Title
Raising an Abstraction Level of Compilation and Optimization for Customized Computing

Author
Yu, Hao

Publication Date
2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Raising an Abstraction Level of Compilation and Optimization for Customized Computing

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

by

Hao Yu

2019

ABSTRACT OF THE DISSERTATION

Raising an Abstraction Level of Compilation and Optimization
for Customized Computing

by

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

The demand for scalable, high-performance computing has increased as the size of datasets has grown in recent years. However, the breakdown of Dennard's scaling [DGR74] has led to energy efficiency becoming an important concern in datacenters, and spawned exploration into using power-efficient processors such as GPUs (Graphic Processing Units) and FPGAs (Field-Programmable Gate Arrays) as accelerators in datacenters. In particular, the FPGA's low power consumption and the re-programmability allow datacenters to use FPGAs as highly energy-efficient accelerators for a variety of application. On the other hand, FPGA has poor programmability compared to instructions-based architectures like CPU and GPU. To facilitate the process of implementing and deploying FPGA accelerators, High-Level Synthesis (HLS) [CLN11] that generates functional-equivalent RTL from C-based programming languages attracts more and more attention since past decades. Nowadays, both FPGA vendors have their commercial HLS products – Xilinx SDx [SDX] and Intel FPGA SDK for OpenCL [INT]. However, modern HLS is still not friendly

for software designers who have limited FPGA domain knowledge. Since the hardware architecture inferred from a syntactic C implementation could be ambiguous, current commercial HLS tools usually generate architecture structures according to specific HLS C code patterns. As a result, even though the authors in [CLN11] have illustrated that the HLS tool is capable of generating FPGA designs with competitive performance as the one in RTL, designers must manually reconstruct the HLS C kernel with specific code patterns to achieve high performance. This problem becomes one of the main impediments to consolidating the FPGA community on cooperation and developments.

In this dissertation, we first present an automated framework that frees human efforts from code reconstruction and design space exploration (DSE). The framework creates a more comprehensive micro-architecture design space from user-written C-based kernel with the Merlin compiler [CHP16a], so the design point should cover the design point with better performance when compared to the HLS-pragma-based design space. To efficiently identify the best design configuration in the tremendous design space, we first propose efficient design space pruning processes that reduce the design space by 24.65×. Accordingly, we develop and evaluate several approaches, including multi-armed bandit hyper heuristic approach, gradient-based approach, and design bottleneck optimization approach. The evaluation result shows that our DSE framework is able to identify the design point that achieves on average (using geometric mean) 93.78% QoR compared to the corresponding manual design.

Based on the proposed DSE framework, we further support automated design optimization for high level domain specific languages (DSLs). Since DSLs might not explicitly provide interfaces for users to specify design configurations, automatic DSE becomes even more important when supporting DSLs for FPGAs. Specifi-

cally, we adopt Merlin C [CHP16a], an OpenMP-like [OMP] C-based programming model, as the intermediate representation (IR) and implement DSL-to-Merlin front-end compilers while preserving the semantic and domain-specific information such as parallel patterns, systolic patterns, and scheduling functions. We first implement Spark-to-Merlin front-end compiler that translates Spark applications in Scala to Merlin C for FPGA acceleration. By leveraging parallel patterns as scheduling hints, the generated accelerators are able to achieve 50× speedup on geometric mean for a set of machine learning kernels. In addition, we also demonstrate that our DSE framework can be even more practical for the DSLs with plenty scheduling functions. Specifically, we implement HeteroCL-to-Merlin front-end that takes HeteroCL [LCH19] programming model embedded in Python. Our DSE framework is capable of exploring a subset of HeteroCL scheduling primitives and let users focus on the platform independent loop transformations. With the help from the DSE framework, we achieve 27.62× speedup on geometric mean over a CPU core for a variety of compute-intensive kernels (chapter 3).

On the other hand, a main challenge of performing design space exploration for a design with arbitrary functionality is the lack of the assumption to underlying micro-architectures. As we will illustrate in the dissertation, the cost of evaluating the quality of a design point is extremely expensive (15-60 minutes) so only a limited number of design points can be explored. In addition, due to the uncertainty of vendor tool behaviors, the development of performance and resource modeling is also unrealistic. As a result, we propose composable, parallel and pipeline (CPP) architecture template to limit the design space to a certain region that is more practical and has less exceptions (chapter 4). With the CPP architecture, we are able to derive an incremental analytical model, which only requires a few HLS run to

be initialized, to facilitate the DSE process. In the last part of this dissertation, we use convolutional neural network (CNN) to demonstrate that the HLS runtime cost can be totally saved with the use of a more domain specific architecture (chapter 5). Specifically, we leverage a systolic array architecture template for CNN accelerator generation. By mapping a CNN model to the pre-defined systolic array template, we can guarantee the model accuracy and DSE efficiency. The experimental result shows that our analytical model for the architecture template achieves 96% accuracy, and the mapped CNN model achieves up to 1.2 Tops throughput on an Intel Arria 10 FPGA.

The dissertation of Hao Yu is approved.

Glenn D. Reinman

Jens Palsberg

Quanquan Gu

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2019

To my family and fiancée.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Pursing a Ph.D. degree has always been my dream and I feel fortunate that I could make it happen. I would like to first express my sincerest gratitude to my advisor, Professor Jason Cong, for his trust, constant support, professional advice, and guidance during my Ph.D. life at UCLA. As a new Ph.D. student with computer-aided design background but not high-level synthesis and FPGAs, Professor Cong guides me into this field. More importantly, the way to systematically address research problems I learned from Professor Cong levels up my research capability and would be one of the most valuable things I got during my research life at UCLA. It is my great honor to have Professor Cong as my advisor.

I wish to express my appreciation to my doctoral committee members, Professor Tyson Condie, Professor Quanquan Gu, Professor Jens Palsberg, Professor Glenn Reinman, for their interest, patient, and advise to improve the quality of this dissertation. I especially wish to thank Professor Palsberg for his inspiration to the Spark to FPGA compilation framework.

I wish to thank the staffs at UCLA that help me a lot during this time. Thanks Alexandra Loung for her help on various paper works and service. Thanks Janice Martin-Wheeler for her help on editing the language of my papers. I cannot imagine what my Ph.D life would be without your helps.

I wish to thank to many of my colleagues for their supports and collaborations. I would like to thank Yu-Ting Chen for his recommendation to this research group and his guidance to research, career, and life experience. Thanks Peng Wei for our collaborations that became a role model to other members in the group. The collaborated work such as CPP architecture contributes a part of this dissertation. Thanks also

go to Di Wu, Muhuan Huang, Peipei Zhou, Po-Tsang Huang, Max Grossman (Rice University), and Sean Lai (Cornell University) for their collaborations; Hui Huang, Bingjun Xiao, Young-Kyu Choi, Yuchen Hao, Zhenman Fang, Chen Zhang for technical discussions and shared experiences; Yuze Chi and Jie Wang for being not only co-workers together for a project but also great gossiping partners and roommates; Zhe Chen, Weikang Qiao, Zhenyuan Ruan, Tong He, Licheng Guo, Jason Lau, Karl Marrett, Atefeh Sohrabizadeh for being great lab mates.

In addition, most research in this dissertation are done when I was an intern at Falcon Computing Solutions, so I would also like to express my gratitude to all the sponsors and partners at Falcon, especially to Peng Zhang, Xuechao Wei, Han Hu and Youxiang Chen, and Yuxin Wang for their supports.

Finally, I would like to quote a speech (with some adaptions) given by Dr. Sheldon Copper from the very last episode of my favorite TV series, The Big Bang Theory, which also coincidentally finishes its journey in May 2019: *"I was under a misapprehension that my accomplishments were mine alone, but nothing could be further from the truth. I have been encouraged, sustained, inspired and tolerated not only by my family, but by the greatest fiancée Wan-Yu anyone ever had."*

The research studies in this dissertation are partially supported by the Center for Domain-Specific Computing under the NSF InTrans Award CCF-1436827, funding from CDSC industrial partners including Google, Huawei, Intel, Mentor Graphics and VMWare; C-FAR, one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA; grants from NIH U54EB020404 Big Data to Knowledge (BD2K) progam.

VITA

2011 B.S. (Computer Science), National Tsing Hua University, Taiwan.

2013 M.S. (Computer Science), National Tsing Hua University, Taiwan.

PUBLICATIONS

Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, Cody H. Yu, Y. Zhou, J. Cong, Z. Zhang, "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing", *FPGA*, 2019. **(Best paper award)**

J. Cong, Z. Fang, M. Huang, P. Wei, D. Wu, Cody H. Yu, "Customizable Computing–From Single Chip to Datacenters," *Proceedings of the IEEE*, 2019.

X. Wei, Y. Liang, X. Li, Cody H. Yu, P. Zhang, J. Cong, "TGPA: Tile-grained Pipeline Architecture for Low Latency CNN Inference," *ICCAD*, 2018.

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

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

J. Cong, P. Wei, Cody H. Yu, and P. Zhang, "Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture," *DAC*, 2018.

Cody H. Yu, P. Wei, P. Zhang, M. Grossman, V. Sarker, and J. Cong, "S2FA: An Accelerator Automation Framework for Heterogeneous Computing in Datacenters," *DAC*, 2018.

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

X. Wei, Cody H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, and J. Cong, "Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs," *DAC*, 2017. **(Best paper nominee)**

M. Huang, D. Wu, Cody H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale," *SoCC*, 2016.

J. Cong, M. Huang, D. Wu, and Cody H. Yu, "Invited - Heterogeneous Datacenters: Options and Opportunities," *DAC*, 2016. **(Invited)**

M.-C. Frank Chang, Y.-T. Chen, J. Cong, P.-T. Huang, C.-L. Kuo, and Cody H. Yu, "The SMEM Seeding Algorithm Acceleration for DNA Sequence Alignment," *FCCM*, 2016.

J. Cong, and Cody H. Yu, "Impact of Loop Transformations on Software Reliability," *ICCAD*, 2015.

CHAPTER 1

Introduction

The demand for scalable, high-performance computing has increased as the size of datasets has grown in recent years. In 2004, Google introduced the MapReduce [DG08] programming framework, a framework for efficiently managing tens of thousands to millions of servers in datacenters with a simple programming model. Inspired by Google MapReduce, open source big data analytic systems such as Apache Hadoop [HAD] and Spark [ZCF10] have developed and evolved rapidly. However, the breakdown of Dennard's scaling [DGR74] has led to energy efficiency becoming an important concern in datacenters, and spawned exploration into using power-efficient processors such as GPUs (Graphic Processing Units) and FPGAs (Field-Programmable Gate Arrays) as accelerators in datacenters.

In particular, the FPGA's low power consumption and the re-programmability allow datacenters to use FPGAs as highly energy-efficient accelerators for a variety of application. Applications with a large fraction of computationally-intensive kernels containing small amounts of control flow, such as string matching [CCL15], searching [PCC14] and sorting [CSP15, HLC14], are suitable to be accelerated using FPGAs. In addition, adopting FPGAs in private datacenters has recently garnered attention from the community. For example, IBM deploys FPGAs for its larger NoSQL data stores [BRH15]. Microsoft has adopted CPU-FPGA systems in its dat-

acenter to help accelerate the Bing search engine [PCC14]. Moreover, the acquisition of Altera by Intel in 2015 promises continued development of closely-integrated CPU-FPGA platforms; the compute instance with FPGAs in Amazon EC2 introduced in 2016 enables the FPGA platform in the public datacenter. As a result, datacenters with FPGAs are expected to be widely used in the near future.

