5.82}{5.51}$=1.06x. As in Mutect2, all three solutions use the same instance, and they all spend the same amount of dollar per hour. The 1.06x performance improvement naturally translates to a 1.06x cost savings. Similar analysis can be performed on Huawei Cloud. As opposed to Blaze, Mocha improves cost efficiency of HTC by $\frac{1.07}{0.87}$ = 1.22x with a 3% performance degradation. For Mutect2, Mocha improves performance by $\frac{2.27}{1.49}$= 1.52x, and that translates to 1.52x cost savings.

6.5 Related Work

Cost Optimization on Cloud Systems. There is a lot of existing work that discusses optimizing the cost saving on cloud systems on aspects other than accelerators. Authors in [MH11b] presented an approach to minimize the cost given a hard job deadline. HCloud focused on cost saving on reserved and on-demand hybrid cloud systems with quality of service (QoS) constraints [DK16]. Paris reduces cost by accurately predicting performance of users across cloud providers [YHG17]. Tributary targeted the cost savings on spot instances [HCT18]. Selecta [KLK18] and Doppio [ZRF18b] both attempted to reduce storage cost while maintaining performance of workload.

FPGA Sharing on Cloud Systems. In past few years, researchers have devoted a lot of effort to integrating FPGAs into the current cloud computing environment by providing vitalization infrastructures for FPGAs. Authors of [WZH16] presented a framework to scatter and gather data on FPGAs in a MapReduce manner. Iordache et al. [IPS16] proposed a notion called FPGA Groups to share FPGAs among multiple tenants who wish to use the same circuit design. A similar model called Blaze [HWY16] was proposed to reduce the programming efforts of the FPGA groups.

Authors of [TLF17] built a framework that can create and manage FPGA clusters on cloud systems based on OpenStack.

PairHMM Accelerator Designs on FPGAs. Previous PairHMM accelerator designs on FPGAs mostly used a systolic array architecture [RPA16, HMR17, IO16, PRA16, RSA15]. The systolic array architecture only performs well on Arria10 FPGAs with the help of hardened DSP blocks [RPA16, HMR17]. On Xilinx platforms, the PairHMM design is either bound by LUTs resource [PRA16, IO16, BET17] or has relatively low frequency [RSA15].

6.6 Conclusion

There exists a computation throughput mismatch problem between the CPU and FPGA for many applications. On current public clouds such as AWS and Huawei Cloud, which leads to extra out-of-pocket costs for the CPU-FPGA integration solution over a pure CPU solution. To address this problem, We propose a framework called Mocha that enables FPGA sharing among multiple nodes through network and partial task offloading policy for CPUs. Mocha guarantees that the cost efficiency of a CPU-FPGA solution is higher than the pure CPU solution as long as the FPGA kernel speedup is higher than the cost ratio. We present a performance comparison of FCS accelerator and provide model-driven cost optimization case studies for Genome Variant Calling applications, HTC and Mutect2, in two public cloud platforms—Amazon EC2 and Huawei Cloud. On AWS, adopting Mocha gives a 2.82x cost saving for HTC, 1.06x for Mutect2, and on Huawei Cloud it gives 1.22x, 1.52x cost savings respectively.

CHAPTER 7

Cost Optimization with Composable Instances in Public Cloud

In Chapter 6, we have proposed Mocha framework that enables FPGA sharing among multiple nodes through network and partial task offloading policy for CPUs. Mocha framework creates composable instances where CPUs and FPGA throuoghput are balanced. These composable instances achieve the best cost efficiency in CPU-FPGA integration by fully utilizing CPUs and FPGA resources for a given application with certain kernel ratio r and accelerator speedup S. In this chapter we will discuss the application of modeling performance and cost at the public cloud where composable instances are enabled—that is, optimize the overall cost of running whole genome processing given certain deadline constraints. In a public cloud, the primary concern is how to run an application with the least out-of-pocket cost; i.e., that is, how to choose instances and schedule genome pipeline stages to achieve the least cost given different deadline constraints? We first show in Section 7.1 how we model this question as a MILP problem. In Section 7.2, we present the experiment results on AWS EC2 and the instance choices for each stage that achieve the optimal cost given certain deadlines.

7.1 Modeling

In a public cloud, we consider that there are multiple CPU instances. Different instances have different CPU types, as shown in Table 7.1. In each CPU type series, different instances have a different number of CPU cores, memory sizes and prices [Ama19b], as shown in Table 7.2. For each instance, there is a different execution time for genome pipeline applications. We can schedule genome pipeline stages on different instances, upload and download intermediate data through the AWS S3 bucket to achieve optimal cost while meeting the deadline targets. In the following sections we abstract away these factors as parameters and explain how we formulate the scheduling problem by introducing the variables, objective function and constraints.

Table 7.1: Amazon EC2 instances series and CPU type.

instance	CPU
m4	Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
m5	Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz
c5	Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz
f1	Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz

Table 7.2: Amazon EC2 instances, number of CPU cores, memory sizes and on-demand prices (dollars per hour).

	m5.large	m4.x	m5.x	c5.2x	m4.2x	m5.2x	f1.2x	c5.4x	m4.4x	m5.4x	m4.10x	f1.16x
# cores	2	4	4	8	8	8	8	16	16	16	40	64
memory (GB)	8	16	16	16	32	32	122	32	64	64	160	976
price ($/hr)	0.096	0.2	0.192	0.34	0.4	0.384	1.65	0.68	0.8	0.768	2	13.2

7.1.1 Input Parameters

The following parameters need to be provided as inputs in the optimization problem:

DDL: Deadline to finish whole genome pipeline processing.

I: Set of different types of compute instances.

M: Maximum number of instances for each type of instance.

N: Set of whole genomes to be processed.

T: Set of stages in genome pipeline. For genome pipeline, stage t depends on stage $t-1$.

p_i: On-demand price of instance $i \in I$.

setup: Setup time of an instance. This time is used to install necessary tools and packages.

$stage_{i,t}$: Runtime of instance i for stage t. The value is an integer.

$upload_{i,t}$: Upload time of intermediate data for stage t on instance i. The value is an integer.

$download_{i,t}$: Download time of intermediate data for stage t on instance i. The value is an integer.

7.1.2 Variables

We introduce the following variables in the optimization problem:

C: Total out-of-pocket expense of running whole genomes in the public cloud.

$g_{i,m}$: Total running time of type i, m-th instance. The value is an integer.

$iv_{i,m}$: A flag indicating whether type i, m-th instance is launched ($iv_{i,m} = 0$) or not ($iv_{i,m} = 1$). If the instance has not been scheduled with any stages for any genomes, it is considered not launched. The value is binary.

$A_{s,t,i,m}$: Allocation variable indicating genome s, stage t is allocated to type i, m-th instance. The value is binary.

$A_{s_0t_0,s_0t_1,i_0m_0,i_1m_1}$: Allocation variable indicating task edge of genome s, stage t_0 to stage t_1 is mapped to physical edge of type i_0, m_0-th instance and type i_1, m_1-th instance. The value is binary.

$S_{s,t}$: Computed start time of genome s, stage t. The value is an integer.

$E_{s,t}$: Computed end time of genome s, stage t. The value is an integer.

$O_{s_0t_0,s_1t_1}$: A flag indicating whether two tasks (genome s_0, stage t_0 and genome s_1, stage t_1) overlap. The value is binary.

7.1.3 Objective Function

The objective function defines the optimal solution target. The optimal solution ensures executing genome pipeline processing on the given public cloud using the least amount of out-of-pocket cost while meeting a given deadline constraint. Therefore, the objective function is:

$$\min C \tag{7.1}$$

7.1.4 Constraints

The optimization problem needs to first consider the out-of-pocket cost constraints as cost is the optimization target. Then, scheduling needs to consider the hardware constraints including that all tasks are allocated to exactly one of the instances, and if two tasks are scheduled on the same instance, they can not overlap one another. Last, scheduling needs to meet the deadline constraints.

To account for cost constraints, we add the following set of constraints:

The first constraint is the definition of C, the total out-of-pocket cost. It is the sum of the cost of all the instances. For each instance, cost is the multiplication of price per hour p_i and total running time $g_{i,m}$.

$$C = \sum_{i \in I, m \in M} p_i \times g_{i,m} \tag{7.2}$$

To capture the total running time of type i, m-th instance $g_{i,m}$, we should add up the setup time, stage run time, upload time (if the subsequent stage is scheduled onto a different machine) and download time (if the previous stage is scheduled onto a different machine). Setup time is only counted in if an instance is launched ($iv_{i,m} = 0$). Stage run time is the time to execute stages that are assigned on the instance. Upload time is the total time spent if a stage is scheduled on the instance and the subsequent stage is scheduled on a different instance. The condition of different instances are characterized as $i_0 \neq i_1$ OR $m_0 \neq m_1$. Similarly, download time is the total time spent if a stage is scheduled on the instance and the previous stage is scheduled on a different instance. Therefore, we add the following constraint:

$$\begin{aligned} g_{i,m} = (1 - iv_{i,m}) \times setup + \sum_{s \in N, t \in T} A_{s,t,i,m} \times stage_{i,t} \\ + \sum_{s_0 \in N, t_0 \in \mathrm{T} \setminus \text{ last stage}, t_1 = t_0 + 1, i_0 = i, m_0 = m, i_0 \neq i_1 \vee m_0 \neq m_1} A_{s_0 t_0, s_0 t_1, i_0 m_0, i_1 m_1} \times upload_{i_0,t} \\ + \sum_{s_0 \in N, t_1 \in \mathrm{T} \setminus \text{0th stage}, t_1 = t_0 + 1, i_1 = i, m_1 = m, i_0 \neq i_1 \vee m_0 \neq m_1} A_{s_0 t_0, s_0 t_1, i_0 m_0, i_1 m_1} \times download_{i_1,t} \end{aligned}$$

To capture an invalid variable $iv_{i,m}$ of type i, m-th instance, that is, $iv_{i,m}$ is only 1 when all $A_{s,t,i,m}$ are 0. If any of the stage is mapped to the instance, that is, one of $A_{s,t,i,m}$ is 1, then $iv_{i,m} = 0$:

$$iv_{i,m} + \sum_{s \in N, t \in T} A_{s,t,i,m} \geq 1 \tag{7.3}$$

$$iv_{i,m} \leq 1 - A_{s,t,i,m}, \forall s \in N, t \in T \tag{7.4}$$

To account for hardware constraints, we add the following set of constraints:

To capture that all the tasks are allocated to exactly one of the instances, that is , $A_{s,t,i,m}$ can only be 1 for 1 instance, 1 machine, we have the following constraint:

$$\sum_{i \in I, m \in M} A_{s,t,i,m} = 1 \tag{7.5}$$

To capture that for each task edge, there is one mapping for the edge to one same node or the physical communication between two nodes, that is, $A_{s_0t_0,s_0t_1,i_0m_0,i_1m_1}$ can only be 1 for just one connection between two machines or one machine:

$$\sum A_{s_0t_0,s_0t_1,i_0m_0,i_1m_1} = 1 \tag{7.6}$$

In addition, to capture that for each task edge, $A_{s_0t_0,s_0t_1,i_0m_0,i_1m_1}$ is 1 for 1 if A_{s_0,t_0,i_0,m_0} and A_{s_1,t_1,i_1,m_1} are both 1, we have the following constraint:

$$A_{s_0t_0,s_0t_1,i_0m_0,i_1m_1} - A_{s_0,t_0,i_0,m_0} - A_{s_1,t_1,i_1,m_1} \geq -1 \tag{7.7}$$

To capture the overlapping CPU cores constraints, for each pair of tasks executing on the same instance a, check that two tasks do not overlap. In other words, for any (s_0t_0,s_1t_1) pairs when there is no dependency for task as genome s_0, stage t_0 and task as genome s_1, stage t_1, we have the following constraints:

$$
\begin{aligned}
S_{s_1,t_1} - E_{s_0,t_0} &\geq -\Phi \times O_{s_1t_1,s_0t_0} \\
S_{s_1,t_1} - E_{s_0,t_0} &< -\Phi \times (1 - O_{s_1t_1,s_0t_0}) \\
S_{s_0,t_0} - E_{s_1,t_1} &\geq -\Phi \times O_{s_0t_0,s_1t_1} \\
S_{s_0,t_0} - E_{s_1,t_1} &< -\Phi \times (1 - O_{s_0t_0,s_1t_1}) \\
A_{s_0t_0,i,m} + A_{s_1t_1,i,m} + O_{s_1t_1,s_0t_0} + O_{s_0t_0,s_1t_1} &\leq 3
\end{aligned}
$$

Φ is a large enough integer constant to bound the difference.

To account for deadline constraints, we have the following set of constraints:

To capture that the whole pipeline processing needs to be finished by deadline DDL, end time of last stage (T-1) in every genome should be smaller than DDL, we have the following constraints:

$$
\begin{aligned}
S_{s_0,T-1} + &\sum_{i \in I, m \in M} A_{s_0,t=T-1,i,m} \times stage_{i,t=T-1} \\
&+ \sum_{t_1=T-1, t_0=T-2, i_0 \neq i_1 \vee m_0 \neq m_1} A_{s_0t_0,s_0t_1,i_0m_0,i_1m_1} \times download_{i_1,t} \\
&\leq DDL, \forall s_0 \in N
\end{aligned}
$$

To capture the precedence in the stages within a genome, the start time of the subsequent stage is larger than the finish time of the previous stage, we have the following constraints:

$$
\begin{aligned}
& S_{s_0,t} + \sum_{i \in I, m \in M} A_{s_0,t,i,m} \times stage_{i,t} \\
& \quad + \sum_{t_1=t+1, t_0=t, i_0 \neq i_1 \vee m_0 \neq m_1} A_{s_0 t_0, s_0 t_1, i_0 m_0, i_1 m_1} \times upload_{i_0,t} \\
& \quad + \sum_{t_1=t, t_0=t-1, i_0 \neq i_1 \vee m_0 \neq m_1} A_{s_0 t_0, s_0 t_1, i_0 m_0, i_1 m_1} \times download_{i_1,t} \\
& \quad \leq S_{s_0,t+1}, \forall s_0 \in N, \forall t \in T \setminus T - 1
\end{aligned}
$$

To capture that start time of the first task in each genome is larger than *setup* time, we have the following constraints:

$$S_{s,0} \geq setup \tag{7.8}$$

7.2 Experiment Setup and Evaluation

On the AWS amazon EC2, there are many CPU instances to choose from. Different instances have different CPU types (Table 7.1), number of CPU cores and prices [Ama19b] (Table 7.2).