On the other hand, FPGA has poor programmability compared to instructions-based architectures like CPU and GPU. Traditionally, Register-Transfer Level (RTL) description languages, such as VHDL and Verilog HDL, are the most widely used languages for FPGA design implementation. The use of these hardware description languages leads to the fact that the development concept of FPGA is circuit design instead of software implementation. According to the desired functionality, the designer comes up with a high-performance architecture, including finite state machine, data flow, and modules, and then implement the circuit in RTL. However, the design usually needs to be refined and improved iteratively, and each iteration takes a great deal of time and effort. To facilitate the process of implementing and deploying FPGA accelerators, High-Level Synthesis (HLS) [CLN11] that generates functional-equivalent RTL from C-based programming languages attracts more and more attention since past decades. Nowadays, both FPGA vendors have their commercial HLS products – Xilinx SDx [SDX] and Intel FPGA SDK for OpenCL [INT]. For example, Code 1.1 shows an intuitive HLS C implementation of Needleman-Wunsch (N-W) algorithm [NW70], a 2-D dynamic programming algorithm for string matching, for Xilinx FPGAs. Xilinx SDx is able to generate 9,694 lines of RTL kernel from Code 1.1 with the same functionality. As a result, it is much more efficient for designers to evaluate and improve their architectures in HLS C than RTL.

Code 1.1: N-W HLS C Code Snippet

```
1  void kernel(int batch, char seqAs[] ①, char seqBs[] ①,
2              char alignedAs[] ①, char alignedBs[] ①) {
3  #pragma HLS INTERFACE m_axi port=seqAs offset=slave
4  #pragma HLS INTERFACE m_axi port=seqBs offset=slave
5  #pragma HLS INTERFACE m_axi port=alignedAs offset=slave
6  #pragma HLS INTERFACE m_axi port=alignedBs offset=slave
7  #pragma HLS INTERFACE s_axilite port=seqAs offset=control
8  #pragma HLS INTERFACE s_axilite port=seqBs offset=control
9  #pragma HLS INTERFACE s_axilite port=alignedAs offset=control
10 #pragma HLS INTERFACE s_axilite port=alignedBs offset=control
11   for (int i=0; i<batch; i++) { ④
12     int M[129][129];
13     ...
14     for(i=0; i<129; i++) ⑤ { M[0][i]=seqAs[...] ② ③}
15     for(j=0; j<129; j++) ⑤ {M[j][0]=...}
16     for(i=1; i<129; i++) ⑤ {
17       for(j=1; j<129; j++) ⑤ { M[i][j]=... }
18     }
19     // Skip ~170 lines of N-W algorithm implementation.
20   }
21 }
```

The primary programming model of commercial HLS tools is based on pragmas. Specifically, users are required to insert tool-dependent pragmas to the kernel code properly in order to trigger certain optimization such as parallel and pipeline. Along with the pragma-based programming model, a number of research work [SW12a, PG02, HKR07, SW12a, MPZ12, XPZ15, SFP11, LC13, FAP18b, FAP18a] attempt to automate the process of identifying the best pragma combination, in terms of pragma positions and values, for user applications. However, all of them target to the design space formed by HLS pragmas, which may fail to cover high-performance design points in many cases (see Chapter 2 for details). The main reason that simply inserting HLS pragmas to user kernel code cannot achieve high-performance is that

modern commercial HLS tools generate architecture structures, such as dataflow, processing element replication, memory burst, and so forth, according to not only pragmas but also specific, clear HLS C code patterns, because the hardware architecture inferred from a syntactic C implementation could be ambiguous. As a result, even though the authors in [CLN11] have illustrated that the HLS tool is capable of generating FPGA designs with competitive performance as the one in RTL, designers must manually reconstruct the HLS C kernel with specific code patterns to achieve high performance, as demonstrated in [CFH18]. In fact, the generated FPGA accelerator from Code 1.1 is 5× slower than a single-thread CPU.

Code 1.2: N-W HLS C Code Snippet with Manual Optimization

```
1  void NW(...) {
2    int M[129][129];
3  #pragma HLS array_partition variable=M cyclic factor=4 dim=1
4    ...
5    for(i=0; i<129; i++) { ⑤
6  #pragma HLS pipeline
7  #pragma HLS unroll factor=8
8      M[0][i] = ..,;
9    }
10   for(j=0; j<129; j++) { ⑤
11 #pragma HLS pipeline
12 #pragma HLS unroll factor=8
13     M[j][0] = ...;
14   }
15   for(i=1; i<129; i++) {
16     for(j=1; j<129; j++) { ⑤
17 #pragma HLS pipeline
18 #pragma HLS unroll factor=8
19       M[i][j] = ...
20   }}
21   ...
22 }
23 void compute(char seqAs[32][16][8], char seqBs[32][16][8],
24              char alignedAs[32][8][32] char alignedBs[32][8][32]) {
```

```
#pragma HLS inline off
  for (int i=0; i<32; i++) { ④
#pragma HLS unroll
    NW(seqAs[i], seqBs[i], alignedAs[i], alignedBs[i]);
}}
void load(...) { ... } // off-chip data load
void store(...) { ... } // off-chip data store
void kernel(int batch, ap_uint<512> ① seqAs[], ap_uint<512> ① seqBs[],
           ap_uint<512> ① alignedAs[], ap_uint<512> ① alignedBs[]) {
#pragma HLS INTERFACE m_axi port=seqAs offset=slave
#pragma HLS INTERFACE m_axi port=seqBs offset=slave
#pragma HLS INTERFACE m_axi port=alignedAs offset=slave
#pragma HLS INTERFACE m_axi port=alignedBs offset=slave
#pragma HLS INTERFACE s_axilite port=seqAs offset=control
#pragma HLS INTERFACE s_axilite port=seqBs offset=control
#pragma HLS INTERFACE s_axilite port=alignedAs offset=control
#pragma HLS INTERFACE s_axilite port=alignedBs offset=control

  char seqAs_buf_0[32][16][8]; ③
#pragma HLS array_partition var=seqAs_buf_0 complete dim=1
#pragma HLS array_partition var=seqAs_buf_0 complete dim=3
  // the declarations for the other buffers are omitted
  for (int i=0; i<batch/32+2; i++) {
    if (i \% 2 == 0) {
      load(/* seqAs_buf_0 <= seqAs, seqBs_buf_0 <= seqBs */); ②
      compute(seqAs_buf_1, seqBs_buf_1, alignedAs_buf_1, alignedBs_buf_1)
      store(/* alignedAs_buf_0 <= alignedAs, alignedBs_buf_0 <= alignedBs
          */); ② ③
    }
    else {
      load(/* seqAs_buf_1 <= seqAs, seqBs_buf_1 <= seqBs */); ②
      compute(seqAs_buf_0, seqBs_buf_0, alignedAs_buf_0, alignedBs_buf_0)
      store(/* alignedAs_buf_1 <= alignedAs, alignedBs_buf_1 <= alignedBs
          */); ② ③
}}}
```

We analyze the performance bottleneck in Code 1.1 and propose a proper architecture structure in Table 1.1. The optimized code is demonstrated in Code 1.2, which has about 2× lines of code compared to Code 1.1 we modified from. As can

be seen in Code 1.2, in order to let the HLS tool generate the desired architecture, we need to manually rewrite the C code with very specific structures. Although the N-W accelerator of Code 1.2 is able to achieve around 1,236× speedup on FPGA over a single-thread CPU, the effort of code reconstruction makes the iterative refinement process time-consuming.

In order to reduce the performance gap that caused by HLS-based code reconstruction, a number of automated frameworks have been developed to perform user code analysis and transformation [CZZ12, WLZ13, SYZ16, PZS13, CHZ14, LBC15, LWC16, TLZ15]. Those frameworks contain one or many optimization techniques to make HLS C programming more intuitively for software programmers. In addition, introducing new domain-specific languages (DSLs) is another widely considerable direction [ABC10, ARV03, BVR12, KPZ16, MPA16, SBC15, SPA16, LCH19], because a DSL is designed for only a domain of applications so it implies more semantic information than HLS C and can apply more specific optimization to further improve the QoR. However, current HLS C optimization frameworks and FPGA DSLs are too distinct to benefit each other in terms of architecture-based optimization and design space exploration strategies. This problem becomes one of the main impediment to consolidating the FPGA community on cooperation and developments.

In this dissertation, we present a unified compilation framework for raising an abstraction level of FPGA acceleration. Figure 1.1 illustrates the proposed framework. In Chapter 3, we first present an automated design space exploration framework for Merlin C [CHP16a, CHP16b] on FPGAs. We choose Merlin C because it is syntactic C with a concise set of useful pragmas for automatic code transformations. As we will introduce in Chapter 2, this concise set of pragmas serves a much higher design space coverage and results in higher possibility of achieving the optimal per-

Table 1.1: Analysis of Poor Performance in Code 1.1

Mark	Reason of Poor Performance	Required Architecture Structure
Corresponding HLS Pragma with Required Code Changes		
①	Low bus bit-width utilization	Memory coalescing
Manually use HLS built-in type \texttt{ap_int} with proper bit-width.		
②	Low DRAM bandwidth utilization	Memory burst
Manually allocate local buffer and use memcpy function to copy data.		
③	Sequential communication and computation	Coarse-grained pipeline
Use \texttt{#pragma HLS pipeline} at a non-innermost loop and manually create load/compute/store functions and double buffering.		
④	Lack of parallelism	Coarse-grained parallelism
Manually create a function to wrap the loop and set the correct memory partition factor.		
⑤	Sequential execution	Fine-grained pipeline
Use \texttt{#pragma HLS pipeline} at an innermost loop.		
⑤	Sequential execution	Fine-grained parallelism
Use \texttt{#pragma HLS unroll} with proper array partition factor.		

formance. Accordingly, our design space is composed of combinations of valid options for Merlin C pragmas in a user program. In order to efficiently search for the best configuration in tremendous design points, we first adapt an open-source framework, OpenTuner [AKV14], to perform design space exploration using multi-armed ban-

dit approach with a set of meta-heuristic algorithms. However, since we leverage commercial HLS tools to evaluate the QoR of design points, our evaluation cost is extremely expensive (15 to 60 minutes for one design point). It results in low search efficiency with meta-heuristics. To further improve the search efficiency, we develop a gradient-based approach as well as a design bottleneck optimization algorithm by considering the characteristics of HLS programming model. Our experimental result shows that the proposed DSE framework is able to find the design point that achieves on geometric mean 93.78% to the corresponding manual design.

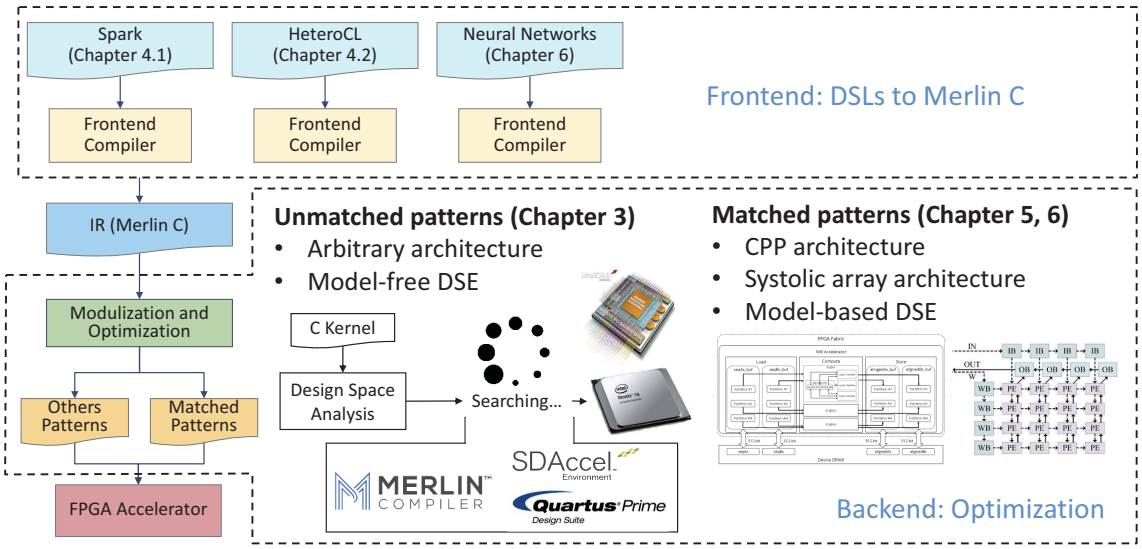


Figure 1.1: The Proposed Framework

Based on the DSE framework, we further design and implement two front-end compilers in Chapter 4 to illustrate how could the proposed framework benefit high-level DSLs. We first target to Apache Spark [ZCD12] in Scala, a widely adopted big-data analytic runtime system in recent years. Specifically, we build a Scala-to-Merlin compiler that guarantees functional correctness. Although the user-defined

functions (UDFs) in Sparks do not have any scheduling information specified by users so our DSE framework has to create a large design space for every possible pragma combinations, we can leverage Spark parallel patterns (e.g. `map`, `reduce`) to prune the design space.

Our second target DSL is HeteroCL [LCH19], a programming infrastructure for FPGAs. Since HeteroCL front-end compiles its DSL to Halide IR [RBA13] which is an in-memory dataflow representation, we implement a HalideIR-to-Merlin compiler to support HeteroCL. Due to the fact that HeteroCL programming model provides prolific scheduling functions to users, our DSE framework automates a part of them and let users focus only on the platform independent loop transformations. Consequently, the development process could be much more efficient.

Instead of supporting arbitrary HLS designs and leveraging commercial HLS tool as an evaluation methodology, sacrificing a degree of generalization could actually make the design space exploration more systematic and stable. In Chapter 5, we propose composable, parallel and pipeline (CPP) micro-architecture template by considering general optimization methodologies as we analyzed in Table 1.1. Although CPP micro-architecture cannot fit arbitrary applications, it is sufficient to support a board class of applications that are suitable to be accelerated on FPGAs. With a fixed micro-architecture template, we are capable of deriving an analytical model for performance and resource utilization. By running only few times of HLS to obtain the design and platform dependent constant values for model initialization, the CPP analytical model can estimate the design quality without HLS tool during the DSE process. The experimental result shows that the DSE process finds the best design configuration under CPP micro-architecture within an hour.

On the other hand, while adopting a generic micro-architecture for a broad class of general designs, we can actually leverage the strengthen of domain specific micro-architectures such as systolic arrays [KUN88] to achieve higher performance for specific applications such as convolutional neural networks (CNNs) without running HLS tool. In Chapter 6, we implement a systolic array architecture template as the accelerator design for CNNs, and we then extend our DSE framework to map a CNN to the architecture template. According to the architecture, we develop a high accurate analytical model for performance and resource estimation. We will demonstrate in Chapter 6 that the DSE can be even more efficient and systematic with an analytical model of a more domain specific architecture.

The remainder of the dissertation is organized as follows. Chapter 2 introduces the background of high-level synthesis (HLS) as well as the Merlin compiler [CHP16a, MER, CHP16b], followed by the motivation of developing an automated DSE framework accordingly and the summary of related work. We then propose the framework in Chapter 3. We first propose a working but inefficient framework based on OpenTuner [AKV14], and gradually improve its design space representation and search algorithms. With the optimized DSE framework, we demonstrate its usability in Chapter 4 by supporting two infrastructures with domain specific languages in Scala and Python, respectively. On the other hand, we demonstrate in Chapter 5 that we can trade generalization with DSE efficiency by proposing a micro-architecture to cover board but not all classes of applications. Finally, we use a CNN case study to illustrate a different but efficient design space exploration approach with limited application domains in Chapter 6. Consequently, the conclusion of this dissertation is given in Chapter 7.

CHAPTER 2

Background

In this chapter, we first introduce the FPGA architecture and the commercial HLS tool for FPGAs, with key focus on required code reconstruction to achieve high performance. Then, we introduce the Merlin compiler [MER, CHP16a, CHP16b] that eases code reconstruction efforts and yet leaves a tremendous design space to users. Finally, we summarize state-of-the-art technologies related to this problem.

2.1 FPGA and High-Level Synthesis C/C++

A field-programmable gate array (FPGA) is a reconfigurable integrated circuit. A typical FPGA architecture [KTR08] is shown in Figure 2.1. Logic blocks, digital processing units (DSPs), and interconnects are reprogrammable to an arbitrary function. Short latency, programmable on-chip block RAMs (BRAMs) also allow developers to implement customized caches and FIFOs with different sizes and bit-widths. This hardware-level customizability allows FPGAs to achieve a significant energy efficiency improvement relative to CPUs and GPUs, as fewer FPGA transistors must be dedicated to control logic.

Researchers have foreseen the opportunity of applying FPGAs into modern datacenters for performance and energy improvement, but the programmability issue

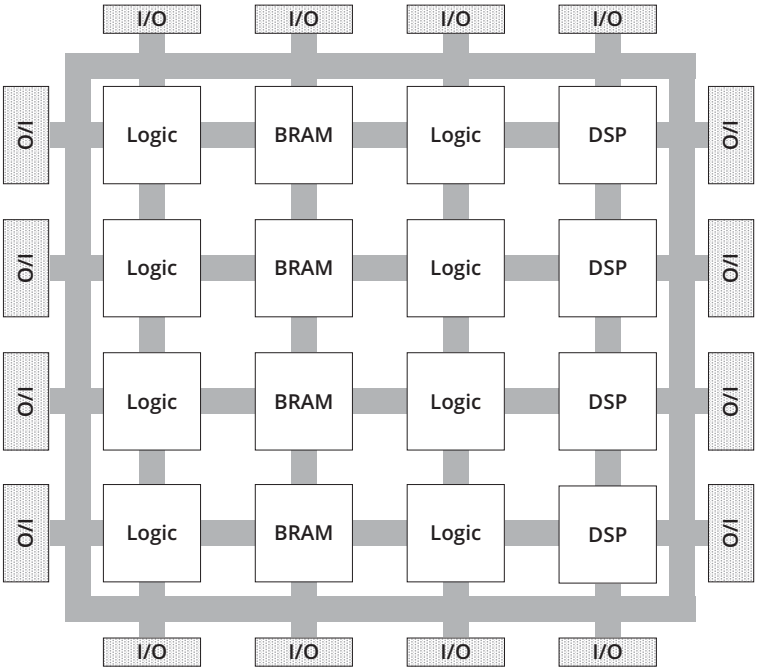


Figure 2.1: A Common FPGA Architecture

emerges as a serious impediment against the adoption of FPGAs to datacenter application developers. In order to design an efficient FPGA kernel, the developer must have a comprehensive understanding of the underlying FPGA architecture. The FPGA kernel is usually implemented in hardware description languages (HDL) such as Verilog and VHDL, which are cycle-sensitive. The learning-curve for FPGAs is usually steep for new programmers.

Fortunately, high-level synthesis languages (HLS) [CLN11] have been developed in recent years to allow programmers to use C-based languages to implement FPGA kernels. Commercial HLS tools such as Xilinx SDAccel [SDX] and Intel FPGA SDK for OpenCL [INT] have been released and widely used to fast prototype user-defined functionalities expressed in high-level languages (e.g., C/C++ and OpenCL)

on FPGAs without involving register-transfer level (RTL) descriptions. The design flows used in these tools are similar, as shown in Figure 2.2. First, a user input program is compiled to the LLVM Intermediate Representation (IR) [LLV07], along with the construction of its control data flow graph (CDFG). Then, the IR-to-HDL (hardware description language) code transformation is performed to map the IR to an RTL design with scheduling optimization. This completes the HLS process that maps the behavioral description of a design to its RTL description. Subsequently, the conventional FPGA design automation flow is launched to generate the design's bitstream file that contains the configuration data for FPGA's logic and RAM blocks.

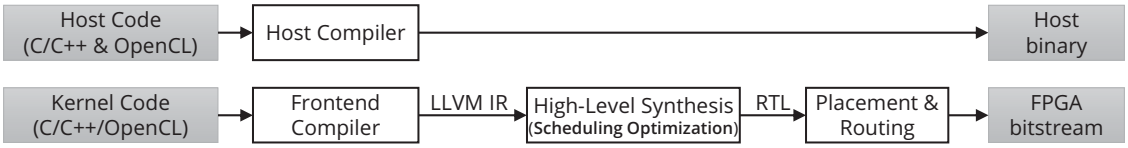


Figure 2.2: Commercial HLS Tool Design Flow

The core HLS code transformation and optimization happens after the LLVM IR is obtained, indicating that the quality of an HLS design highly depends on its IR structure. In other words, two programs with the same functionality but different coding styles (leading to different IR structures) may result in a significant performance difference. In fact, this difference can be up to several orders of magnitude based on our experiences. As a consequence, programmers have to pay attention to every detail that may affect the generated IR structure, which often requires profound understanding of the FPGA architecture and circuit design.

In summary, HLS technologies improve the FPGA programmability by leveling it up from register-transfer level to behavioral level, but do not relieve the burden of manual code transformation that requires hardware design expertise.

2.2 Merlin Compiler

To alleviate the manual effort of heavy code reconstruction when improving a HLS C program, Merlin compiler [MER, CHP16a, CHP16b], a source-to-source transformation tool for FPGA acceleration based on the CMOST [ZHX15] compilation flow, was developed at Falcon Computing Solutions [FCS]. The Merlin compiler provides a set of pragmas with prefix "`#pragma ACCEL`" to represent optimization from the perspective of architecture design. According to user-specified Merlin pragmas, the compiler applies the corresponding architecture structure to the program by invoking abstract syntax tree (AST) analysis, vendor pragma insertion, and source-to-source code transformation. Table 2.1 illustrates the most commonly used Merlin pragmas with corresponding architecture structures. Note that the `flatten` option in the coarse-grained loop pipeline mode refers to the code transformation that tries to apply fine-grained pipelining to a nested loop by fully unrolling all its sub-loops.