7.2.1 Profiling

The runtimes of a whole genome sequence NA12878-Garvan (Table 6.5) of each stage in different CPU instances are profiled in Table 7.3.

Table 7.3: Amazon EC2 instances and runtime (seconds) for different stages.

	m5.large	m4.x	m5.x	c5.2x	m4.2x	m5.2x	f1.2x	c5.4x	m4.4x	m5.4x	m4.10x	f1.16x
Aligner	NA	NA	NA	NA	77076	65654	73688	36080	40283	33297	19684	11889
BQSR	24536	19401	12820	16896	9988	8073	10107	12352	5722	3650	3065	2374
HTC	73800	46246	37597	42875	27319	23726	17251	15728	17539	13931	11607	4442

7.2.2 MILP Solving

We use the profiling data to build ILP formulation as described in Section 7.1. We automate the modeling part by writing Python scripts to generate MIP LP files [IBM18b] that consider different deadline constraints. The generated LP files are solved by calling IBM's CPLEX solver [IBM18a]. CPLEX solver gives the optimal configuration that costs the least under certain time constraints. We sweep the deadline time constraints from 19000 seconds (5.3 hours) to 99000 seconds (27.5 hours), get the cost and corresponding configurations as shown in Figure 7.1. We conduct a polynomial curve fitting and the fitting equation is $y = -3E^{-23}x^5 + 2E^{-17}x^4 - 4E^{-12}x^3 + 4E^{-07}x^2 - 0.0174x + 295.55$ with $R^2 = 94\%$.

7.2.3 When Mocha is Applied

As explained in Chapter 6, on Amazon EC2, running HTC can use the Mocha framework to improve the latency and cost at the same time. In this way, we introduce the new instance type that combines one `f1.2xlarge` and `m4.16xlarge` instance; that is, `f1.2x + m4.16x`. `f1.2x + m4.16x` runs HTC as fast as `f1.16xlarge`.[1] However, they are priced at 1.65+ 3.2 = 4.85 dollars per hour, which is much less than 13.2 dollars per hour of `f1.16x`. We take into consideration this instance, and sweep

[1]Runtime of three stages for this configuration are 15657, 2380, 4267 seconds.

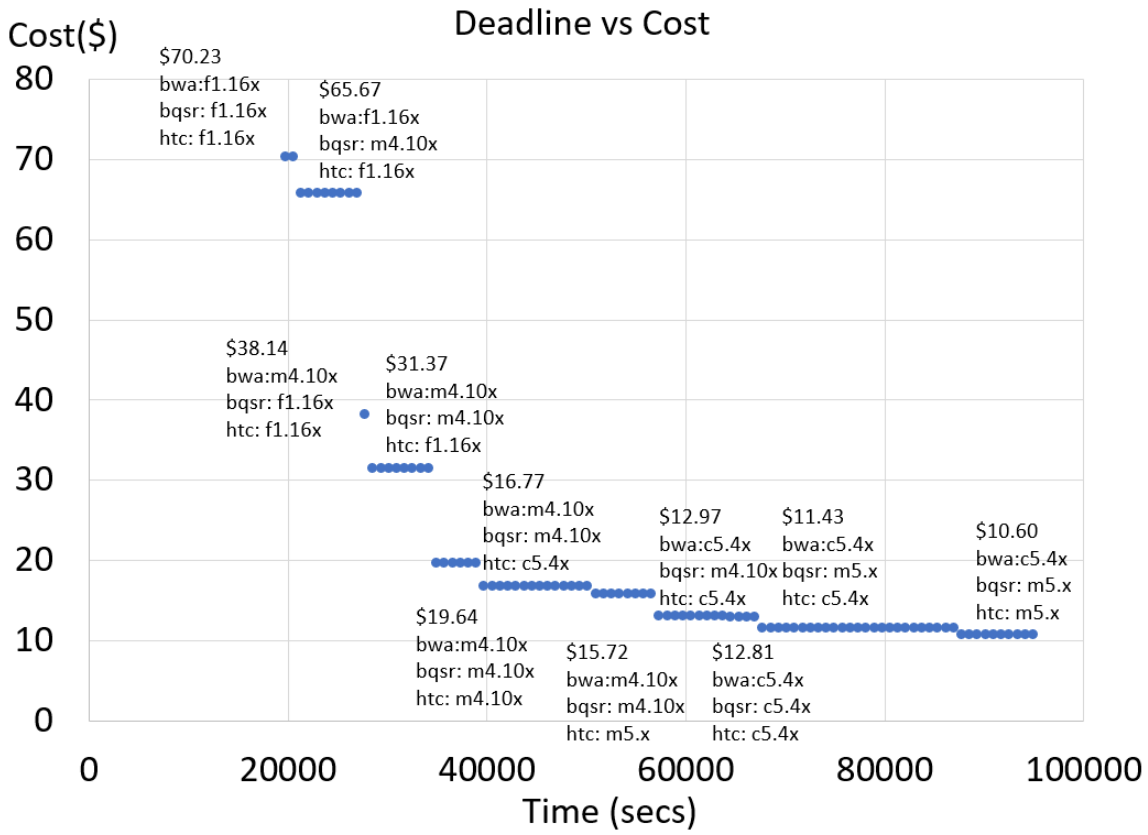


Figure 7.1: Cost under different time constraints.

the deadline time constraints from 19000 seconds (5.3 hours) to 99000 seconds (27.5 hours). The cost and corresponding configurations are shown in Figure 7.2. As shown here, when using `f1.2x+m4.16x`, the cost to meet a deadline as small as 19800 seconds (5.5 hours) reduces from \$70 to \$56. We conduct a polynomial curve fitting and the fitting equation is $y = -9E^{-22}x^5 + 3E^{-16}x^4 - 3E^{-11}x^3 + 2E^{-06}x^2 - 0.0478x + 518.53$ with $R^2 = 95\%$.

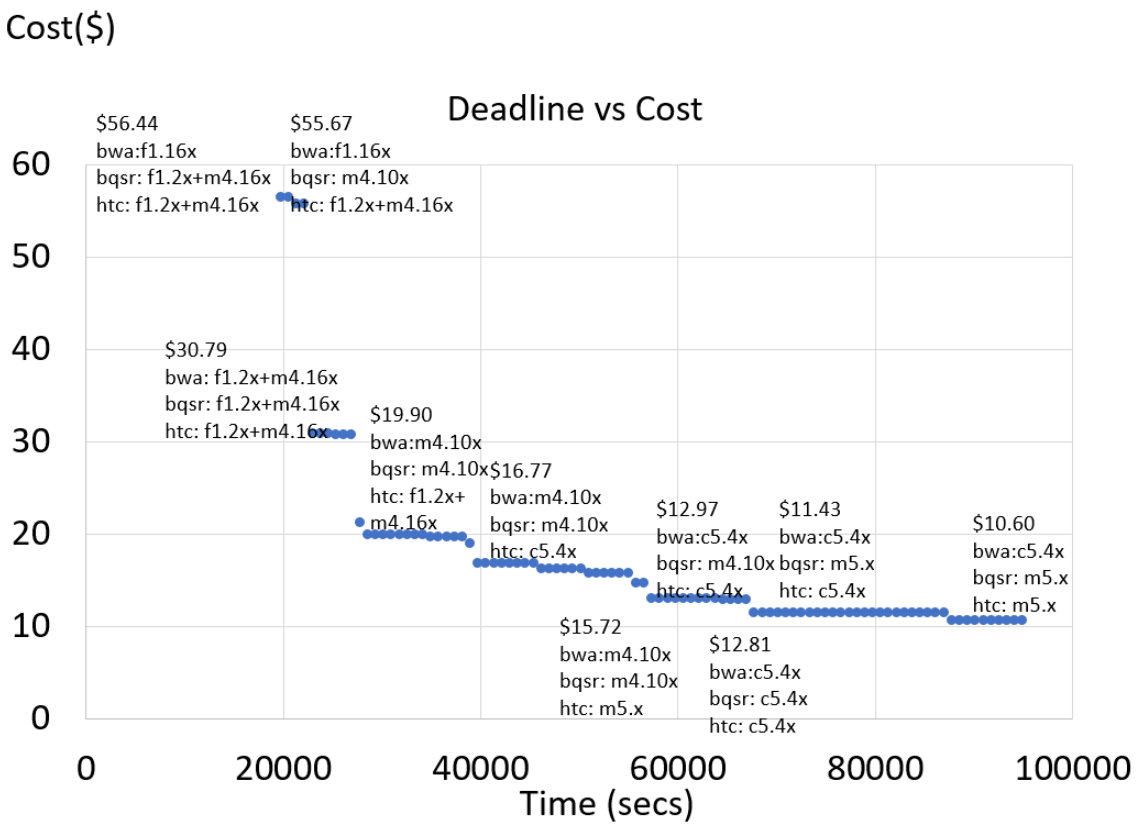


Figure 7.2: Cost under different time constraints when Mocha framework is used.

7.2.4 Multiple Genomes

When there are multiple genomes, we can sweep the time deadline to a larger range and examine the cost. When a deadline is smaller than 123800 seconds, the optimal configuration for #genome = 1 and #genome = 2 are the same. When there are two genomes and the deadline constraint is larger than 123800 seconds, the cost to run two genomes can be further reduced from $10.603 to $10.563. The minimum cost of running two genomes is $10.551, where two genomes run in sequential on the same machine. Here, after the first genome has finished execution of BQSR, HTC on the

`m5.xlarge` instance, the second genome starts execution of BQSR HTC stages. The savings from $10.603 to $10.563 comes from reusing the same machine, thus saving the one-time effort in machine setup.

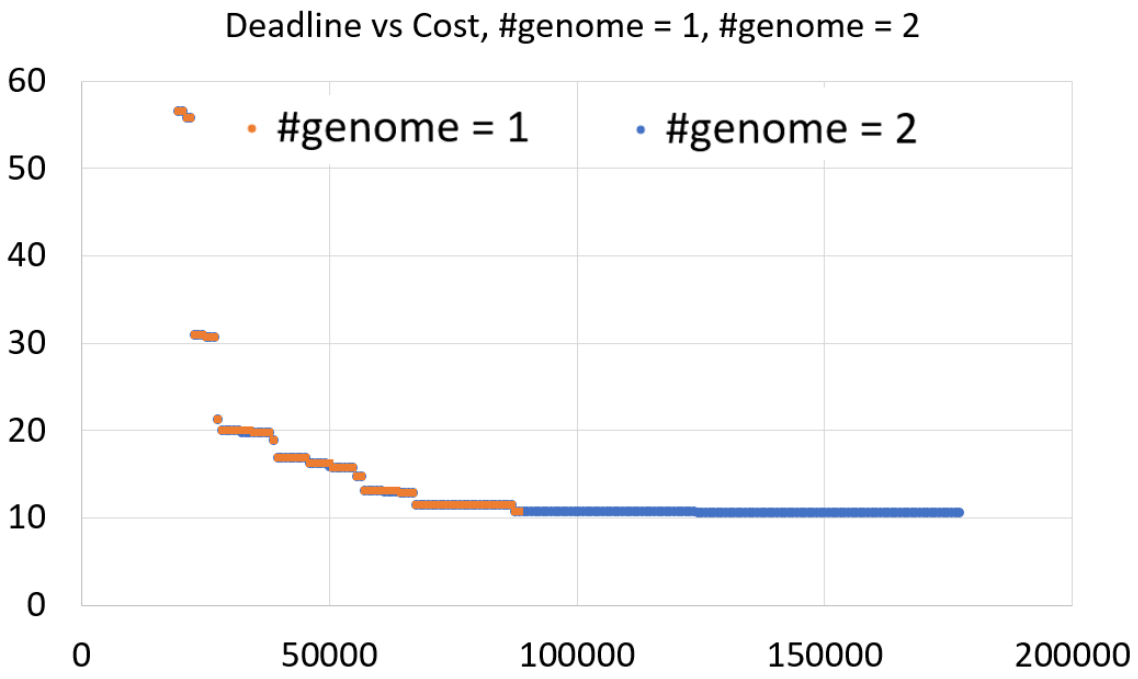


Figure 7.3: Cost under different time constraints when number of genome is one and two.

When there are more genomes—for example, three, four and five genomes—the costs show trends similar to when there is only one genome. However, when there are more genomes, the minimum cost-per-genome further reduces, as shown in Table 7.4. The reason is that when there are more genomes and the deadline constraints are loose enough, it is possible to execute all the genomes in the same machine sequentially. Thus, the savings is in the one-time effort to set up the machine. However, this saving gives less and less marginal benefits. In this way, duplication of configurations when

the number of genomes = 2 is good enough for any given number of genomes (larger than 2).

Table 7.4: Minimum cost per genome when number of genome(s) are 1,2,3,4 and 5.

#genome(s)	deadline (seconds)	minimum cost per genome	marginal benefits (%)
1	87800	10.603	NA
2	138200	10.551	0.49%
3	188600	10.533	0.17%
4	239000	10.524	0.09%
5	289400	10.519	0.05%

7.3 Discussion on Spot Instances

Amazon AWS offers spot instances that are unused and priced much lower than on-demand prices. Instead of fixed prices for on-demand instances, spot instances are priced dynamically based on demand. The spot instances will be interrupted when the bid prices are lower than current spot prices [Amab]. We look at three months of pricing history [Amae, Amad, Amac, Bri] for spot instances in US regions and report the highest spot instance prices for each instance as shown in Table 7.5 (accessed on July 10th 2019). We request the spot instances at the highest prices within three months to minimize the interrupted rate (less than 5% in a month [Amab]).