Based on the transformation library, Figure 2.3 presents the Merlin compiler execution flow. It leverages the ROSE compiler infrastructure [ROS] and polyhedral framework [ZLC13] for abstract syntax tree (AST) analysis and transformation. The frontend stage analyzes the user program and separates host and computation kernel. The kernel code transformation stage then applies multiple code transformations according to user-specified pragmas. Note that the Merlin compiler will perform all necessary code reconstructions to make a transformation effective. For example, when performing loop unrolling, the Merlin compiler not only unrolls a loop but also conducts memory partitioning for the sake of avoiding bank conflict [CJL11]. Finally, the backend stage takes the transformed kernel and uses the HLS tool to generate the FPGA bit-stream.

Table 2.1: Merlin Pragmas with Architecture Structures

Keyword	Target	Available Options	Architecture Structure
memory_burst	Interface	length=<int>	Large DRAM bandwidth
coalescing	Interface	bitwidth=<2^n>	Memory coalescing
parallel	CG loop	factor=<int>	CG parallelism
parallel	FG loop	factor=<int>	FG parallelism
pipeline	CG loop	off,on,flatten	CG/FG pipeline
pipeline	FG loop	N/A	FG pipeline

CG: Coarse-grained; FG: Fine-grained

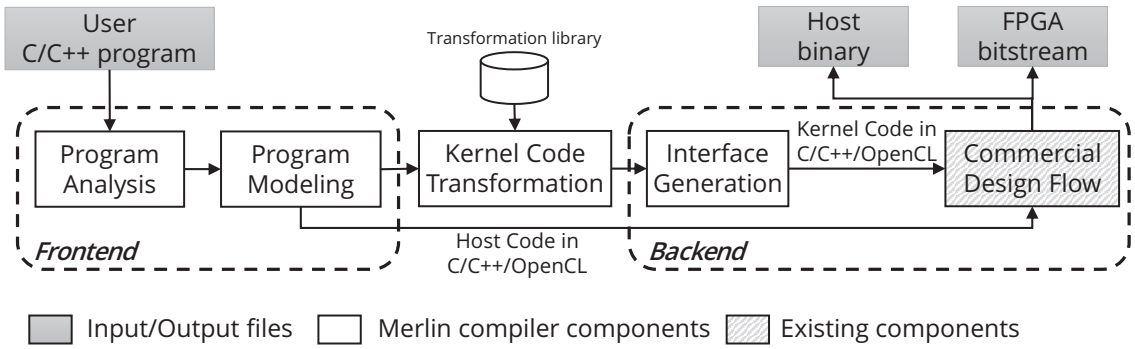


Figure 2.3: The Merlin Compiler Execution Flow

We demonstrate the usability of Merlin compiler pragmas in Code 2.1. As can be seen, by adding a few line of pragmas, the Merlin compiler is able to transform Code 2.1 to Code 1.2 automatically. It means that we can achieve the same performance as the manual optimized HLS C program with less human efforts, and it is much easier for human to explore the best design configuration by simply changing the pragma factors.

Code 2.1: N-W Code Snippet in Merlin C

```
1 void kernel(int batch, char seqAs[], char seqBs[],
2             char alignedAs[], char alignedBs[]) {
3 #pragma ACCEL coalescing var=seqAs bitwidth=512
4 #pragma ACCEL coalescing var=seqBs bitwidth=512
5 #pragma ACCEL coalescing var=alignedAs bitwidth=512
6 #pragma ACCEL coalescing var=alignedAs bitwidth=512
7
8 #pragma ACCEL pipeline
9 #pragma ACCEL parallel factor=32
10  for (int i=0; i<batch; i++) {
11    int M[129][129];
12    ...
13 #pragma ACCEL pipeline
14 #pragma ACCEL parallel factor=8
15    for(i=0; i<129; i++) { M[0][i]=seqAs[...] }
16 #pragma ACCEL pipeline
17 #pragma ACCEL parallel factor=8
18    for(j=0; j<129; j++) { M[j][0]=...}
19 #pragma ACCEL pipeline
20 #pragma ACCEL parallel factor=8
21    for(i=1; i<129; i++) {
22      for(j=1; j<129; j++) { M[i][j]=... }
23    }
24    ...
25 }}
```

Although Merlin pragmas eliminates the manual code reconstruction, a designer still has to manually search for the best option for each pragma, including position, type, and factors. In fact, the N-W design in Code 2.1 has $\sim 10^{10}$ design configurations in terms of Merlin pragma combinations. It motivates this thesis to develop an automation framework to find the best configuration efficiently.

2.3 Related Work

2.3.1 Automated DSE Framework for Hardware Designs

Previous studies have attempted to propose various solutions to address individual design optimization problems. For example, [PSK15] focus on the problem of on-chip memory partitioning; [CHZ14] deal with processing element duplication; [CWY17] handles the improvement of off-chip bandwidth utilization. While these studies model the trade-offs between different design choices and realize the optimal choice via automatic design space exploration, they do not take inter-strategy trade-offs into consideration. In contrast, a number of recent studies have started paying attention to the interaction of different optimization strategies. [WHZ16] and [ZMS16] provide valuable guidance for hardware designers to make good use of various optimization strategies. However, since they do not come up with an automation solution, accelerator developers still have to manually conduct design space exploration.

On the other hand, there are a number of previous work that proposes an automated framework to explore the HLS design space by considering multiple optimization in a design space. We summarize them to two categories according to their search approaches and evaluation methodologies.

Model-based DSE: The first category builds a model using sampled data to realize the performance and resource utilization for each explored design point without actual running the HLS, and use the model to guide the DSE. The authors in [OMC08] use artificial neural networks and linear regression to build performance models for fast design quality estimation. Similarity, authors in [ZKM12] build a regression

model based on Gaussian processes to model area and throughput. Authors in [STW09] and [SW12b] adapt simulated annealing and pattern matching algorithms as search approaches. To the same end, authors in [LC16] leverage transfer-learning to predict design qualities based on the knowledge transferred from the sampled data. Although this approach eliminates the human efforts of porting the framework to another platform since the model training process could be automated, it does not guarantee if the selected learning model fits the target HLS tool or not. For example, the suitable layer numbers and sizes of a neural network for a specific platform may not suitable for another. Even worst, the coverage of training features may not be held for different platforms. Migrating the framework to another HLS tool may violate the assumption and result in a low accuracy of the model.

In addition, another group of work build an analytical model by carefully studying the target HLS tools. [ZPL16, ZFS17] provides a more comprehensive model with the consideration of DSPs and BRAMs, but the author in [ZPL16, ZFS17] does not model the consumption of LUTs which can also be the resource bottleneck in FPGA designs in many cases. In addition, [ZPL16, ZFS17] aim to improve the performance by realizing the optimal HLS directives without code transformation. As a result, its qualify of results highly depends on the structure and coding style of the user input kernel code. [KPZ16, ZPW17] leverage machine learning to model the LUT consumption. However, the model has to be trained for each specific tool implementation, which means the model has to be retrained once the tool is changed or updated. In summary, although those frameworks are able to realize the best design point by searching thousands of design points in a short time, it is hard to port them to another HLS tool in different version or vendor, as the model assumes the underlying architecture and HLS tool implementation are fixed.

Model-free DSE: On the other hand, the other category treats HLS tool as a block box and develops iterative refinement searching algorithms by referring HLS reported result qualities. Most of them first randomly samples a set of design points and find a Pareto set as their starting points, and then apply different algorithms to improve the Pareto set on the fly. The authors in [SW12a], [PG02] and [HKR07] build a predictive model using a sampled data set and use genetic algorithm to refine it. In addition, the design space coverage for those work is relatively low. For example, the design space in [SW12a] only includes loop unroll factor, function scheduling and array resource types, which is insufficient to cover the optimal solution for a board class of applications. [MPZ12, XPZ15, SFP11] adopts response surface models (RSM) and spectral analysis to predict the quality of design points without actual running HLS, but it is hard to guarantee the implementation changes of vendor HLS tools can always be captured by the model.

In addition, the authors in [LC13] proposes a framework that utilizes random forest and randomized transductive experimental design (RTED) to select representative points for training a predictive model that can be used to approximate a Pareto set. However, RTED is failing to random sampling when the design space is too large, so the overall mechanism is not scalable. [FAP18a] resolves this problem by proposing a clustering approach that selects only few clusters of configurations. On the other hand, the author in [FAP18a] also proposes another approach [FAP18b] that groups the design space only based on the variance of design points. They first analyze the design space using principle component analysis (PCA) and claim that the Pareto efficient design points can be clustered with small variances of their configurations. Accordingly, their exploration algorithm randomly selects a neighbor design point of the current Pareto efficient point as the next target. Based on their claim, once the

algorithm finds a global Pareto efficient point, other global Pareto efficient points can be easily explored. This approach, however, might not be efficient with a more complicate and large design space like we have adopted in this dissertation. For example, the design space formed by Merlin pragmas include coarse-grained pipeline and parallelism. Coarse-grained pipeline and parallelism have more uncertainty to the performance and area and it is hard to prove that the global Pareto efficient points are always clustered. Moreover, when the design space is tremendous large (e.g., the scale of 10^{10} to 10^{30}), the cost of initial sampling is not negligible. Even it only samples 1% in the design space, it means 10^8 to 10^{28} design points. Without enough samples, however, the exploration process may not cover the entire design space. For example, in order to explore the design point that has two different parameter values, the proposed algorithm has to reach another design point with only one different parameter value and that point has to be a dominate point to the current point. If the dominate point cannot be reached by changing one parameter, the exploration terminates immediately.

2.3.2 Domain Specific Frameworks for FPGAs

There has been a fair amount of previous work that generates FPGA code from high-level programming languages. Vivado HLS [VIV, CLN11] is a commercial tool that performs high-level synthesis to generate FPGA kernel code from C-based languages. CHiMPS [PBD08] takes ANSI C code as an input and generates VHDL blocks for FPGAs. However, using C-based programming languages to describe parallelism is not trivial because their execution model and design logic are fundamentally sequential. In contrast, the domain-specific language [BVR12, ARV03, ABC10] leverages

specialized programming models to guide the compiler for more optimizations.

In addition, some FPGA-based frameworks are developed for a specific application domain such as machine learning [MPA16] and SQL queries [CDL13]. Since these frameworks map user programs to pre-defined hardware templates with specific functionality, they only support limited kernels. On the other hand, [PKB16] develops a parallel pattern language for FPGAs and compiles the source code to DHDL [KPZ16], an intermediate representation language for FPGAs. The DHDL kernel is able to be transformed to an FPGA design with hardware template and design space exploration. However, [PKB16] is designed for single node applications and does not consider programmability as well as system integration.

Furthermore, some other work performs FPGA code generation in the context of datacenter runtime systems. [SMC14] integrates AMD APARAPI [APA] into Apache Hadoop [HAD] and targets FPGAs. However, this work only supports primitive types, and requires manual design optimization. Their followed work, SparkCL [SCN15], extended the framework to Spark but a detail evaluation is missing. Melia [WZH16] is a MapReduce framework that automatically generates FPGA kernels in OpenCL from user-written functions, and optimizes the generated kernels by leveraging an analytical performance model [WHZ16]. The generated FPGA kernel is invoked by the Melia runtime system. However, the source language in Melia is still a syntactically C language, so programmability is limited. In addition, Melia is not compatible with any widely used big data analytics frameworks, so users must rewrite their applications using the Melia programming model in order to adopt this framework.