We sweep the deadline time constraints from 19000 seconds to 99000 seconds when spot instances are considered; the cost and corresponding configurations are shown in Figure 7.4. As shown here, the total out-of-pocket cost is reduced from when `f1.2x+m4.16x` is used; cost that meets a deadline as small as 19800 seconds reduces from $56 to $17—which saves around 70% of on-demand instances. When the deadline

constraint is loosened to more than 87800 seconds (24 hours), the out-of-pocket cost can be as small as $4.84 (less than $5) per whole genome.

Table 7.5: Amazon EC2 instances and highest spot instance prices within three months (dollars per hour).

	m5.large	m4.x	m5.x	c5.2x	m4.2x	m5.2x	f1.2x	c5.4x	m4.4x	m5.4x	m4.10x	f1.16x	m4.16x
on-demand price ($/hr)	0.096	0.2	0.192	0.34	0.4	0.384	1.65	0.68	0.8	0.768	2	13.2	3.2
spot instance price ($/hr)	0.0323	0.0615	0.0771	0.1365	0.1236	0.1592	0.495	0.2805	0.2548	0.3566	0.6155	3.96	0.9847

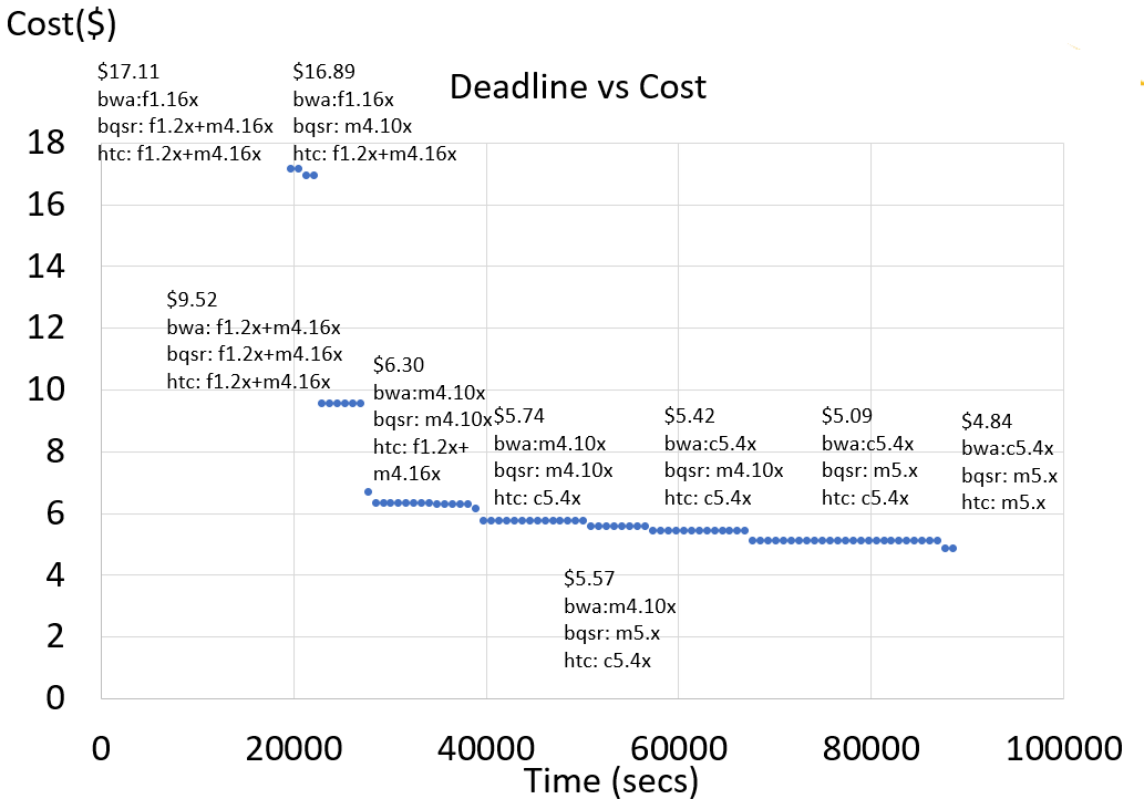


Figure 7.4: Cost under different time constraints when spot instances are used.

We also design a mechanism to check whether a spot instance is interrupted. Once each launched task starts, we append one line of information in corresponding instance status files shared on the S3 bucket. For each launched spot-instance, the instance

status file includes input genome information, stage information, and start/finish flags. Once the task finishes, we append another line of information in the instance status file. The master node that is responsible for launching all the spot-instances and tasks periodically (every minute) checks the status of all launched spot-instances. Once it detects that a spot-instance is interrupted, it checks the corresponding status file and relaunches the task that does not have the finish tag line information.

7.4 Related Work in Scheduling for Optimal Cost

We first summarize the scheduling problem for optimal cost. Many prior work study optimal cost scheduling on public clouds considering using different computation resources, storage resources and data management. [DSL08] examines the cost of running scientific workloads (astronomy application) on public clouds by using different execution and resource provisioning policies that includes different numbers of provisioned processors requested for the application and different data management solutions. This paper uses simulator to evaluate different execution cost models. Many other work use integer programming model to solve the cost optimal scheduling problem. [VVB10] studies the cost-optimal scheduling in hybrid clouds, where internal private IT infrastructures are in tandem with the public cloud services. It formulates the problem as a linear programming problem and investigates when there are different instances with different memory, CPU, and data communication overheads. However, the paper does not consider the persistent storage cost. [ZZT13] also studies the cost optimal scheduling problem for hybrid clouds where external public cloud resources are requested when private local resources are not efficient. They propose integer programming models to solve the problem. In their modeling, they

omit the storage resource in the model as well. HCOC [BM11] decides what tasks in a workflow are executed in a private cloud or in a public cloud provider to minimize the monetary costs and makespan at the same time in hybrid cloud settings. [MH11a] propose auto-scaling methods to guarantee that the jobs are finished within specified deadlines at minimum cost. They assume the application owners do not know the workloads ahead. Thus, it requires a monitor-control loop, where in each loop a new scheduling and a splicing decision are reevaluated based on updated workload information. This also belongs to dynamic scheduling problems while our work belongs to static scheduling problems. [MJD15] discuss new scheduling algorithm based on both static and dynamic strategies for task scheduling and cloud resource provisioning. They also take into consideration runtime prediction uncertainties, task failures and provisioning overheads. [Man15] focuses on virtual machine (VM) allocation problems to minimize costs using the host, with an emphasis on cost from energy consumption and penalties in migrations of VMs while meeting quality of service (QoS) requirements. [CCS10] investigates the cost optimization by using spot instances (SIs) for MapReduce workloads. Unlike on-demand and reserved VM instances, SIs are inexpensive; however, they are prone to early termination.

Instead of extensively profiling each application in each instance ahead of time, many prior work [DK13, DK14] use feature extractions and predictions to characterize an a new workload. Paragon [DK13] is an online datacenter scheduler that considers heterogeneity and interference to guarantee QoS. Instead of extensively profiling every single application in each instance, Paragon uses collaborative filtering algorithms to characterize every new application by using the historical data of previously seen applications in the system. Paragon is very scalable and applies to a large number of servers (over tens of thousands) and experiment results show that Paragon achieves

91% performance guarantees and improves server utilization in large-scale cloud systems where large amounts of workloads are collocated. Heterogeneity comes from the fact that servers are usually replaced in 15-years. Heterogeneity incurs 2x runtime difference for a single application. Interference stems from the fact that multiple workloads are usually scheduled on a same server to achieve higher utilization and cost efficiency. However, these workloads usually impair performance of each other as caches, memory, networking, and storage resources are shared. Paragon uses offline training on previously seen applications and requires minimal profiling for a newly arrived workload. Experiment results show that Paragon effectively characterizes new workloads while maintaining high QoS for different workload cases. Quasar [DK14] presents a cluster management system that solves the low resource utilization issue in the public cloud and provides high QoS at the same time. When users do not understand the workload characteristics and resource requirements, they might allocate too many reserved instances and the reserved resource is underutilized (<20% as demonstrated in this paper for a production cluster at Twitter). Low utilization in the cloud incurs wasted energy and unnecessary operational expenses. Quasar is a cluster manager that determines how to allocate the appropriate resources for each workload and how to choose the right instances. In order to achieve these goals, it needs to meet performance and resource utilization targets at the same time. Quasar lets users to simply tell the performance requirements instead of detailed planned resource reservations, which allows transparent handling of resource allocations to end users. In addition, Quasar uses small profiling information from the newly arrived workload, compares it to the previously scheduled workloads, and performs machine learning algorithms to evaluate the impact of scaling-up, scaling-out, different server configurations, and collocated workloads interference. Quasar then

predicts results to allocate the right set of resources that are needed for the workload. If the performance severely differs from the QoS requirements, Quasar adjusts the resource allocation during the runtime. Quasar is evaluated on public cloud servers with a broad range of application workloads—including data analytic frameworks, latency-driven workloads, and batch workloads as well. Experiments show that server utilization is improved by 47% and individual workload performance is also improved. These two work can be useful in our framework to ease the burden to do extensive profiling for the studied application.

While our work mainly studies scheduling strategies for on-demand resources in public cloud, some other work also take reserved resources into consideration. HCloud [DK16] presents hybrid scheduling strategies that use both reserved and on-demand resources in the public cloud considering the overall workload and instance unpredictability. There are two types of resources in the public cloud—reserved instances and on-demand resources. For each type of resource, there are different types and sizes of instances. The paper studies general workload scenarios that include three different sets that have a combination of batch (throughput-driven) and latency-driven applications with varying levels of load variability. Knowledge of the applications is not needed to be known ahead of scheduling. HCloud, as a hybrid provisioning system, uses both reserved instance and on-demand instance, which provides the best cost efficiency for both long-term and short-term loads. There are three baseline scheduling strategies: statically reserved resources strategy (SR), dynamic on-demand full (OdF), dynamic on-demand mixed scheduling (OdM). There are also two hybrid provisioning strategies: one only uses larger (HF) and stable on-demand instances (16 vCPUs) and the other uses a mix (HM) of different instance types including smaller instances. HCloud determines the number of resources

and where to schedule the workload to satisfy QoS constraints. By leveraging the lightweight profiling information [DK14], hybrid strategies HF and HM achieve within 8% of the performance of SR strategy and outperform OdF and OdM strategies. In terms of cost, HF and HM incur lower costs than SR, OdF and OdM. Although these prior work made significant contributions to cost optimization in public cloud, to the best of our knowledge, we are the first to propose composable instances from a combination of general purpose instances and CPU-FPGA instances in public cloud to save more in cost optimization.

CHAPTER 8

Latency Optimization for Domain Specific Application in Private Cloud

In this chapter we discuss the application of modeling performance in the private cloud level—that is, optimizing the overall latency of running a specific application, e.g., whole genome processing given certain cloud settings. We study latency optimization in private workload specific clusters (WLSC) considering this scenario: an institute, a hospital or a company has a highly repetitive workload and need to finish all the work as soon as possible. Optimizing the latency of these workloads in these private WLSC is of the most importance. WLSC tend to have more constraints because the hardware resource is more likely to be fixed after the private clusters are setup. For example, there are certain number of CPU cores in a cluster node, and there are certain size of storage space in a cluster node. Moreover, the workload to be studied has more detailed performance characterization and modeling. For example, the whole genome sequencing pipeline we study has two stages and the runtime of each stage is characterized as a sequential part and a parallizable part. A key question arises: how can we schedule highly repetitive workloads on these WLSC in a minimum latency? We show in Section 8.1 that we can model this problem as a MILP problem. In Section 8.2, we present the results from the solver for small problem sizes when the number of genomes in the workload is small. When the number of genomes is

larger, runtime from the solver is huge. We propose three heuristics that achieve smaller optimality gap with those of the optimal solution, and use the runtime from heuristics as base cases to construct optimal scheduling when there are more genomes. The proposed heuristics are scalable to a larger number of genomes. Due to the time constraint, we consider CPU optimization only and leave the integration of FPGA accelerations as future work.

8.1 Modeling

In a private cloud, we consider that there are multi-core servers with certain sizes of storage disks (SSDs or HDDs). When running genome processing in a server, launching multiple threads can reduce the execution time for a single genome. Based on this observation, we model the runtime of each processing stage as a sequential part c_0 and parallelizable part t_0. The sequential part represents time in reading the reference genome, dbsnp files, or other constant overheads. The parallelizable part represents time in processing genome base pairs that can be fully distributed among multiple threads. If scheduling a genome to use all the threads, every thread is paying idle CPU cycles while waiting for the constant parts. If scheduling a genome to use just one thread and there are multiple genomes running in parallel, then it requires a lot of storage space for intermediate data. Therefore, we first model the problem as an optimal latency scheduling problem under certain CPU core constraints and storage space in the following sections.

8.1.1 Input Parameters

N: Set of whole genomes to be processed.

T: Set of stages for all genomes.

R: Set of cores in a server.

Y: Set of storage spaces. Each storage space is for a whole genome.

c_0, t_0, c_1, t_1: Time for series part and parallel part for stage 0 and stage 1 in genome processing. For example, when there are 4 cores for a stage 0 task, runtime of the task is $c_0 + \frac{t_0}{4}$.

8.1.2 Variables

$p_{i,j}$: Allocated number of cores variables of genome $i, i \in N$, stage j. Runtime of a stage is $c_j + \frac{t_j}{p_{i,j}}$ The value $p_{i,j}$ is an integer.

$S_{i,j}$: Computed start time of genome i, stage j. The value is an integer.

$E_{i,j}$: Computed end time of genome i, stage j. The value is an integer.

$A_{i,j,a}$: Allocation variables indicating genome i, stage j is allocated to core $a, a \in R$. The value is binary.

$C_{i,b}$: Allocation variables indicating genome i is scheduled to storage space $b, b \in Y$. The value is binary.

$O_{i_0 j_0, i_1 j_1}$: A flag indicating whether two tasks overlap (on a same CPU core). The value is binary.