CHAPTER 3

An Automated Design Optimization Framework

3.1 Overview

As we have elaborated in Chapter 1 and Chapter 2, the HLS programming models for modern commercial tools require manual code reconstructions to help the tool realize certain architecture patterns such as dataflow, processing element replication, memory burst, and so on. There have some existing work [ZVL14, ZFS17] that have proposed automation frameworks to free humans from the tedious code reconstruction process in development cycles by more or less fixing the architecture patterns and building analytical models. [ZVL14] uses a simple analytical mode to estimate performance and area of loops in the kernel, and decides whether the dataflow architecture should be enabled or not accordingly. [ZFS17] also proposes an analytical model for performance estimation and applies dataflow architecture to user designs when applicable. Unfortunately, their models are based on the assumption that an individual design parameter will affect the performance/area in a smooth and/or monotonic way, which is not true in general with latest HLS tools as well as the larger design space used in this thesis. For instance, Figure 3.1 depicts the execution cycle of the N-W algorithm with different parallel factors for its 5 loops synthesized by Xilinx SDAccel [SDX]. Although the performance trend of `CG-loop-1`, `FG-loop-2`,

and `FG-loop-3`[1] are ideal (so we use the same dash line in the figure), the rest 2 loops (`CG-loop-2` and `FG-loop-1`) are not. Note that these behaviors may differ from version to version; therefore it is impractical to maintain an analytical model for DSE.

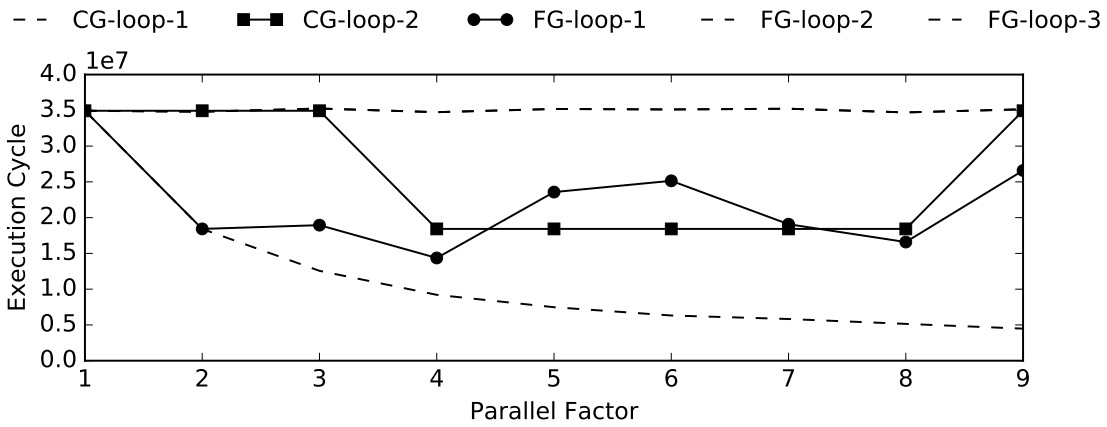


Figure 3.1: HLS Cycles of N-W with Different Factors on Loops

On the other hand, the main challenge of using vendor HLS tools for DSE to capture all vendor tool behaviors is the evaluation cost, since vendor HLS tools usually take 15-60 minutes to generate RTL and estimate the performance, and it usually takes a longer time if the design has a high performance. As a result, general iterative learning approaches are unstable to find the high quality of result (QoR) design configuration in a reasonable amount of time. To improve the DSE efficiency, in this chapter, we propose a comprehensive design space representation that only includes meaningful design point while preserving a regular design space shape. By explicitly representing the design space with application-specific knowledge, we are able to avoid the meaningless design points and improve the DSE efficiency. In

[1] `CG` and `FG` mean coarse-grained and fine-grained, respectively.

addition, based on the design space we study the effectiveness of multi-armed bandit approach [FDS10] with meta-heuristic optimization algorithms and further propose new algorithms to facilitate the search process. In summary, this chapter makes the following contributions:

- We propose an efficient but comprehensive design space representation to support dependent design space parameters.
- We use multi-armed bandit approach with evolutionary algorithms [BNK98] and particle swarm optimization [GSZ18] for DSE and analyze their challenges.
- Based on the insights from multi-armed bandit approach, we develop a gradient search algorithm that leverages finite difference method to approximate gradient values for systematically approaching to high-QoR design points.
- By reasoning the gradient search process, we further improve the gradient approach with performance bottleneck analysis to improve the search efficiency.

Our experimental result shows that the proposed DSE framework is able to find the design point that achieves on geometric mean 92.56% to the corresponding manual design.

The rest of this chapter is organized as follows. Section3.2 presents the initial version of our framework that contains working flow with the use of multi-armed bandit approach for several meta-heuristic optimization methods. Then, Section 3.3 shows an improved framework with HLS DSE specific optimization to its execution flow. In order to reason the DSE process for resolving the low QoR issue for certain designs, Section 3.4 proposes an algorithm to statically prune the design space, as well

as a gradient-based search algorithm to systematically identify better design points. Finally, Section 3.5 further presents a comprehensive design space representation and an improved gradient-based search algorithm. Finally, the summary is given in Section 3.6.

3.2 Version 1: An Initial Framework

3.2.1 Framework Overview

Traditionally, numerical approaches such as linear programming are widely used for performing DSE. Unfortunately, it is inapplicable to our case because such approaches require at least an analytical form to evaluate the design quality. Since our goal is to cover the difference of commercial HLS tools, we treat the evaluation function as a black-box and only accepts its outputs (QoR report) by feeding design points. As a result, our initial idea is to use a set of meta-heuristic algorithms, such as evolution genetic algorithms [BNK98], and particle swarm optimization [GSZ18], to perform DSE.

However, a well-tuned meta-heuristic algorithm is usually too specific to cover a board class of applications. In order to assemble multiple meta-heuristic algorithms to improve the generalization, hyper-heuristic, which searches for an optimal solution by selecting one of the meta-heuristic algorithms in a pool iteratively, is proposed. Hyper-heuristic usually uses an exploitation versus exploration (EvE) approach as follows to rank a meta-heuristic algorithm a for specific applications:

$$Score(a) = Exploitation(a) + c \times Exploration(a) \tag{3.1}$$

where the former term guarantees the meta-heuristic with better performance will be selected more frequently; while the latter term leaves opportunities to other meta-heuristics. c is a constant for exploitation-exploration dilemma.

We use OpenTuner [AKV14], an open-source auto-tuning framework for software programs as the hyper-heuristic engine to explore the design space. OpenTuner leverages multi-armed bandit (MAB) approach [FDS10] with sliding window area under curve (AUC) credit assignment [PAA12]. Every meta-heuristic in the MAB approach is an arm with a dynamic EvE credit as its score. The credit of a meta-heuristic m is defined as:

$$C_m = AUC_m + c \times \sqrt{\frac{2log|H|}{H_m}} \tag{3.2}$$

where AUC_m is the quantified performance (exploitation) of meta-heuristics while $\sqrt{\frac{2log|H|}{H_m}}$ is the quantified uncertainty (exploration). In addition, H_m is the number of times that m has been selected during a history sliding window with length $|H|$. At each iteration, the MAB selects the meta-heuristic with the highest credit and updates creates based on the result. Consequently, the meta-heuristic that can efficiently finds high-quality design points will be rewarded and activated more frequently by the MAB, and vice versa.

Figure 3.2 shows an initial framework based on OpenTuner. The framework accepts a user-written C kernel as input and first performs static code analysis to identify the design space (Section 3.2.2), which is composed of Merlin pragmas and their valid options. The design space is then explored using MAB approach with several meta-heuristic algorithms (Section 3.2.3). Every design point generated by search algorithms will apply corresponding Merlin source-to-source code transforma-

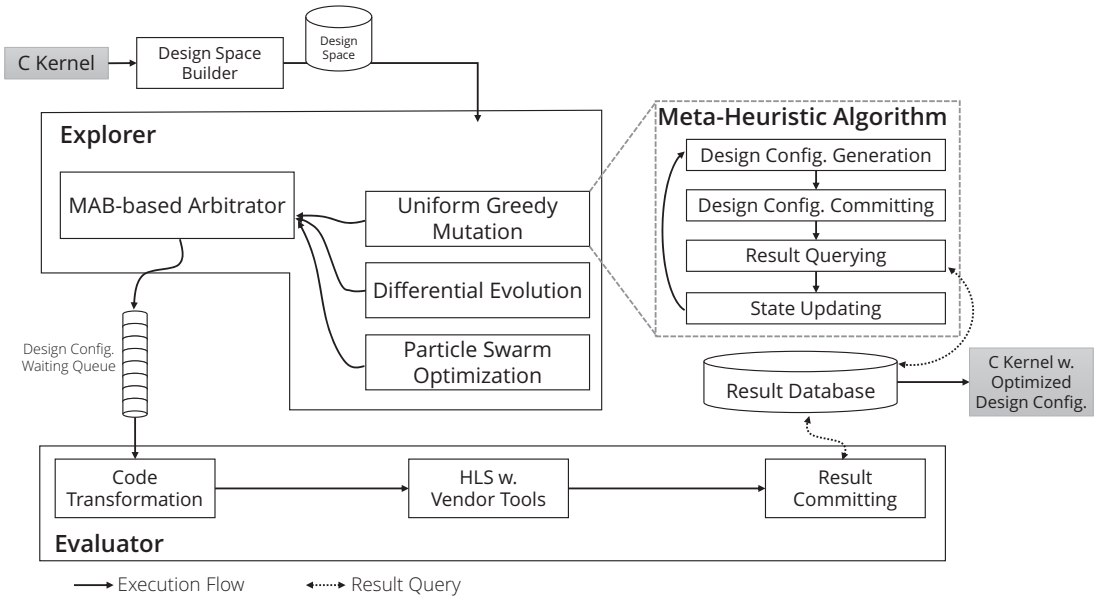


Figure 3.2: The Framework based on OpenTuner [AKV14]

tions and be evaluated using commercial HLS tools. The evaluation result is stored in a shared result database that can be accessed by all algorithms so that population-based meta-heuristic algorithms could go through a shortcut by taking others better results anytime. Finally, when the exploration is terminated due to the time limit, the framework outputs the so far best design point to continue the rest accelerator generation process. In the rest of this section, we detail introduce the design space as well as the meta-heuristic we adopted in the framework.

3.2.2 Design Space Identification

We formulate the problem of identifying a design space in a C program as follows: Given a C program $\mathcal{P}$ as the FPGA accelerator kernel, find a set $\mathbb{S}_{\mathcal{P}}$ that contains

possible combinations of Merlin pragmas for $\mathcal{P}$ as design configurations.