$Q_{i,j}$: A flag indicating whether two genomes overlap (on a same storage space). The value is binary.

X: An artificial variable that represents the makespan of the application. The value is an integer.

8.1.3 Objective Function

The objective function defines the optimal solution target. The optimal solution ensures executing genome pipeline processing on the given private cloud using the least amount of time given a set of CPU resource and storage resources. Therefore, the objective function is:

$$\min X \tag{8.1}$$

8.1.4 Constraints

The optimization problem needs to first consider deadline constraints since the latency is the optimization target. Then, scheduling needs to consider the hardware constraints including that there are a certain number of cores allocated for a genome stage, there is one storage space allocated for a genome, and if two tasks are scheduled on the same CPU core or storage space, they can not overlap with one another. Last, in order to obtain a MILP problem, we use piecewise-linear approximation to replace the nonlinear terms $\frac{1}{p_{i,j}}$.

To account for deadline constraints, we have the following set of constraints:

Runtime X is the maximum end time of the last stage (here is stage 1) in every genome:

$$S_{i,1} + c_1 + \frac{t_1}{p_{i,1}} \leq X, \forall i \in N \tag{8.2}$$

To capture the precedence in the stages within a genome, the start time of the later

stage is larger than the finish time of the earlier stage:

$$S_{i,1} \geq S_{i,0} + c_0 + \frac{t_0}{p_{i,0}}, \forall i \in N \tag{8.3}$$

To account for hardware constraints, we have the following set of constraints:

To capture that there are $p_{i,j}$ cores allocated for the task genome i, stage j:

$$\sum_{a \in R} A_{i,j,a} = p_{i,j} \tag{8.4}$$

To capture that each genome needs one storage space:

$$\sum_{b \in Y} C_{i,b} = 1 \tag{8.5}$$

To capture the overlapping CPU core constraints, for each pair of tasks executing on the same node a, check that two tasks do not overlap. Therefore for any (i_0j_0,i_1j_1) pairs when there is no dependency for task as genome i_0, stage j_0 and task as genome i_1, stage j_1, we have the following constraints:

$$\begin{aligned}
S_{i_1,j_1} - E_{i_0,j_0} &\geq -\Phi \times O_{i_1j_1,i_0j_0} \\
S_{i_1,j_1} - E_{i_0,j_0} &< -\Phi \times (1 - O_{i_1j_1,i_0j_0}) \\
S_{i_0,j_0} - E_{i_1,j_1} &\geq -\Phi \times O_{i_0j_0,i_1j_1} \\
S_{i_0,j_0} - E_{i_1,j_1} &< -\Phi \times (1 - O_{i_0j_0,i_1j_1}) \\
A_{i_0j_0,a} + A_{i_1j_1,a} + O_{i_1j_1,i_0j_0} + O_{i_0j_0,i_1j_1} &\leq 3
\end{aligned}$$

Φ is a large enough integer constant to bound the difference.

To capture the overlapping storage resource constraints, for each pair of genomes executing on the same storage space b, check that two genomes do not overlap. Then for any pairs of two genomes (i,j), we have the following constraints:

$$
\begin{aligned}
S_j - E_i &\geq -\Phi \times Q_{i,j} \\
S_j - E_i &< -\Phi \times (1 - Q_{i,j}) \\
S_i - E_j &\geq -\Phi \times Q_{j,i} \\
S_i - E_j &< -\Phi \times (1 - Q_{j,i}) \\
C_{i,b} + C_{j,b} + Q_{i,j} + Q_{j,i} &\leq 3
\end{aligned}
$$

To be noted here, depending on j is 0 or 1, the end time for genome i, stage j is :

$$E_{i,j} = S_{i,j} + c_j + \frac{t_j}{p_{i,j}} \tag{8.6}$$

The start time of genome i is the start time of genome i, stage 0 and end time for genome i is end time of genome i, last stage (here is stage 1).

$$S_i = S_{i,0}, E_i = E_{i,1} \tag{8.7}$$

Transformation of nonlinear $\frac{1}{p_{i.j}}$ terms into piecewise-linear approximation:

The above constraints have nonlinear terms $\frac{1}{p_{i,j}}$. In order to obtain a MILP problem, we transform $p_{i,j}$ into discrete values and introduce split factors as the following

$split_{i,j,a}$: Split flag indicating genome i, stage j has used a cores to do processing. The value is binary. After introducing this value, end time of genome i, stage j is rewritten as the following:

$$E_{i,j} = S_{i,j} + (c_j + \frac{t_j}{1}) \times \text{split}_{i,j,1} + (c_j + \frac{t_j}{2}) \times \text{split}_{i,j,2} + ... + (c_j + \frac{t_j}{R}) \times \text{split}_{i,j,R} \quad (8.8)$$

To capture that only one split factor is valid, we have the following constraint:

$$\text{split}_{i,j,1} + \text{split}_{i,j,2} + ... + \text{split}_{i,j,R} = 1 \quad (8.9)$$

To capture that there are in total $p_{i,j}$ cores, Equation 8.4 is rewritten as:

$$\sum_{a \in R} A_{i,j,a} = p_{i,j} = 1 \times \text{split}_{i,j,1} + 2 \times \text{split}_{i,j,2} + ... + R \times \text{split}_{i,j,R} \quad (8.10)$$

8.2 Evaluations

We evaluate the private cloud setting in the CDSC cluster where each server is a 56-core machine. The CPU is Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz. Each server has 2TB HDD storage space. For the WGS genome pipeline, the vcf file needs 20GB, fasta file needs 3GB. Input fastq file and bam file needs around 100GB. Thus, in a server, there can be at most 8 concurrent whole genomes being processed. We profile a BWA, GATK pipeline (BQSR+HTC) stage on the server, and obtain the constant runtime and parallelizable runtime as the following:

By using the profiling runtime, we automate the modeling part by writing Python scripts to generate a corresponding MILP LP file for a different number of input

Table 8.1: Constant and parallelizable runtime in private cloud.

stage	constant part	parallel part
BWA	5075	580122
BQSR+HTC	3994	372040

genomes. The generated MILP file is solved by the IBM CPLEX solver. The optimal runtime for different numbers of genomes is shown in Table 8.2.

Table 8.2: Optimal runtime when there are different numbers of genomes.

#genome	optimal runtime (seconds)
1	26071
2	43074
3	60269
4	77080

8.2.1 Configurations of Optimal Results

When #genome is 1, both stages use all CPU cores. When #genome is 2, both stages in two genomes use 28 cores. Thus runtime is $5075 + \frac{580122}{28} + 3994 + \frac{372040}{28} = 43074$. When #genome is 3, the three genomes are allocated with # cores as shown in Figure 8.1.

When #genome is 4, both stages in four genomes use 14 cores.

More genomes. When #genome is larger than 4, it takes an extremely long time to get the optimal solution. For example, when #genome = 6, the solver can not give optimal results after 366 hours even if the solver is calculating using multithreading. As shown in Section 8.1, the number of variables is $6N + 5NR + NY + 5C_N^2$. The number of constraints is $9N + 20 \times C_N^2 + 4C_N^2 \times R + C_N^2 \times Y$. We show the number

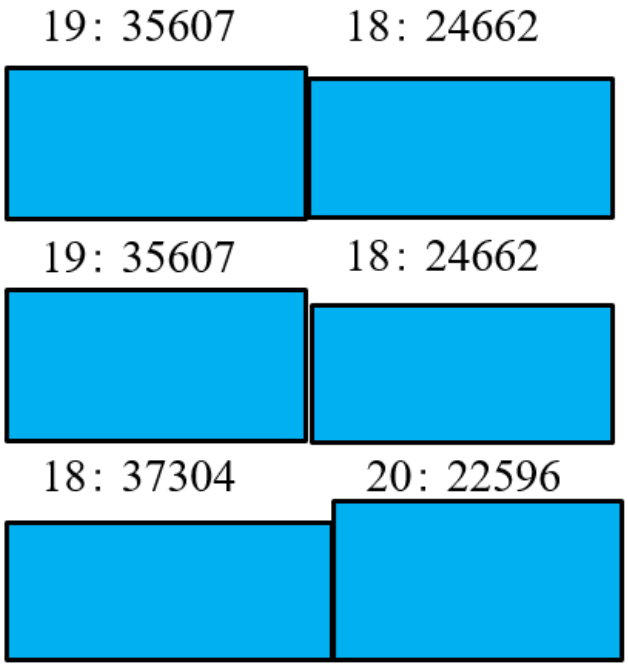


Figure 8.1: Configurations of optimal cost when #genome = 3.

of variables and constraints for different numbers of genomes in Figure 8.2. The fast-growing number of variables and constraints account for the long runtime in solving the MILP problem. In the following section, we propose three heuristics and compare their optimality gaps against the optimal solutions.

8.2.2 Heuristics

In order to compare the heuristics results with the optimal results, we design three problem sizes and try to obtain the optimal solutions as references for heuristic results.

- Case A: #cores = 14, #storage space = 2, which means the disk size is large enough for two genomes to be processed in parallel.

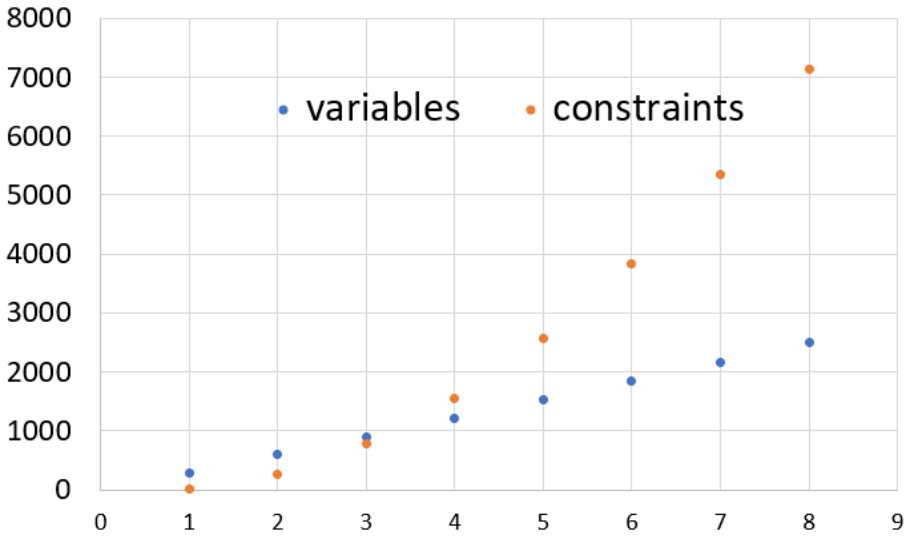


Figure 8.2: Number of variables and constraints (Y axis) for different numbers of genomes (X axis).

- Case B: #cores = 28, #storage space = 4.
- Case C: #cores = 56, #storage space = 8.

We propose three heuristics as demonstrated in the following sections.

Heuristic 1: Construct from smaller sequential patterns.

When there are more genomes, we can accomplish scheduling by using smaller sequential patterns then when there are fewer genomes, given the observations that calculating #genomes = 1,2 are fast for all three problem sizes. When the optimal solutions are known for #genome = 1, and #genome = 2, we can use these base results to construct any number of genomes. For example, we show how we can schedule #genome = 3, 4, 5 by using patterns of #genome = 1, 2.

We show the optimal results and heuristic 1 results in the following table for problem size Case A.

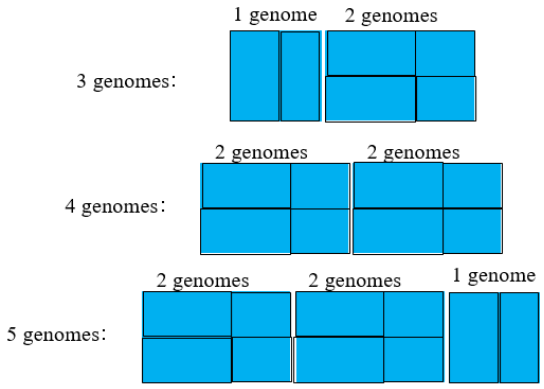


Figure 8.3: Heuristic 1: schedule by using #genome = 1,2.

Table 8.3: Optimal results and heuristic results for problem size Case A.

#genome	optimal	heuristic 1	optimality gap
1	77080	77080	0.000%
2	145091	145091	0.000%
3	221392	222171	0.352%
4	290182	290182	0.000%
5	365340	367262	0.526%
6	435273	435273	0.000%
7	510431	512353	0.377%
8	580364	580364	0.000%

We also show optimal results and heuristic 1 results in the following Table 8.4 and 8.5 for problem size Cases B and C .

Heuristic 2: Distribute cores among genomes as evenly as possible. When #genomes $\leq$ #storage space, we can always distribute cores among genomes evenly. For example, for problem size Case C, we can distribute the cores for the three genomes as 19,19,18—in total 56 cores as shown in the left part of Figure 8.4.

We show optimal results and heuristic 2 results in the following Tables 8.6 and

Table 8.4: Optimal results and heuristic 1 results for problem size Case B.

#genome	optimal	heuristic 1	optimality gap
1	43074	43074	0.000%
2	77080	77080	0.000%
3	113586	120154	5.782%
4	145091	154160	6.251%

Table 8.5: Optimal results and heuristic 1 results for problem size Case C.

#genome	optimal	heuristic 1	optimality gap
1	26071	26071	0.000%
2	43074	43074	0.000%
3	60269	69145	14.727%
4	77080	86148	11.764%

8.7 for problem size Case B and C . Compared to heuristic 1, heuristic 2 further reduces the optimality gap compared to the optimal results.

Table 8.6: Optimal, heuristic 1, heuristic 2 results for problem size Case B.

#genome	optimal	heuristic 1	optimality gap	heuristic 2	optimality gap
1	43074	43074	0.000%	43074	0.000%
2	77080	77080	0.000%	77080	0.000%
3	113586	120154	5.782%	114864	1.125%
4	145091	154160	6.251%	145091	0.000%

We examine one optimal result versus heuristic 2 when the optimality gap is not 0% for problem size Case C—that is, when #genome = 3 in Figure 8.4. The optimal solution does not schedule the same number of cores for all the stages of a genome. If a genome is scheduled with fewer cores than other genomes in the first stage, it might result in less runtime if it is given more cores in the second stage than than

Table 8.7: Optimal, heuristic 1, heuristic 2 results for problem size Case C.

#genome	optimal	heuristic 1	optimality gap	heuristic 2	optimality gap
1	26071	26071	0.000%	26071	0.000%
2	43074	43074	0.000%	43074	0.000%
3	60269	69145	14.727%	61966	2.816%
4	77080	86148	11.764%	77080	0.000%
5	NA	103151	NA	95628	NA
6	NA	129222	NA	114864	NA
7	NA	146225	NA	128089	NA
8	NA	172296	NA	145091	NA

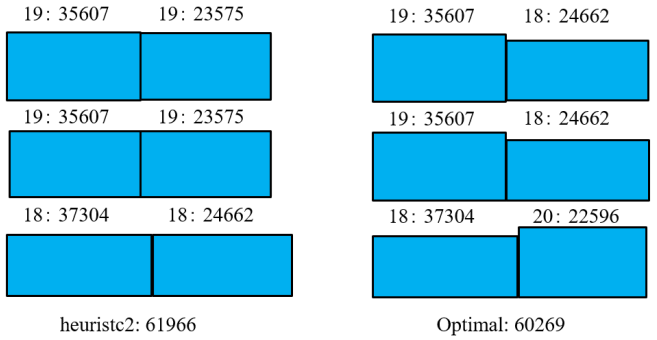


Figure 8.4: Comparing optimal configuration versus heuristic 2, #genome = 3, problem size Case C (#core = 56, # storage space = 8).

being scheduled with the same cores as in the first stage. We call this "balance-aware" heuristic 2. For other #genome values when mod(#core,#genomes) is not 0, we examine whether "balance-aware" heuristic 2 further improves the runtime compared to heuristic 2.

However, this strategy does not always work. For example, when #genome = 5, as shown in Figure 8.5, heuristic 2 schedules five genomes with cores 12, 11, 11, 11, 11. If each genome that is scheduled with 11 cores is scheduled with one more core in

the second stage, the genome that is scheduled with 12 cores in the first stage will be scheduled with 8 cores in the second stage. The total runtime increases to 103917 seconds and is worse than the heuristic 2 result.

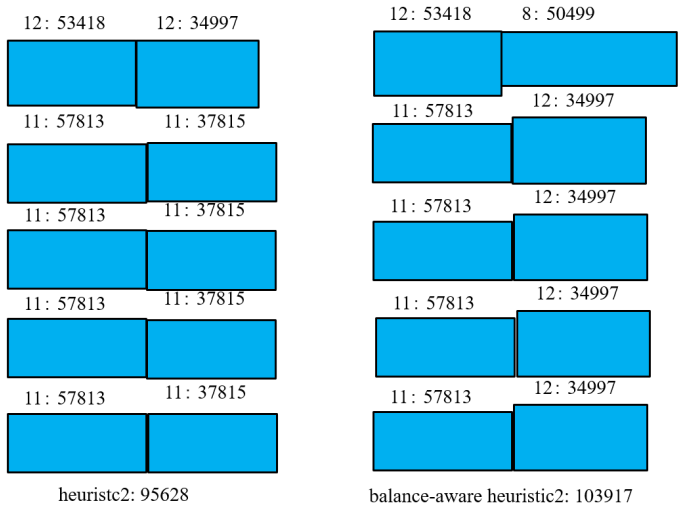


Figure 8.5: Heuristic 2, #genome = 5, problem size Case C (#core = 56, # storage space = 8). Runtime of "balance-aware" heuristic 2 is longer than heuristic 2.

When #genome = 6, as shown in Figure 8.6, the "balance-aware" heuristic 2 improves the execution compared to heuristic 2. We summarize the runtime from optimal results, heuristic 1, heuristic 2 and "balance-aware" heuristic 2 in Table 8.8 for problem size Case C.

8.2.3 Experiments

As shown in Figure 8.7. The modeling results achieve average error rate less than 4%, we verify the modeling by running NA12878Garvan [Sta16] on local clusters and comparing the experiment results with the modeling results of heuristic 2.

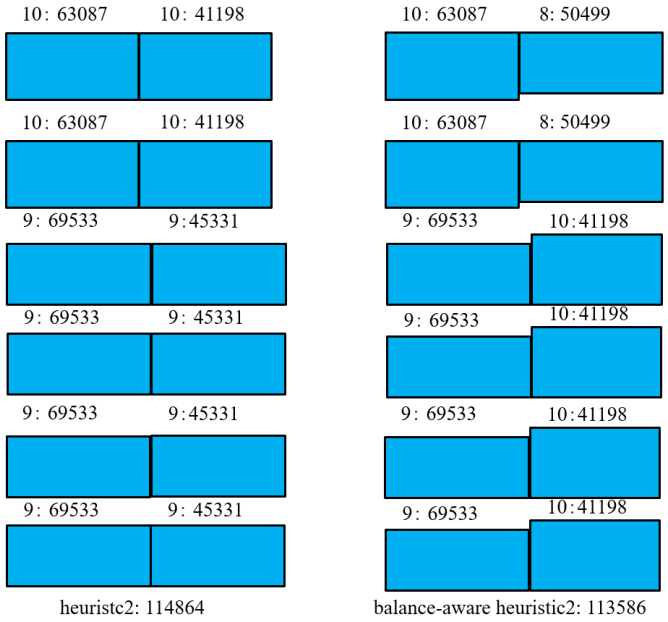


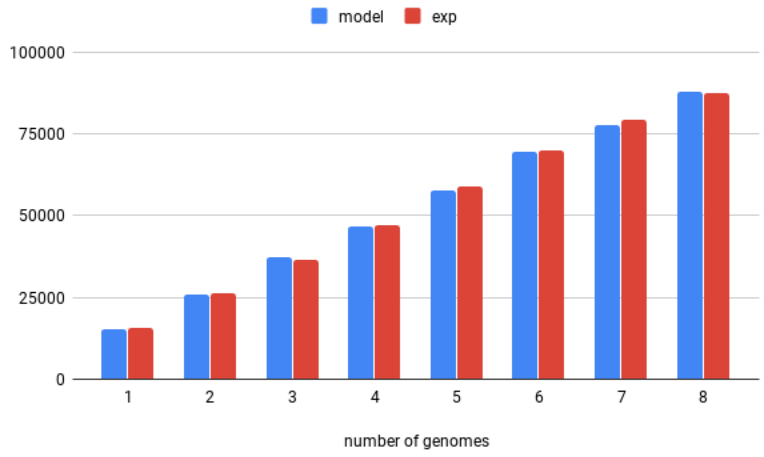
Figure 8.6: Heuristic 2, #genome = 6, problem size Case C (#core = 56, # storage space = 8). Runtime of "balance-aware" heuristic 2 is shorter than heuristic 2.

Table 8.8: Optimal, heuristic 1, heuristic 2,"balance-aware" heuristic 2 results for problem size Case C.

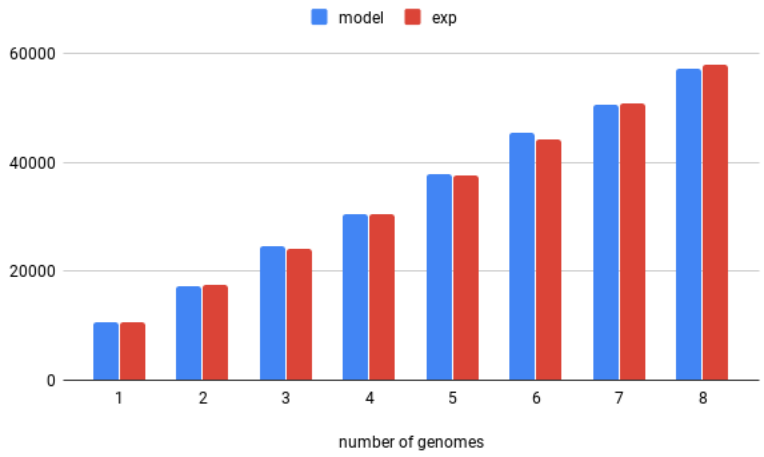
#genome	optimal	heuristic 1	optimality gap	heuristic 2	optimality gap	"balance-aware" heuristic 2	optimality gap
1	26071	26071	0.000%	26071	0.000%	26071	0.000%
2	43074	43074	0.000%	43074	0.000%	43074	0.000%
3	60269	69145	14.727%	61966	2.816%	60269	0.000%
4	77080	86148	11.764%	77080	0.000%	77080	0.000%
5	NA	103151	NA	95628	NA	95628	NA
6	NA	129222	NA	114864	NA	113586	NA
7	NA	146225	NA	128089	NA	128089	NA
8	NA	172296	NA	145091	NA	145091	NA

8.3 More Genomes: When #genome is Larger Than #storage space

By using the proposed heuristics and optimal solutions, we get a set of configurations for running genomes when #genome $\leq$ #storage space. We can use these configura-



(a) BWA



(b) GATK (BQSR+HTC)

Figure 8.7: Runtime of modeling and experiment measurements for stages.

tions as base cases to construct configurations for any number of genomes. Here we also propose two methods for the construction.

- Method 1: Treat every 8 genomes as a batch. For the last batch, pick the configuration from the base cases.
- Method 2: Formulate overall runtime as an ILP problem by introducing the number of base case, A_i—that is, using A_i times of base case i configuration to achieve optimal overall runtime.

We explain method 2 in detail in the following sections.

8.3.1 Input Parameters

N: Set of whole genomes to be processed.

Y: Limit of storage spaces. Each storage space is for a whole genome. For example, Y is 8 means at most 8 genomes can run in parallel.

T_i: Runtime of base case i. Base case i is the configuration of running $i <= Y$ genomes, that is base case 1, 2, 3, 4, 5, 6, 7, 8.

8.3.2 Variables

A_i : Allocated number of base case i, that is using A_i times of base case i configuration. Integer.

X: An artificial variable that represents the makespan of the application.

8.3.3 Objective Function

$$\min X \tag{8.11}$$

8.3.4 Constraints

Runtime X is the sum of runtime of using all base cases:

$$\sum_{i \in Y} A_i \times T_i \leq X \tag{8.12}$$

To capture the number of genomes to be processed is N:

$$\sum_{i \in Y} A_i \times i = N \tag{8.13}$$

8.3.5 Evaluations

We write Python scripts to simulate method 1 and generate LP files for method 2 to solve. We plot the runtime for the two methods as #genome increases from 9 to 100 in Figure 8.8.

Method 1 and method 2 achieve same runtime for some cases while method 2 ILP composition are better for some N values. We plot the runtime difference of method 1 minus method 2 on different numbers of genomes in Figure 8.9. We highlight three representative cases and explain the difference here. When #genome = 11, method 1 gives the scheduling for running 8 genomes in parallel followed by 3 genomes in parallel, while method 2 gives base cases running 7 genomes in parallel followed by 4 genomes in parallel. Method 2 saves 191 seconds. When #genome = 14, method

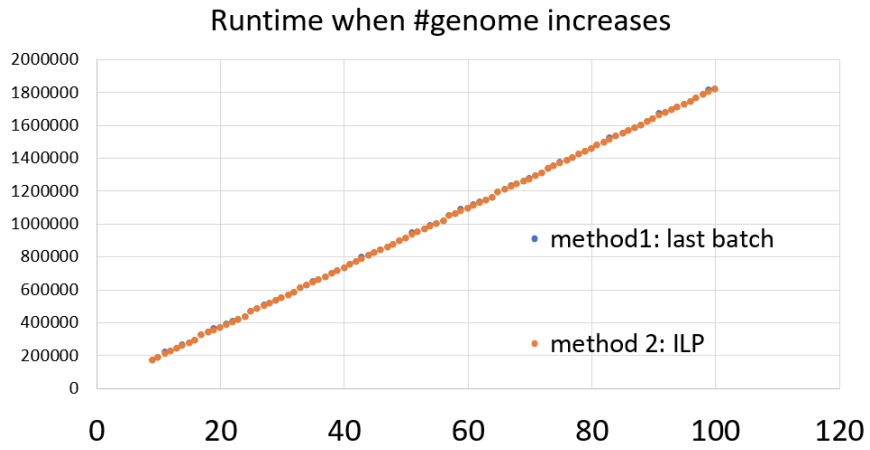


Figure 8.8: Runtime of method 1 and method 2 when N = 9..100.

1 schedules running 8 genomes in parallel followed by 6 genomes in parallel, while method 2 gives base cases running 7 genomes in parallel followed by 7 genomes in parallel. Method 2 saves 2499 seconds. When #genome = 21, method 1 schedules running two batches of 8 genomes in parallel followed by 6 genomes in parallel, while method 2 gives base cases running three batches of 7 genomes in parallel. Method 2 saves 1543 seconds.

8.4 Discussions on Other Applications

In order to examine the generality of our proposed methodology, we take another genome pipeline, VCPA (Variant Calling Pipeline and data management tool) [LVC18] as an example, run the experiments on local clusters and compare this to the modeling results. VCPA was developed to consistently and efficiently process sequencing data for the Alzheimer's disease according to whole-genome sequencing (WGS) and whole-exome sequencing (WES) genome analysis best practices. The VCPA pipeline follows

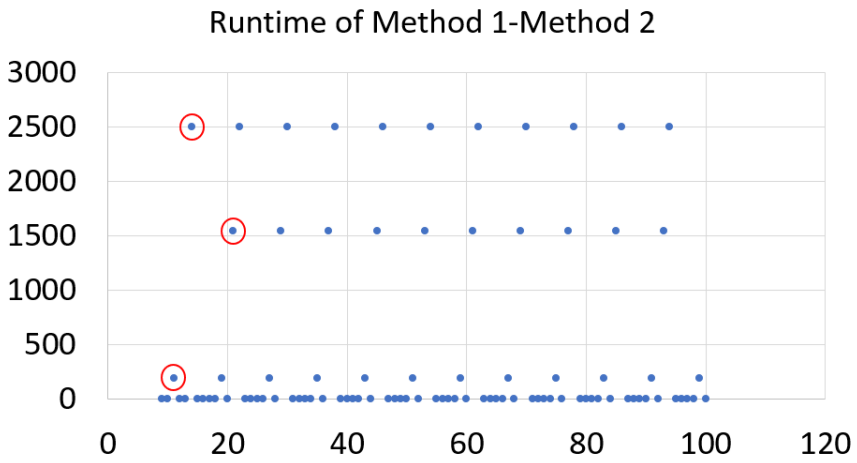


Figure 8.9: Runtime different between method 1 and method 2 when N = 9..100.