Table 3.1: Merlin Pragma Formed Design Space

Factor	Design Space (Values)
Memory coalescing	$\{b \mid b = bw(B) \in \mathbb{B}, 8 < b = 2^n < 512\}$
Memory burst	$\{t \mid t = T(L) \in \mathbb{L}, 1 < t < TC(L)\}$
CG-loop parallel	$\{u \mid u = UF(L) \in \mathbb{L}, 1 < u < TC(L)\}$
FG-loop parallel	$\left\{ u \mid u = UF(L) \in \mathbb{L}, \begin{cases} 1 < u < TC(L), & \text{TC(L)} > 16 \\ u = TC(L), & \text{otherwise} \end{cases} \right\}$
CG-loop pipeline	$\{p \mid p = P(L) \in \mathbb{L}, p \in \{off, on, flatten\}\}$
FG-loop pipeline	$\{p \mid p = P(L) \in \mathbb{L}, p = fg\}$

CG: Coarse-grained; FG: Fine-grained; TC: Loop trip-count

We list available design spaces based on Merlin pragmas in Table 3.1. We identify the design space for each kernel by analyzing the kernel AST using the ROSE compiler infrastructure [ROS] and polyhedral framework [ZLC13] to realize loop trip-counts, available bit-widths, and so on. In addition, since vendor HLS tools usually schedule fine-grained loops well, we only explore the parallel factor of fine-grained loops when its trip-count is larger than 16; otherwise, we simply apply fully unroll and pipeline to small fine-grained loops to reduce the design space. As can be seen, it is impractical to explore this tremendous design space exhaustively. For example, the design space of the N-W example contains more than a 10^{13} design points. This illustrates the importance of search algorithm efficiency for reaching a near-optimal solution in a few iteration.

3.2.3 Meta-Heuristic Optimization Methods

Given a C program $\mathcal{P}$ as the FPGA accelerator kernel along with its design space set $\mathbb{R}^K_{\mathcal{P}}$ which is identified in the previous section, and a commercial HLS tool $\mathbf{H}$ that estimates the execution cycle $Cycle(\mathbf{H}, \mathcal{P})$ and resource utilization $Util(\mathbf{H}, \mathcal{P})$ of the given $\mathcal{P}$ as a black-box evaluation function, find a configuration $\theta \in \mathbb{R}^K_{\mathcal{P}}$ in a given time limit so that the generated design $\mathcal{P}(\theta)$ with θ can fit in the FPGA, and execution cycle is minimized. Formally, we define the problem as:

$$\min_{\theta} Cycle(\mathbf{H}, \mathcal{P}(\theta)) \tag{3.3}$$

subject to

$$\theta \in \mathbb{R}^K_{\mathcal{P}} \tag{3.4}$$

$$u < T_u \ \forall u \in Util(\mathbf{H}, \mathcal{P}(\theta)) \tag{3.5}$$

where u is the utilization of one of the FPGA on-chip resources and T_u is a user-available resource threshold on FPGAs. We set all T_u to be 0.8 in our experiments to reserve the resource used by FPGA firmware. With multi-armed bandit (MAB) approach as a hyper-heuristic search algorithm, we choose and implement the following discrete value friendly meta-heuristic optimization algorithms to achieve high generalization.

Uniform Greedy Mutation: Mutation is one of widely used genetic algorithms that analogizes biological mutation in order to iteratively optimize the solution, since it is easy to use and apply to almost all optimization problems. Specifically in our

problem, the algorithm generates a set of next generation design points (∼10) by mutating the design points in the current generation based on the given mutation probability (0.1 in our framework). Since our design space options are discrete values and the cost function (commercial HLS tool) does not have specific trend or distribution, we simply leverage uniform distribution when mutating design points. In addition, to facilitate the evolution efficiency, we greedily select the best design point as the parent for the next generation instead of maintaining an active set. Although it is obvious that uniform greedy mutation may not perform well due to less diversity, we complement this problem by leveraging differential evolution algorithm.

Differential Evolution: Differential evolution [SP97] is an optimization algorithm and has been widely applied to many problems in different fields, because it makes very few assumption about the problem. The core idea behind differential evolution algorithm is that it maintains a set of active design points (∼30) and crossovers them to create new candidates, as shown in Algorithm 1. As can be seen, unlike traditional gradient descent, differential evolution only requires the quality of results of generated design points instead of differentiating the cost function, so it is suitable to be used for the problems that does not have differentiable formulation. Note that although differential evolution does not guarantee to find the global optimal, the maintenance of the active set preserves its ability of jumping out of local optimal. In addition, thanks to the shared result database, the differential evolution in OpenTuner is able to include the global best design point to the active set at every iteration (line 6-8), so it can complement the mutation algorithm to improve the search efficiency.

Algorithm 1 Differential Evolution Implementation in OpenTuner [AKV14]

Require: A design space $\mathbb{S}$; time limit T; crossover rate C

Ensure: A design configuration θ with the best QoR.

1: $CurrPoints \leftarrow RandomPoints(\mathbb{S}, N)$

2: $t \leftarrow 0$

3: **while** $t < T$ **do**

4: **for all** $curr \in CurrPoints$ **do**

5: $child \leftarrow copy(curr)$

6: **if** $EvalQoR(GetBest(CurrPoints)) < EvalQoR(GlobalBest())$ **then**

7: $CurrPoints.add(GlobalBest())$

8: **end if**

9: $p_1, p_2, p_3 \leftarrow RandomPoints(CurrPoints, 3)$

10: **for all** $param \in curr.parameters$ **do**

11: $c \leftarrow UniformRandom(0, 1)$

12: **if** $c < C$ **then**

13: $param \leftarrow p_1.param + \frac{UniformRandom(0,2)+0.5}{2} \times (p_2.param - p_3.param)$

14: **end if**

15: **end for**

16: **if** $EvalQoR(child) > EvalQoR(curr)$ **then**

17: $curr \leftarrow child$

18: **end if**

19: **end for**

20: $t \leftarrow t + ElapsedTime()$

21: **end while**

22: **return** $GetBest(CurrPoints)$

Particle Swarm Optimization Algorithm: Particle swarm optimization [KE95], or PSO, is a population-based stochastic optimization algorithm originally developed for social behavior simulation. Although PSO also maintains a set of active design

points like differential evolution, it does not generate the next design point candidate by combining the parent points. Instead, PSO moves a set of active points according to their positions and velocities to generate new candidates, as shown in Algorithm 2. In line 7 of Algorithm 2, the moving velocity and direction are stochastically determined by the local best point found by the PSO (*particle.best*) and the global best point found by other algorithms ($GlobalBest()$). Note that $c1$ and $c2$ are the weights of local and global best points, respectively. By including PSO in our meta-heuristic algorithm set for MAB, we are able to have a high probability to explore better design points around the current set.

Algorithm 2 Particle Swarm Optimization in OpenTuner [AKV14]

Require: A design space $\mathbb{S}$; time limit T; crossover rate C

Ensure: A design configuration θ with the best QoR.

1: $Particles \leftarrow NewParticles(RandomPoints(\mathbb{S}, N))$
2: $t \leftarrow 0$
3: **while** $t < T$ **do**
4: **for all** $particle \in Particles$ **do**
5: **for all** $p \in particle.parameters$ **do**
6: $r1, r2 \leftarrow UniformRandom(0, 1)$
7: $particle.velocity_p \leftarrow c \times particle.velocity_p + c_1 \times r1 \times (particle.best_p - particle.curr_p) + c_2 \times r \times (GlobalBest()_p - particle.curr_p)$
8: $particle.curr_p \leftarrow move(particle.curr_p, particle.velocity_p)$
9: **end for**
10: **if** $EvalQoR(particle.curr) > EvalQoR(particle.best)$ **then**
11: $particle.best \leftarrow particle.curr$
12: **end if**

13: **end for**

14: $t \leftarrow t + ElapsedTime()$

15: **end while**

16: **return** $GetBest(CurrPoints)$

3.2.4 Experimental Results

3.2.4.1 Experimental Setup

Our evaluation is performed on Amazon EC2 [AWS]. We use a memory-optimized CPU instance, `r5.4xlarge`, with 16 cores and 122 GiB memory to perform the DSE and generate FPGA accelerator bit-streams. Note that the HLS tool we used for evaluating design points is Xilinx SDAccel 2018.2 [SDX] which requires 2 GB host memory but recommends 64 GB, so we allocate at most one case with 8 threads running in parallel to reduce an out-of-memory issue when performing the DSE. The maximum DSE time is set to 4 hours. The generated FPGA accelerators are evaluated on AWS F1 instance [AWS] (`f1.2xlarge`) that includes an 8-core CPU with 122 GiB of main memory and one Xilinx Virtex UltraScale+^{TM} VU9P FPGA with three separated dies and 300 MHz working frequency. In addition, our benchmark is selected from the MachSuite [RAS14] benchmark suite and the FPGA-friendly Rodinia benchmark [CFL18]. We describe the benchmark as well as the input data information in Table 3.2.

Table 3.2: Benchmark Description and Lines-of-Code (LOC)

Kernel	Description and Input Information
AES	Advanced encryption standard. (LOC: 198) Input: 256-bit key; 64MB data.
GEMM	General matrix multiplication. (LOC: 34) Input: two 10241024 double-precision matrices
KMP	Knuth-Morris-Pratt string matching. (LOC: 84) Input: 128MB string; 16B substring.
NW	Needleman-Wunsch sequence alignment. (LOC: 213) Input: 64K pairs of 128-nucleotide sequence.
SPMV	Sparse matrix-vector multiplication. (LOC: 59) Input: 4096512 ELLPACK data and index.
STENCIL-2D	2-D stencil computation. (LOC: 54) Input: a 4096^2 image.
STENCIL-3D	3-D stencil computation. (LOC: 77) Input: a 4096^3 image.
BACKPROP	The weight updating step in back propagation. (LOC: 35) Input: 65536 neuron outputs and 17 weight values.
KMEANS	K-Means clustering algorithm. (LOC: 66) Input: 819,200 data points with 32 features for 5 clusters.
KNN	Distance calculation of K-nearest neighbors. (LOC: 38) Input: 1,048,576 2-D data points.
PATHFINDER	Shortest path finder on a 1024×1024 grid. (LOC: 83)
CONV	One convolutional layer in AlexNet [KSH12]. (LOC: 54)

For each benchmark case, we manually implement an optimal version using Merlin compiler pragmas to evaluate the result quality of the proposed DSE framework, and the results are shown in Figure 3.3 with geometric mean 14.8× speedup over the CPU. Since whether the Merlin pragma formed design space is capable of covering the optimal solution is out of scope of this thesis, the discussion in the rest of this chapter will focus on the optimality achievement over the manual design instead of the speedup over CPU.