GATK 3.7 best practices on Germline Single Nucleotide Polymorphisms (SNPs) and Insertion/Deletion (Indel) Discovery [DBP11]. It contains the following stages: **Mapping**: This stage generates BAM files using BWA-MEM and marks duplicated reads.

Local Realignment: This stage performs local realignment near known indel sites (Align) and performs base quality scores recalibration (BQSR) by using GATK3.

Variant Calling: This stage performs variant calling on SNPs and indels. It generates genome variant call format (gVCF) files for each sample by using GATK3.

We profile the mapping stage (BWA) and local realignment and variant calling (Align+BQSR+VariantCalling) stage by using GATK3.7 in the CDSC cluster for NA12878-Garvan and obtain the following constant runtime and parallelizable runtime.

By applying the modeling formulation as shown in Section 8.1 and scheduling methods in Section 8.2, we verify the modeling by comparing the experiment results

Table 8.9: Constant and parallelizable runtime in private cloud for VCPA pipeline.

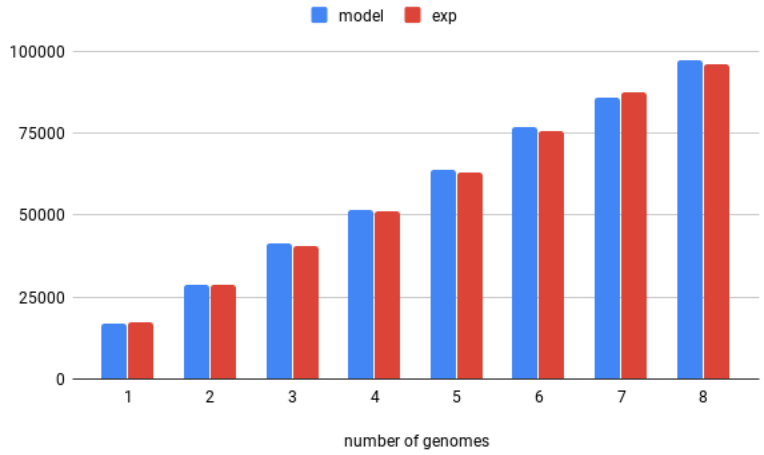
stage	constant part	parallel part
BWA	5633	642490
Align+BQSR+VariantCalling	5072	473935

with the modeling results of heuristic 2 in Figure 8.10. The modeling results achieve an average error rate of less than 3%. For "balance-aware" heuristic 2, we observe that the runtime experimental results reduce by 3% and 2% than heuristic 2 when number of genomes is set to 3 and 6, which match with with modeling results.

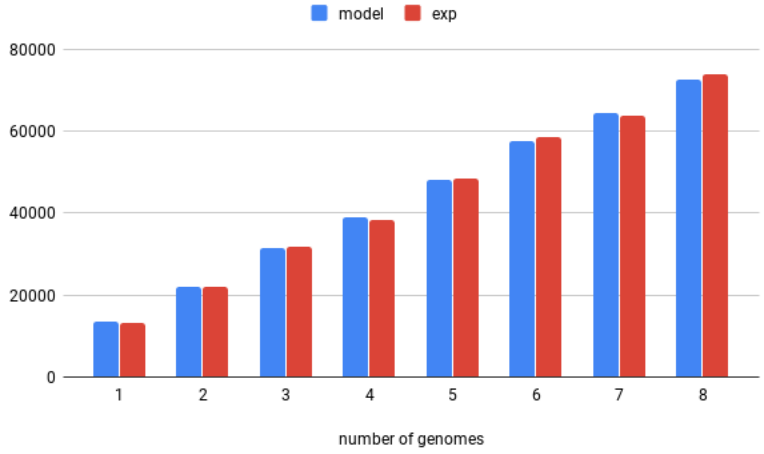
The total CPU time of each genome of modeling and experiment measurements for stages in VCPA pipeline are shown in Figure 8.11. When the number of genomes running in parallel increases, each genome is allocated with less CPU cores. Therefore, less constant overhead is paid in each CPU core, which leads to less total CPU time.

8.5 Related Work in Scheduling for Optimal Runtime

In this section we summarize the scheduling problem for an optimal runtime. Many prior work use MILP formulation to do optimal runtime scheduling considering computation resources including CPUs, GPUs and FPGAs, storage resources and I/O communications. [BSH18] presents a generic MILP formulation for the performance modeling of a heterogeneous cluster with active storage devices and an accelerator as well. In the heterogeneous cluster, there are different types of compute nodes, including client nodes, middleware servers, and high performance nodes with accelerators and SSDs. The paper formulates the optimization problem to achieve the least latency given applications represented by directed acyclic graphs (DAG) and heterogeneous

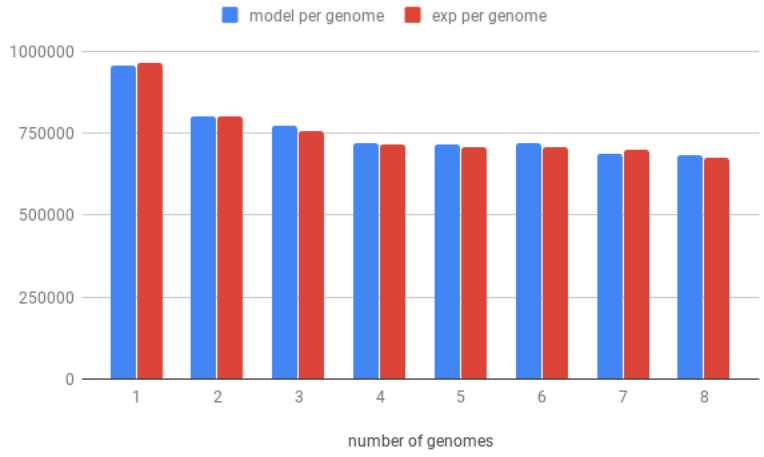


(a) BWA

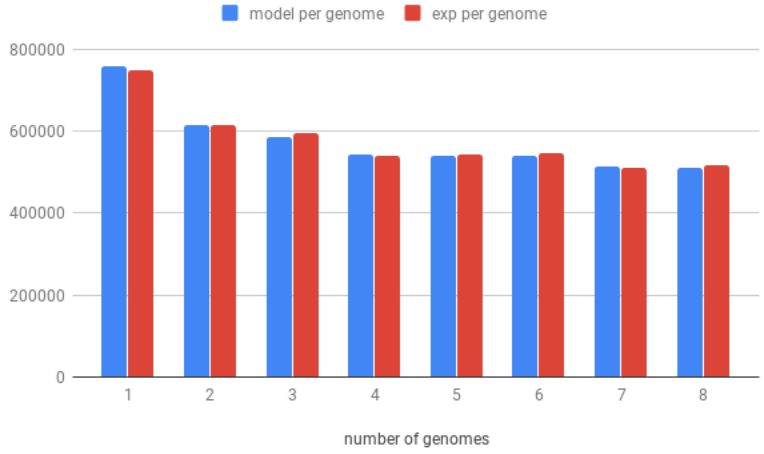


(b) GATK 3.7 (Align+BQSR+VariantCalling)

Figure 8.10: Runtime of modeling and experiment measurements for stages in VCPA pipeline.



(a) BWA



(b) GATK 3.7 (Align+BQSR+VariantCalling)

Figure 8.11: Total CPU time of modeling and experiment measurements for stages for each genome in VCPA pipeline.

system topology. It gives optimal solutions for small-scale clusters and genetic algorithms for larger sized problems. This work belongs to the static task scheduling problem to achieve max performance. Our work is most related to this work as we use similar method in MILP formulation. However, our work studies a repetitive workload with two stages and propose efficient heuristics to solve the scheduling problem.

In our work, we model the runtime of a task when using different cores as a nonlinear function of the number of cores. This is inspired by the runtime model proposed in [NST18], which studies fork-join malleable tasks scheduling. Malleable tasks are those that can be split into multiple subtasks and can be executed by multiple cores simultaneously. The scheduling is fine grained and each CPU core is considered as a hardware resource. The runtime of the task is a nonlinear function of the specified parallelism.Our work also adopts split factors in ILP formulation to describe task runtime given a set of CPU cores from the paper. The paper employs the ILP approach. In the ILP approach, $split_{ik}$ factors, indicating tasks i split into k subtasks are introduced to determine the parallelism of task i. Our work puts forward heuristics that achieve very small optimality gaps and are more scalable solutions to larger input.

Instead of using ILP, there are also other approaches to solve the optimality problem. [SM91] present the approach of multiprocessor scheduling by using the theory of optimal control. They assume that every task can be executed by an arbitrary continuous number of processors at any time in parallel.

Our work only considers homogeneous computation resources and some other work take adaptive multicores and other heterogeneous systems into consideration. [PM13]

addresses optimal scheduling problems where parallel and sequential applications both exist on an adaptive multicore system. Smaller cores can also be coalesced at runtime to join as a complex core for sequential applications that can exploit instruction-level parallelism. For parallel applications that exploit thread-level parallelism, multiple simple cores are executed in parallel. HDSS [BBG13] proposes a dynamic load balancing self-scheduler for loop iterations on heterogeneous systems to optimize performance. HDSS also dynamically sends block size to an accelerator (GPU or FPGA) to minimize underutilization and load imbalance between host machines and accelerators. [SRF12] creates a runtime system that partitions an accelerated OpenMP code region across the CPU cores and GPU cores to achieve performance improvement over using CPUs or GPUs only. The paper also discusses three schedulers: static, dynamic and combined scheduling policies. [ATT10] designs a versatile distributed framework and proposes dynamic task scheduling policies on heterogeneous clusters that have CPUs, FPGAs, and GPUs to optimize the performance. The paper considers computation and communication patterns on different hardware platforms. It also takes into consideration the synchronization and data communication overhead across hosts and accelerators. [BLB11] presents a supervised online learning approach to evaluate the performance of CPU/GPU/FPGA systems for incoming computational tasks in discrete time steps. The paper evaluates the proposed learning module and subsequent dynamic scheduling and resource allocation policies for workloads where performance of the underlying hardware resource is not known ahead.

While our work assumes finishing a large number of tasks that can be processed in batches, there are also many prior work in real time systems that study optimal latency scheduling problems for dynamic real-time system tasks that arrive dynamically or in a periodic fashion. They use online scheduling algorithms. [ZR87] studies the

problem of scheduling n tasks in a system with r resources. Each task has a processing time T_P, deadline T_D and required resources. The paper discusses dynamic real-time systems where tasks arrive dynamically, and new decisions are made regarding whether the newly arriving tasks are schedulable—while guaranteeing that the previously scheduled tasks are retained. The online scheduling algorithm used in such dynamic real-time systems is important in real-time application environments. Experiments show that a few linear combinations of heuristics achieve very low optimality gaps to the optimal algorithms that do exhaustive search. [ZRS87] studies tasks that have hard real-time limits. While most prior work and also our work assume complete and accurate prior knowledge of tasks ahead, the paper develops a flexible scheduling algorithm where prior information is not needed and the loosely coupled distributed system dynamics are also taken into consideration. The basic approach proposed in the paper assumes that each node has a local scheduler, a bidder and a dispatcher. The local scheduler tries to make sure that a new task can finish before the deadline constraint. If an estimate is made that the tasks will not be finished on time, the bidder on the node then sends a request for the tasks, and also evaluates bid requests from other nodes, and then sends the tasks to the best bidder node. The node also has a dispatcher that schedules the guaranteed tasks. In this work, tasks are independent and resources are assumed to be always available to the executing tasks, including CPU and I/O storage. The algorithm proposed focuses on an extension of the local scheduler portion of the hard real-time scheduling scheme, and it is applied to perform scheduling for tasks that are in a batch, or in a periodic fashion. [RSZ89] proposes a set of heuristic algorithms that dynamically schedule essential tasks considering deadlines and resource constraints in distributed systems. Essential tasks are tasks that have deadlines. If deadlines are not met, system performance becomes severely

impaired. While much prior work assume that users have prior knowledge of task runtime and use static scheduling and allocation strategies, the paper addresses inflexibility, low resource utilization, incompatibility of deadline and task priority problems from static scheduling. Their proposed scheduling guarantees that the local scheduler first attempts to finish the task before its deadline within the node. If it fails, four heuristic corrective methods are proposed to let other nodes cooperate with the local node to finish the task, and communication delay is considered. The methods achieve the highest guarantee ratio—that is, the number of tasks are finished before the deadlines among all tasks.

CHAPTER 9

Conclusions

This dissertation discusses various design targets, analytic modelings, and summarizes optimization problems for field programmable gate array (FPGA) based customized computing at chip level, node level and cluster level. FPGAs have gained popularity in the acceleration of a broad range of applications with 10x-100x performance/energy efficiency over the general-purpose processors. The design spaces of FPGA accelerators at different levels for different design targets are enormous. Modeling is inevitable for designers to optimize energy consumption, performance, and cost given certain constraints. At chip-level, we first study the energy efficiency of full pipelining and propose performance and energy modeling for chip-level design. Then we propose the Latte microarchitecture to address timing degradation problems in the high-level synthesis (HLS-based) accelerator design. At node level, We propose a Doppio framework to understand performance and cost models for customized computing in light of the fact that performance and cost are primary concerns when deploying applications and services in a pay-as-you-go public cloud. The performance and cost modeling are discussed in two aspects—computation resources, with CPUs and locally PCIe-attached accelerators, and storage resources including SSDs and HDDs. At the cluster level, we propose the Mocha framework as a distributed runtime system to share one FPGA among multiple CPU instances, or co-schedule accelerated kernels

on CPUs and FPGAs to improve underlying hardware utilization in order to optimize the out-of-pocket cost. To demonstrate the performance improvement and cost saving of modeling in customized computing, we use genome pipeline optimization in both the public cloud and private cloud as case studies showing how to conduct optimal scheduling under certain deadline constraints or hardware (CPUs, storage) constraints.

References

[ABV16] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. "Architectural Impact on Performance of In-memory Data Analytics: Apache Spark Case Study." *CoRR*, **abs/1604.08484**, 2016.

[ACH13] Geraldine A Van der Auwera, Mauricio O Carneiro, Christopher Hartl, Ryan Poplin, Guillermo Del Angel, Ami Levy-Moonshine, Tadeusz Jordan, Khalid Shakir, David Roazen, Joel Thibault, et al. "From FastQ data to high-confidence variant calls: the genome analysis toolkit best practices pipeline." *Current protocols in bioinformatics*, **43**(1):11–10, 2013.

[ALC17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics." In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[Alpdf] AlphaData. "ADM-PCIE7V3.", https://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf.

[Amaa] Amazon. "Amazon EC2 F1 instance." https://aws.amazon.com/ec2/instance-types/f1/.

[Amab] Amazon. "AWS Spot Instance Interruptions." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-interruptions.html.

[Amac] Amazon. "AWS Spot Instance Pricing History." https://us-west-1.console.aws.amazon.com/ec2sp/v1/spot/home?region=us-west-1.

[Amad] Amazon. "AWS Spot Instances Advisor." https://aws.amazon.com/ec2/spot/instance-advisor/.

[Amae] Amazon. "AWS Spot Instances Pricing." https://aws.amazon.com/ec2/spot/pricing/.

[Ama19a] Amazon. "Amazon EC2 Placement Groups." https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html, 2019. [Online; accessed Jan-2019].

[Ama19b] Amazon. "AWS EC2 Pricing." https://aws.amazon.com/ec2/pricing/on-demand/, 2019.

[amac2] "Amazon EC2.", https://aws.amazon.com/ec2.

[AP15] Sungyong Ahn and Sangkyu Park. "An Analytical Approach to Evaluation of SSD Effects under MapReduce Workloads." *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, **15**(5):511–518, 2015.

[ATT10] HT Anson, David B Thomas, Kuen Hung Tsoi, and Wayne Luk. "Dynamic scheduling Monte-Carlo framework for multi-accelerator heterogeneous clusters." In *2010 International Conference on Field-Programmable Technology*, pp. 233–240. IEEE, 2010.

[Awa16] Ahsan Javed Awan et al. "How Data Volume Affects Spark Based Data Analytics on a Scale-up Server." In *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer International Publishing, 2016.

[BBG13] Mehmet E Belviranli, Laxmi N Bhuyan, and Rajiv Gupta. "A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures." *ACM Transactions on Architecture and Code Optimization (TACO)*, **9**(4):57, 2013.

[BCM18] BCM. "HGSC Resources." https://www.hgsc.bcm.edu/resources, 2018. [Online; accessed Jan-2019].

[BET17] Subho S Banerjee, Mohamed El-Hadedy, Ching Y Tan, Zbigniew T Kalbarczyk, Steve Lumetta, and Ravishankar K Iyer. "On accelerating pair-HMM computations in programmable hardware." In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8. IEEE, 2017.

[BLB11] Marcin Bogdanski, Peter R Lewis, Tobias Becker, and Xin Yao. "Improving scheduling techniques in heterogeneous systems with dynamic, on-line optimisations." In *2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, pp. 496–501. IEEE, 2011.

[BM11] Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. "HCOC: a cost optimization algorithm for workflow scheduling in hybrid clouds." *Journal of Internet Services and Applications*, **2**(3):207–227, 2011.

[BM13] James K Bonfield and Matthew V Mahoney. "Compression of FASTQ and SAM format sequencing data." *PloS one*, **8**(3):e59190, 2013.

[Bri] Simon Briggs. "Cheapest Amazon EC2 Spot Price Region." https://simonpbriggs.co.uk/amazonec2/.

[BSH18] Mohammed S Bensaleh, Yaman Sharaf-Dabbagh, Hazem Hajj, Mazen AR Saghir, Haitham Akkary, Hassan Artail, Abdulfattah M Obeid, and Syed Manzoor Qasim. "Optimal Task Scheduling for Distributed Cluster With Active Storage Devices and Accelerated Nodes." *IEEE Access*, **6**:48195–48209, 2018.

[BT13] Sam Behjati and Patrick S Tarpey. "What is next generation sequencing?" *Archives of Disease in Childhood-Education and Practice*, **98**(6):236–238, 2013.

[CCP16] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. "A cloud-scale acceleration architecture." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, Oct 2016.

[CCS10] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser N Tantawi, and Chandra Krintz. "See spot run: using spot instances for mapreduce workflows." *HotCloud*, **10**:7–7, 2010.

[cds17] "Google Cloud Disk Profile." [url deleted to maintain the integrity of the review process], 2017.

[CFG09] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants." *Nucleic acids research*, **38**(6):1767–1771, 2009.

[CFH04] J. Cong, Yiping Fan, Guoling Han, Xun Yang, and Zhiru Zhang. "Architecture and synthesis for on-chip multicycle communication." *TCAD*, pp. 550–564, April 2004.

[CFH18] J. Cong, Z. Fang, M. Huang, L. Wang, and D. Wu. "CPU-FPGA Coscheduling for Big Data Applications." *IEEE Design Test*, **35**(1):16–22, Feb 2018.

[Che13] R. Chen et al. "Energy efficient parameterized FFT architecture." In *FPL*, 2013.

[CHM14] J. Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. "A Fully Pipelined and Dynamically Composable Architecture of CGRA." In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pp. 9–16, May 2014.

[CHW] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. "Invited - Heterogeneous Datacenters: Options and Opportunities." In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16.

[CJL11] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization." *ACM Trans. Des. Autom. Electron. Syst.*, **16**(2):15:1–15:25, April 2011.

[CLC13] Kristian Cibulskis, Michael S Lawrence, Scott L Carter, Andrey Sivachenko, David Jaffe, Carrie Sougnez, Stacey Gabriel, Matthew Meyerson, Eric S Lander, and Gad Getz. "Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples." *Nature biotechnology*, **31**(3):213, 2013.

[Clo13] Cloudera. "How-to: Select the Right Hardware for Your New Hadoop Cluster." http://blog.cloudera.com/blog/2013/08/how-to-select-the-right-hardware-for-your-new-hadoop-cluster, 2013.

[CLS08] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach. "Accelerating Compute-Intensive Applications with GPUs and FPGAs." In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 101–107, June 2008.

[CMC16] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. "Scaling Spark on HPC Systems." In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pp. 97–110, New York, NY, USA, 2016. ACM.

[CO16] Tatsuhiro Chiba and Tamiya Onodera. "Workload characterization and optimization of TPC-H queries on Apache Spark." In *ISPASS*, April 2016.

[Com18] Falcon Computing. "Enabling Faster, More Cost-Eective Genomics Analytics Through Heterogeneous Computing Solutions." https://www.falconcomputing.com/download/cost-effective-genomics-analytics-white-paper/, 2018. [Online; accessed May-2019].

[CSR11] J. Cong, V. Sarkar, G. Reinman, and A. Bui. "Customizable Domain-

Specific Computing." *IEEE Design Test of Computers*, **28**(2):6–15, March 2011.

[Cut15] Shannon Cutt. "Investigating Sparks performance." https://www.oreilly.com/ideas/investigating-sparks-performance, 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] J. Cong, P. Wei, C. H. Yu, and P. Zhou. "Latte: Locality Aware Transformation for High-Level Synthesis." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 125–128, April 2018.

[CWY18b] 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.

[CYK15] I. S. Choi, W. Yang, and Y. S. Kee. "Early experience with optimizing I/O performance using high-performance SSDs for in-memory cluster computing." In *2015 IEEE International Conference on Big Data (Big Data)*, pp. 1073–1083, Oct 2015.

[Dar10] Darin. "Can't Stop Love." Universal Music, 2010.

[DBP11] Mark A DePristo, Eric Banks, Ryan Poplin, Kiran V Garimella, Jared R Maguire, Christopher Hartl, Anthony A Philippakis, Guillermo Del Angel,

Manuel A Rivas, Matt Hanna, et al. "A framework for variation discovery and genotyping using next-generation DNA sequencing data." *Nature genetics*, **43**(5):491, 2011.

[DeH15] André DeHon. "Fundamental Underpinnings of Reconfigurable Computing Architectures." *Proceedings of the IEEE*, **103**(3):355–378, March 2015.

[DEK98] Richard Durbin, Sean R Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[DG08a] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM*, 2008.

[DG08b] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM*, **51**(1):107–113, 2008.

[DJ09] Cagdas Dirik and Bruce Jacob. "The Performance of PC Solid-state Disks (SSDs) As a Function of Bandwidth, Concurrency, Device Architecture, and System Organization." In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 279–289, New York, NY, USA, 2009. ACM.

[DK12] A. Duran and M. Klemm. "The Intel Many Integrated Core Architecture." In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pp. 365–366, July 2012.

[DK13] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-Aware

Scheduling for Heterogeneous Datacenters." In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013.

[DK14] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management." In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[DK16] Christina Delimitrou and Christos Kozyrakis. "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems." *SIGARCH Comput. Archit. News*, **44**(2):473–488, March 2016.

[DR13] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[DSL08] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. "The cost of doing science on the cloud: the montage example." In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–12. Ieee, 2008.

[GAB15] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. "Scheduling the I/O of HPC Applications Under Congestion." In *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 1013–1022, May 2015.

[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*, 1998.

[GMP10] Alan E Guttmacher, Amy L McGuire, Bruce Ponder, and Kári Stefánsson. "Personalized genomic information: preparing for the future of genetic medicine." *Nature Reviews Genetics*, 2010.

[Goo19] Google. "Protocal Buffer." https://developers.google.com/protocol-buffers/, 2019. [Online; accessed Jan-2019].

[GRB17] Eugenio Gianniti, Alessandro Maria Rizzi, Enrico Barbierato, Marco Gribaudo, and Danilo Ardagna. "Fluid Petri Nets for the Performance Evaluation of MapReduce and Spark Applications." *SIGMETRICS Perform. Eval. Rev.*, **44**(4):23–36, May 2017.

[HAL16] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R.W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. "Scalable I/O-Aware Job Scheduling for Burst Buffer Enabled HPC Clusters." In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pp. 69–80, New York, NY, USA, 2016. ACM.

[HCT18] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R Ganger, and Phillip B Gibbons. "Tributary: spot-dancing for elastic services with latency SLOs." In *2018 USENIX Annual Technical Conference*. USENIX Association, 2018.

[HMR17] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W Hwu, and Deming Chen. "Hardware ac-

celeration of the pair-HMM algorithm for DNA variant calling." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 275–284. ACM, 2017.

[Hua19] Huawei. "Elastic Cloud Server Price Details." https://www.huaweicloud.com/en-us/price_detail.html#/ecs_detail, 2019.

[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*, SoCC '16, 2016.

[IBM18a] IBM. "CPLEX for Python users." https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cplex.help/CPLEX/UsrMan/topics/APIs/Python/01_title_synopsis.html, 2018. [Online; accessed May-2019].

[IBM18b] IBM. "MIP features in LP file format." https://www.ibm.com/support/knowledgecenter/SSSA5P_12.9.0/ilog.odms.cplex.help/CPLEX/FileFormats/topics/LP_MIP.html, 2018. [Online; accessed May-2019].

[Ill19] Illumina. "Illumina Downloads." https://support.illumina.com/downloads.html, 2019. [Online; accessed Jan-2019].

[Insa] Broad Institute. "Broad Institute sequences its 100,000th whole human genome on National DNA Day." https://www.broadinstitute.org/news.

[Insb] Broad Institute. "Introduction to the GATK Best Practices." https://software.broadinstitute.org/gatk/best-practices/.

[Ins17a] Broad Institute. "Base Quality Score Recalibration (BQSR)Methods and Algorithms." *The Broad Institute*, 2017.

[Ins17b] Broad Institute. "GATK." https://github.com/broadinstitute/gatk, 2017.

[Ins19] Broad Institute. "Genome Analysis Toolkit HaplotypeCaller." https://software.broadinstitute.org/gatk/documentation/tooldocs/3.8-0/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php, 2019.

[Int] Intel. "Intel FPGA SDK for OpenCL." http://www.altera.com/.

[IO16] Megumi Ito and Moriyoshi Ohara. "A power-efficient FPGA accelerator: Systolic array with cache-coherent interface for pair-HMM algorithm." In *Low-Power and High-Speed Chips (COOL CHIPS XIX), 2016 IEEE Symposium in*, pp. 1–3. IEEE, 2016.

[IPS16] Anca Iordache, Guillaume Pierre, Peter Sanders, Jose Gabriel de F Coutinho, and Mark Stillwell. "High performance in the cloud with FPGA groups." In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pp. 1–10. ACM, 2016.

[JCP05] Ju wook Jang, Seonil Choi, and Viktor K. Prasanna. "Energy-Efficient Matrix Multiplication on FPGAs." *TVLSI*, **13**(11):1305–1319, November 2005.

[KC14] Karthik Kambatla and Yanpei Chen. "The Truth About MapReduce Performance on SSDs." In *28th LISA*. USENIX Association, November 2014.

[KLD15] Edin Kadric, David Lakata, and André DeHon. "Impact of Memory Architecture on FPGA Energy Consumption." In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pp. 146–155, New York, NY, USA, 2015. ACM.