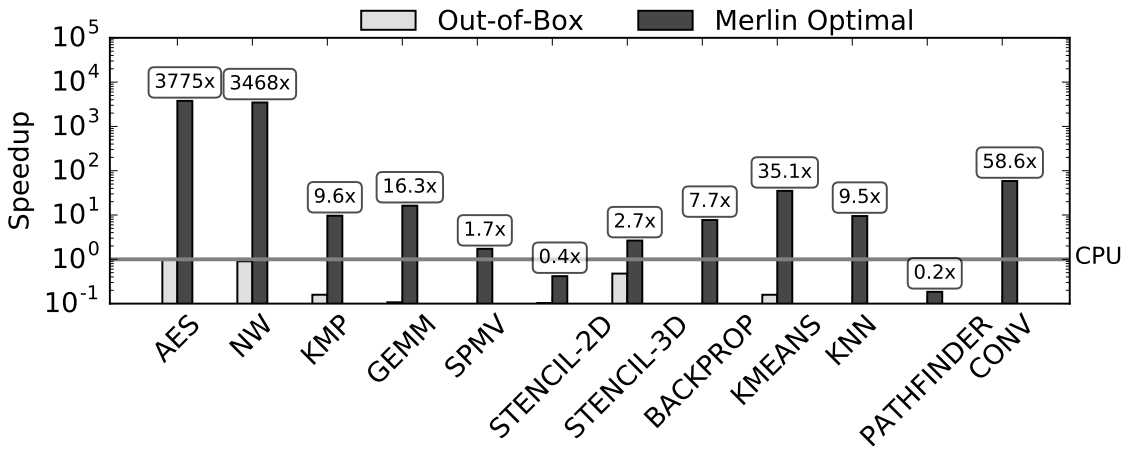


Figure 3.3: Benchmark Speedup with Manual Merlin Pragma Optimization. Note that the Out-of-Box performance of some cases may be too poor to be visualized.

3.2.4.2 Result and Analysis

We first analyze the overall result quality achieved by the DSE framework in Table 3.3 and Table 3.4. In Table 3.3, the second column presents the design space size of each case with $\sim 10^{10}$ as their geometric mean. With such tremendous design space, our customized OpenTuner-based framework realizes the design point that achieves more

than 80% of the optimal latency for about an half cases in 4 hours. This is definitely inefficient, however, for the DSE problem to HLS on FPGAs.

Table 3.3: Overall Comparison to CPU and Manual Designs and the Dominated Heuristic (GM: Greedy Mutation, EV: Differential Evolution, PSO: Particle Swarm Optimization.)

Benchmark	Design Space	Best Point Found by	DSE Latency	Ratio to Manual	Speedup over CPU
AES	2.81E+11	PSO	21171289	39.64	1496.37
NW	2.70E+10	GM	620102	95.8	3222.4
KMP	5.76E+03	GM	44165122	100	9.65
GEMM	2.52E+09	GM	50634229	54.62	8.88
SPMV	1.73E+04	EV	3143273	13.23	0.27
STENCIL-2D	4.37E+11	GM	3370012	80.64	0.34
STENCIL-3D	1.78E+08	GM	337927263	5.29	0.14
BACKPROP	5.18E+04	EV	278747	96.1	7.71
KMEANS	1.67E+06	EV	1722572	99.18	34.82
KNN	1.03E+05	PSO	1019906	38.73	3.68
PATHFINDER	1.66E+04	EV	41915562	51.40	0.01
CONV	7.35E+28	GM	1.29E+09	16.82	9.86
Geometric Mean	6.80E+10			44.60	6.59

We dive into the DSE process of some designs with poor performance compared to the manual and summarize two highly possible reasons. For `AES` and `CONV`, since both of them have relatively large design space and our evaluation methodology is time-consuming, the meta-heuristic algorithms do not perform sufficient iterations to

identify the direction toward to a better design point. For `STENCIL-3D` and `KNN`, although their design spaces are relatively small, their design parameters have impacts on each other so it is also hard to capture the direction of improving the result quality. For example, `STENCIL-3D` implementation includes an if-statement to deal with the stencil boundary. The if-block and else-block of the statement contains similar loop structures that access the same 3-D array. As a result, an improper combination of parallel factors of those loops may result in bad array partition factor and degrade the performance. Note that other designs that achieve better results may also have the same issue but just do not expose in our DSE process. This illustrates the unstability of adopting non-deterministic approach for DSE which only allows a small number of search iterations.

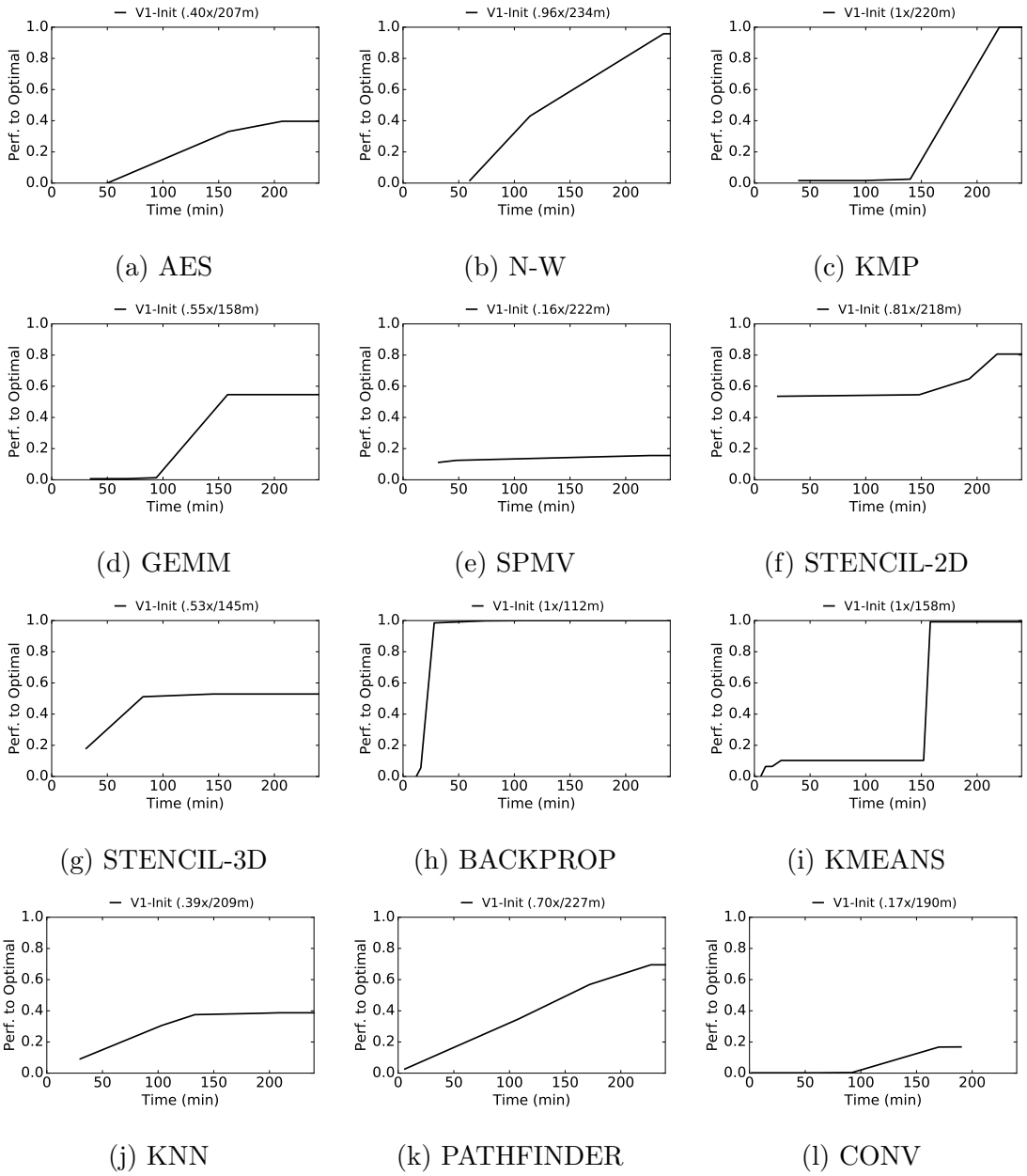
In addition, we can see from Table 3.4 that most of the best design points found by our framework use low on-board resources. It means that the search process has not yet found a right direction to toward to in the time limit. We note that most previous work as we have illustrated in Chapter 2 would also encounter the same issue because their experiments take hundreds of iterations to explore the design space with the scale of 10^4, which is several orders of magnitude smaller than ours. As a result, most approaches proposed by previous work is inapplicable to our problem, since 4-hour DSE time can only explore $\sim$100 points.

We then evaluate the DSE process in Figure 3.4. The x-axis depicts the DSE time while the y-axis is the speedup over the manual design using Merlin compiler (i.e., 1.0 means DSE is able to find the optimal design point in the time limit). As can be seen, the trend of most cases does not start from the the first minute. This is because the framework starts from a random design point which may not be synthesizable due to design complexity or resource issue. As a result, the framework needs to spend

Table 3.4: FPGA Resource Utilization

Benchmark	BRAM (%)	LUT (%)	FF (%)	DSP (%)
AES	33	13	4	0
NW	39	18	14	0
KMP	24	13	1	0
GEMM	51	35	6	31
SPMV	19	10	1	6
STENCIL-2D	2	1	1	1
STENCIL-3D	3	3	2	1
BACKPROP	71	37	8	10
KMEANS	50	26	1	18
KNN	8	3	2	1
PATHFINDER	7	7	1	2
CONV	56	9	6	6

some iterations on exploring the first available design point. In addition, we can find an obvious steep curve in most cases. Since meta-heuristic algorithms generate new design points by combining current points, the steep curve implies that the direction to a better is obscure and hard to be reasoned.



(a) AES (b) N-W (c) KMP

(d) GEMM (e) SPMV (f) STENCIL-2D

(g) STENCIL-3D (h) BACKPROP (i) KMEANS

(j) KNN (k) PATHFINDER (l) CONV

Figure 3.4: Design Space Exploration with OpenTuner [AKV14]. The legend notes the best speedup over the manual design and the time to achieve it.

3.2.5 Insights and Summary

In this section, we build a working DSE framework based on OpenTuner, an extensible auto-tuning framework, for HLS on FPGAs. Although the adopted meta-heuristic algorithms in the framework are guaranteed to reach the entire design space, it is inefficient to find the best design point in 4 hour time limit. We summarize the main impediments as follows:

Impediment 1: Expensive evaluation approach: In order to cover all possible user-written kernels in our framework, we use the Xilinx SDAccel [SDX] to perform HLS for resource and cycle estimation instead of building an analytical model. However, HLS takes several minutes to evaluate one design point so only tens of design points can be evaluated in one hour.

Impediment 2: Complex factor dependencies: Many design space factors a have high dependency on each other. For example, enabling fine-grained pipelining to a nested loop (`flatten` in Table 3.1) causes all sub-loops to be fully unrolled and results in the invalidation of corresponding design space factors. This phenomenon might mislead the iterative optimization algorithm and result in more iterations on realizing the best design point.

The above two impediments motivate us to improve the framework in the next section.

3.3 Version 2: Framework Optimization

In this section, we introduce methodologies for addressing the two impediments to improve the DSE efficiency. Section 3.3.1 introduces the overall improved framework,

followed by three sections to detail describe the framework implementation. Section 3.3.5 presents experimental results and the summary is placed in Section 3.3.6.

3.3.1 Framework Overview

A straightforward but effective approach to address *Impediment 1* is searching the design space in parallel. Since our meta-heuristic algorithms may not have enough active design points to fully utilize CPU threads and may fall into sequential search, one of the main improvements in the version 1 framework is to guarantee our framework could make use of all CPU threads all the time.