[KLK18] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. "Selecta: heterogeneous cloud storage configuration for data analytics." In *2018 USENIX Annual Technical Conference*, pp. 759–773. USENIX Association, 2018.

[KMP12] Mihail N. Kolountzakis, Gary L. Miller, Richard Peng, and Charalampos E. Tsourakakis. "Efficient Triangle Counting in Large Graphs via Degree-Based Vertex Partitioning." *Internet Mathematics*, **8**(1-2):161–185, 2012.

[Koh16] Christopher Kohlhoff. "Boost. asio." *Online: http://www. boost. org/-doc/libs/1*, **48**(0):2003–2013, 2016.

[Lam88] M. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines." In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pp. 318–328, New York, NY, USA, 1988. ACM.

[Lei80] Charles E. Leiserson. "Area-efficient graph layouts (for VLSI)." In *Foundations of Computer Science, 1980., 21st Annual Symposium on*, pp. 270–281, Oct 1980.

[LHW09] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, et al. "The sequence alignment/map format and SAMtools." *Bioinformatics*, **25**(16):2078–2079, 2009.

[LJS89] Wayne Luk, Geraint Jones, and Mary Sheeran. "Computer-Based Tools For Regular Array Design." In *Systolic Array Processors*, 1989.

[LLH05] Fei Li, Yan Lin, Lei He, Deming Chen, and J. Cong. "Power modeling and characteristics of field programmable gate arrays." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **24**(11):1712–1724, Nov 2005.

[LR71] B.S. Landman and Roy L. Russo. "On a Pin Versus Block Relationship For Partitions of Logic Graphs." *Computers, IEEE Transactions on*, **C-20**(12):1469–1479, Dec 1971.

[LTW15] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. "SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark." CF. ACM, 2015.

[LVC18] Yuk Yee Leung, Otto Valladares, Yi-Fan Chou, Han-Jen Lin, Amanda B Kuzma, Laura Cantwell, Liming Qu, Prabhakaran Gangadharan, William J Salerno, Gerard D Schellenberg, et al. "VCPA: genomic variant calling pipeline and data management tool for Alzheimers Disease Sequencing Project." *Bioinformatics*, **35**(10):1768–1770, 2018.

[LZP15] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. "Resource-Aware Throughput Optimization for High-Level Synthesis." In *Pro-*

ceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15, pp. 200–209, New York, NY, USA, 2015. ACM.

[Man15] Zoltán Ádám Mann. "Allocation of Virtual Machines in Cloud Data Centers&Mdash;A Survey of Problem Models and Optimization Algorithms." *ACM Comput. Surv.*, **48**(1):11:1–11:34, August 2015.

[MBY16] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. "Mllib: Machine learning in apache spark." *Journal of Machine Learning Research*, **17**(34):1–7, 2016.

[MH11a] M. Mao and M. Humphrey. "Auto-scaling to minimize cost and meet application deadlines in cloud workflows." In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov 2011.

[MH11b] Ming Mao and Marty Humphrey. "Auto-scaling to minimize cost and meet application deadlines in cloud workflows." In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pp. 1–12. IEEE, 2011.

[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*, 2010.

[MJD15] Maciej Malawski, Gideon Juve, Ewa Deelman, and Jarek Nabrzyski. "Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds." *Future Generation Computer Systems*, **48**:1–18, 2015.

[NKS12] Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. "Hadoop-BAM: directly manipulating next generation sequencing data in the cloud." *Bioinformatics*, **28**(6):876–877, 2012.

[NST18] Hiroki Nishikawa, Kana Shimada, Ittetsu Taniguchi, and Hiroyuki Tomiyama. "Scheduling of Malleable Fork-Join Tasks with Constraint Programming." In *2018 Sixth International Symposium on Computing and Networking (CANDAR)*, pp. 133–138. IEEE, 2018.

[ORR15] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks." In *NSDI*. USENIX Association, May 2015.

[PBM99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The PageRank Citation Ranking: Bringing Order to the Web." Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[Pla17] Google Cloud Platform. "Storage Options." https://cloud.google.com/compute/docs/disks/, 2017.

[PM13] Mihai Pricopi and Tulika Mitra. "Task scheduling on adaptive multi-core." *IEEE transactions on Computers*, **63**(10):2590–2603, 2013.

[PRA16] Johan Peltenburg, Shanshan Ren, and Zaid Al-Ars. "Maximizing systolic array efficiency to accelerate the PairHMM forward algorithm." In *Bioinformatics and Biomedicine (BIBM), 2016 IEEE International Conference on*, pp. 758–762. IEEE, 2016.

[PWY05] Kara K. W. Poon, Steven J. E. Wilton, and Andy Yan. "A Detailed Power Model for Field-programmable Gate Arrays." *ACM Trans. Des. Autom. Electron. Syst.*, **10**(2):279–302, April 2005.

[PYC06] Hee Kong Phoon, M. Yap, and Chuan Khye Chai. "A Highly Compatible Architecture Design for Optimum FPGA to Structured-ASIC Migration." In *Semiconductor Electronics, 2006. ICSE '06. IEEE International Conference on*, pp. 506–510, Oct 2006.

[QDF18] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. "High-throughput lossless compression on tightly coupled CPU-FPGA platforms." In *FCCM*, pp. 37–44, 2018.

[RA11] S.T. Rajavel and A. Akoglu. "An analytical energy model to accelerate FPGA logic architecture investigation." In *Field-Programmable Technology (FPT), 2011 International Conference on*, pp. 1–8, Dec 2011.

[rosrg] "ROSE Compiler Infrastructure.", http://rosecompiler.org/.

[RPA16] Chris Rauer, George Powley, Mir Ahsan, and Nicholas Finamore. "Accelerating Genomics Research with OpenCL and FPGAs." *Altera, Now Part of Intel, Tech. Rep*, 2016.

[RSA15] Shanshan Ren, Vlad-Mihai Sima, and Zaid Al-Ars. "FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis." In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pp. 1465–1470. IEEE, 2015.

[RSZ89] Krithi Ramamritham, John A. Stankovic, and Wei Zhao. "Distributed scheduling of tasks with deadlines and resource requirements." *IEEE Transactions on Computers*, **38**(8):1110–1123, 1989.

[Ruc15] E. Rucci et al. "Smith-Waterman Protein Search with OpenCL on an FPGA." In *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[SAA11] Khaled Salah, M Al-Saba, M Akhdhor, O Shaaban, and MI Buhari. "Performance evaluation of popular Cloud IaaS providers." In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pp. 345–349. IEEE, 2011.

[SKR10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System." In *MSST*, May 2010.

[SLF15] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. "Big data: astronomical or genomical?" *PLoS Biol*, **13**(7):e1002195, 2015.

[SM91] GN Srinivasa Prasanna and Bruce R Musicus. "Generalised multiprocessor scheduling using optimal control." In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pp. 216–228. ACM, 1991.

[Sni98] Marc Snir. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[Spa17a] Spark. "Apache Spark Shuffle Operations." http://spark.apache.org/docs/latest/programming-guide.html#shuffle-operations, 2017.

[Spa17b] Apache Spark. "Apache Spark Shuffle Operations Performance Impact." http://spark.apache.org/docs/latest/programming-guide.html#performance-impact, 2017.

[Spa17c] Apache Spark. "Hardware Provisioning." http://spark.apache.org/docs/latest/hardware-provisioning.html, 2017.

[Spa17d] Apache Spark. "Triangle Count in Spark." https://github.com/apache/spark/blob/master/graphx/src/main/scala/org/apache/spark/graphx/lib/TriangleCount.scala, 2017.

[SRF12] Thomas RW Scogland, Barry Rountree, Wu-chun Feng, and Bronis R De Supinski. "Heterogeneous task scheduling for accelerated openmp." In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pp. 144–155. IEEE, 2012.

[Sta16] Standford. "GIAB Resources." http://jimb.stanford.edu/giab-resources/, 2016. [Online; accessed Jan-2019].

[SV99] Johan AK Suykens and Joos Vandewalle. "Least squares support vector machine classifiers." *Neural processing letters*, **9**(3):293–300, 1999.

[SV11] Siddharth Suri and Sergei Vassilvitskii. "Counting Triangles and the Curse of the Last Reducer." In *Proceedings of the 20th International*

Conference on World Wide Web, WWW '11, pp. 607–614, New York, NY, USA, 2011. ACM.

[Tay12] Michael B. Taylor. "Is Dark Silicon Useful? Harnessing the Four Horesemen of the Coming Dark Silicon Apocalypse." In *Design Automation Conference*, 2012.

[TBN07] R. Tessier, V. Betz, D. Neto, A. Egier, and T. Gopalsamy. "Power-Efficient RAM Mapping Algorithms for FPGA Embedded Memory Blocks." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**(2):278–290, Feb 2007.

[TLF17] Naif Tarafdar, Thomas Lin, Eric Fukuda, Hadi Bannazadeh, Alberto Leon-Garcia, and Paul Chow. "Enabling flexible network FPGA clusters in a heterogeneous cloud data center." In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 237–246. ACM, 2017.

[TSP16] Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Ioannis Koltsidas, and Nikolas Ioannou. "On The [Ir]relevance of Network Performance for Data Processing." In *HotCloud*, June 2016.

[TT14] A. Tavakkoli and D. B. Thomas. "Low-latency option pricing using systolic binomial trees." In *FPT*, 2014.

[Tuc17] L. Di Tucci et al. "Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL." In *DATE*, 2017.

[UCL16] UCLA-VAST. "Blaze: Deploying Accelerators at Datacenter Scale." https://github.com/UCLA-VAST/blaze, 2016.

[VVB10] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove. "Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads." In *2010 IEEE 3rd International Conference on Cloud Computing*, pp. 228–235, July 2010.

[VYF16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. "Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics." In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, March 2016.

[WAH15] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf. "Enabling FPGAs in Hyperscale Data Centers." In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pp. 1078–1086, Aug 2015.

[Wan16] Z. Wang et al. "A performance analysis framework for optimizing OpenCL applications on FPGAs." In *HPCA*, 2016.

[Wan17] S. Wang et al. "FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs." In *DAC*, 2017.

[Wan19] Jie Wang. "PairHMM using shared memory/shuffle instructions." https://github.com/whbldhwj/IPDPS, 2019. [Online; accessed Jan-2019].

[Whi12a] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[Whi12b] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[WK15] K. Wang and M. M. H. Khan. "Performance Prediction for Apache Spark Platform." HPCC-CSS-ICESS '15. IEEE Computer Society, 2015.

[WXC17] Jie Wang, Xinfeng Xie, and Jason Cong. "Communication Optimization on GPU: A Case Study of Sequence Alignment Algorithms." In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pp. 72–81. IEEE, 2017.

[WXW16] Li Wang, Tianni Xu, Jing Wang, Weigong Zhang, Xiufeng Sui, and Yungang Bao. "Understanding the Behavior of Spark Workloads from Linux Kernel Parameters Perspective." In *17th International Middleware Conference*, Middleware Posters and Demos '16, pp. 1–2, New York, NY, USA, 2016.

[WZH16] Zeke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. "Melia: A mapreduce framework on opencl-based fpgas." *IEEE Transactions on Parallel and Distributed Systems*, **27**(12):3547–3560, 2016.

[WZL14] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. "BigDataBench: A big data benchmark suite from internet services." In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*, pp. 488–499, 2014.

[XGF13] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "GraphX: A Resilient Distributed Graph System on Spark." In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pp. 2:1–2:6, New York, NY, USA, 2013. ACM.

[Xila] Xilinx. "Vivado High-Level Synthesis.".

[Xilb] Xilinx. "Xilinx UltraScale+ MPSoC." https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[Xilc] Xilinx 7 Series DSP48E1 Slice User Guide.

[Xild] Xilinx 7 Series FPGAs Memory Resources.

[YHG17] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach." In *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 452–465. ACM, 2017.

[YWZ15] J. Yin, J. Wang, J. Zhou, T. Lukasiewicz, D. Huang, and J. Zhang. "Opass: Analysis and Optimization of Parallel Data Access on Distributed File Systems." In *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 623–632, May 2015.

[ZCD12a] 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 *NSDI*, Berkeley, CA, USA, 2012.

[ZCD12b] 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, pp. 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[ZCL15] Jianfeng Zhang, Paul Chow, and Hengzhu Liu. "CORDIC-Based Enhanced Systolic Array Architecture for QR Decomposition." *TRETS*, 2015.

[Zho16] P. Zhou et al. "Energy Efficiency of Full Pipelining: A Case Study for Matrix Multiplication." In *FCCM*, 2016.

[Zoh16] H. R. Zohouri et al. "Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs." In *SC*, 2016.

[ZPF16] P. Zhou, H. Park, Z. Fang, J. Cong, and A. DeHon. "Energy Efficiency of Full Pipelining: A Case Study for Matrix Multiplication." In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 172–175, May 2016.

[ZR87] Wei Zhao and Krithi Ramamritham. "Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints." *Journal of Systems and Software*, **7**(3):195–205, 1987.

[ZRF18a] P. Zhou, Z. Ruan, Z. Fang, D. Roazen, M. Shand, and J. Cong. "Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-

Memory Computing Framework." In *2018 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 1–11, April 2018.

[ZRF18b] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. "Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-Memory Computing Framework." In *Performance Analysis of Systems and Software (ISPASS), 2018 IEEE International Symposium on*, pp. 22–32. IEEE, 2018.

[ZRS87] Wei Zhao, Krithi Ramamritham, and John A. Stankovic. "Scheduling tasks with resource requirements in hard real-time systems." *IEEE transactions on software engineering*, (5):564–577, 1987.

[ZZR16] Z. Zhuang, S. Zhuk, H. Ramachandra, and B. Sridharan. "Designing SSD-Friendly Applications for Better Application Performance and Higher IO Efficiency." In *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, June 2016.

[ZZT13] Xingquan Zuo, Guoxiang Zhang, and Wei Tan. "Self-adaptive learning PSO-based deadline constrained task scheduling for hybrid IaaS cloud." *IEEE Transactions on Automation Science and Engineering*, **11**(2):564–573, 2013.