We present the improved framework in Figure 3.5. The red part of the framework highlights the changes from version 1. The idea of our parallel DSE process is partially inspired by DATuner [XLZ17], a parallel auto-tuner for Verilog-to-Routing (VTR) FPGA compilation. DATuner finds the best parameter values of the VTR tool to achieve better resource utilization and frequency in a given, fixed time period by dynamically partitioning the design space and allocating more CPU cores to the partition with better QoR. In contrast, our flow in Figure 3.5 parallelizes the DSE process based on static partition rules (Section 3.3.2) to avoid set-up time. Also, unlike DATuner that uses random seeds and a time limit to start and terminate the DSE of a partition, our flow generates effective seeds for each partition to reduce the probability of being trapped in the infeasible design space region (Section 3.3.3), and sets up a stopping criteria to avoid long tails (Section 3.3.4).

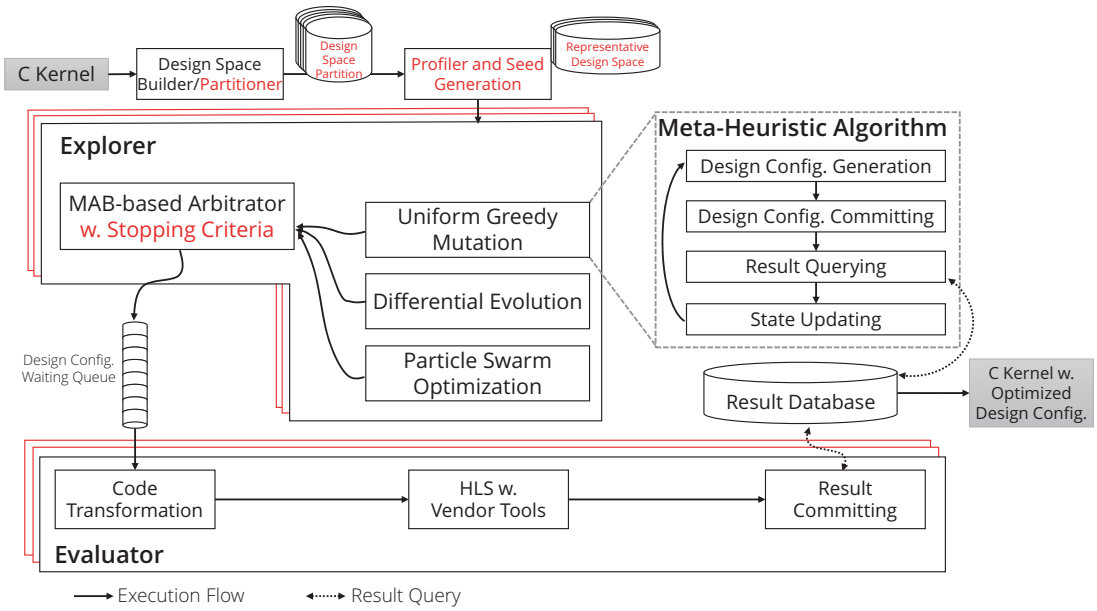


Figure 3.5: The Improved Parallel Exploration Framework

3.3.2 Design Space Partition

Since the meta-heuristic algorithms we adopted are iterative algorithms that have a strong dependency between iterations, we cannot simply increase the DSE efficiency using more CPU cores to address *Impediment 1*. As a result, we statically separate the design space into independent partitions and assign different cores to different partitions to perform the DSE in parallel. As shown in Figure 3.5, our flow has a mechanism that adopts the first-come-first-serve approach to schedule partitions to threads, so it perfectly solves *Impediment 1* as long as the partition number is larger than or equal to the number of CPU cores.

Although the authors in [XLZ17] claim that the dynamic partition is more case-specific and results in a better convergence rate than the "one-for-all" static partition,

it needs several iterations for sampling at the beginning of the DSE process for every partition. Consequently, to take advantage of both, we adopt the "some-for-all" static partition approach. We statically create some sets of rules and choose the set that is most suitable to the design for partitioning only at the beginning of a DSE process.

Our partition rule is created based on the program loop hierarchy in order to reflect the design factor dependency (*Impediment 2*). According to our observation, the same loop level could have similar impact on performance even in different applications, so we attempt to partition the design space based on the loop level. However, it is impractical to build application-specific loop-level based rules without any training data. As a result, we use a heuristic approach by grouping the applications with similar loop hierarchy geometrically and generate training data to establish the rules.

To rank the importance of loop levels, we build a binary decision tree that clusters the design points which potentially have similar resource utilization or latency so that the exploration process could be more efficient. Decision tree is a popular method for classification and regression. Each tree node represents a rule that is composed of a parameter and a condition (e.g., `parallel factor < 16`). A path from the tree root to a leaf with all rules on the path are conjugated to form a partition. These nodes are determined by greedily selecting the best rule to maximize the information gain. Formally, we choose nodes from the set $argmax_n IG(n, D)$ where $IG(n, D)$ is the information gain if we apply node n to the dataset D, as it has been defined as follows [RM05]:

$$IG(n, D) = Imp(D) - \frac{N_l}{N} \times Imp(D_l) - \frac{N_r}{N} \times Imp(D_r) \tag{3.6}$$

where N_l, N_r, N are the size of the left partition subset D_l, right partition subset D_r and overall dataset D respectively. $Imp(D)$ is an impurity measurement of dataset D. Impurity function is usually selected based on the type of decision tree task (classification or regression). Since the value of each partition in our case is a regressed number (latency), we choose variance as our impurity function.

For example, in our design space formed by Merlin pragmas, as shown in Table 3.1, the most two common partition factors identified by decision trees in almost all cases are 1) the pipeline pragmas on coarse-grained loops with *flatten* options, and 2) the parallel pragmas on fine-grained loops. Those two pragmas are exactly the pragmas with the highest impact on performance change we have observed in the previous section. By separating those pragmas to different partitions using a decision tree, we can efficiently alleviate *Impediment 2* because the meta-heuristic algorithm is able to approach to better design points without being disturbed by outliers. We note that since all partitions are disjoint and the union of all partitions is the original design space, our design space partition approach preserves the optimality while improving the DSE efficiency.

3.3.3 Seed Generation

Although we have partitioned the design space systematically in the previous section, a partition may still contain millions of design points. However, it is too aggressive to prune the design space using heuristics such as limiting parallel factor or local buffer size, because the boundary of those factors varies from arbitrary user-written

kernels and results in a different infeasible region in the design space. For instance, performing coarse-grained parallelism with factor 256 to the outermost loop might be infeasible for most designs due to high routing complexity, but it could be an optimal choice for certain designs that have a very simple computational pattern. As a result, instead of heuristic pruning, we preserve an entire design space but increase the probability of finding the best design point in fewer iterations by providing seeds, the starting point for learning algorithms.

We generate two seeds for each partition with different strategies. The first seed is *performance-driven*. For this seed, we enable fine-grained pipelining for all loops, set the parallel factor of every loop to 16, and set the buffer bit-width to 512. Although this configuration might fail to be synthesized for some designs, we can significantly reduce the iteration number of the DSE process for others. On the other hand, the second seed is *area-driven*. For this seed, we disable all optimizations so all loops are performed sequentially and all off-chip buffers are set to the minimum bit-width. As a result, this seed has the most conservative configuration in terms of resource utilization and design complexity, so it is less likely to be infeasible from the perspective of the high-level synthesis tool. With both *performance-driven seed* and *area-driven seed* as the starting points in parallel, the learning algorithm may achieve high performance in the first iteration and is guaranteed to start searching in the feasible region and avoid being trapped in the infeasible region all the time.

3.3.4 Early Stopping Criteria

Since the vanilla OpenTuner does not have a systematic stopping criteria but only adopts the limitation of either execution time or searched point count, the long tail

is almost inevitable. In fact, the long tail becomes a serious problem for exploring FPGA accelerator designs because we need minutes to an hour to evaluate a single design point using HLS.

To solve the long tail problem without the knowledge of optimal performance, we add one more criteria in addition to the time limit to stop the DSE process earlier based on the following concept. According to the dataset of explored results D_i after i iterations, and its subset of the uphill performance results between any two consecutive iterations D_i^u, let $P_{D_i}(D_i^u \mid t_j)$ be the experimental conditional probability by mutating design factor t_j, and let $P(t_j)$ be the theoretical probability with equal likelihood to other factors. Our early stopping criteria function should converge when $P_{D_i}(D_i^u \mid t_j)$ is close enough to $P(t_j)$. We use $H(D_i)$—the Shannon entropy [SHA01], a widely used approach in information theory for quantifying uncertainty—to formulate this concept. That means we will terminate the DSE process for a partition at iteration i if we have a low enough uncertainty of finding a better result in that partition at the next iteration. Formally, our early stopping criteria with the Shannon entropy is defined as follows.

$$\begin{aligned} |H(D_i) - H(D_{i-1})| &\leq \theta \\ H(D_i) &= -\sum_j P_{D_i}(D_i^u \mid t_j) \log P_{D_i}(D_i^b \mid t_j) \end{aligned} \tag{3.7}$$

where θ is the threshold for termination. Note that this metric has also been used in other fields such as image processing [RTS12]. In practice, we terminate the DSE process after the entropy difference is lower than θ for consecutive N iterations to avoid pulses. As we will illustrate in the next section, this systematic criteria works better than the trivial one that simply stops the process if a better result cannot be

found for a number of iterations.

3.3.5 Experimental Results

The experimental setup used for this framework is same as the previous one and the overall experimental results are shown in Table 3.5 as well as Table 3.6. When compared with v1, the improved framework is able to find the decent design point for almost all cases in 4 hours, except for `SPMV` and `CONV`. Since `SPMV` (sparse matrix-vector multiplication) is a memory-bounded design with very limit data reuse, the impact of the data tiling size on performance is much higher than any other design parameters. However, it is hard for meta-heuristic algorithms to identify a single important parameter in a few iteration, so it fails to achieve a decent performance. On the other hand, `CONV` has 40 design parameters and the largest design space among all cases, so it is hard for general hyper-heuristic approach to achieve decent performance with just hundreds of iterations.

Next, we evaluate the DSE process of the improved framework in Figure 3.6. The solid lines in sub-figures represents the DSE process of the improved framework. In summary, the DSE process saves 32.8% execution time on geometric mean while achieving 1.8× performance improvement over the previous version. We analyze the effectiveness of our optimization strategies as follows. First, we can find that almost all solid lines start earlier than the previous version, and some of them even have much better starting performance, such as `GEMM` and `KMP`. This illustrates the effectiveness of seed generation, since the area-driven seed is always synthesizable, and the performance-driven seed may reach high-performance in one iteration. Second, the DSE process of v2 grows faster than v2 in most case due to an effective design

Table 3.5: Overall Comparison to CPU and Manual Designs

Benchmark	Design Space	Speedup over v1	Ratio to Manual (%)	Speedup over CPU
AES	2.81E+11	2.02	79.88	3015.35
NW	2.70E+10	1.00	95.80	3322.41
KMP	5.76E+03	1.00	100	9.65
GEMM	2.52E+09	1.83	100	16.25
SPMV	1.73E+04	1.64	25.57	0.44
STENCIL-2D	4.37E+11	1.24	100	0.42
STENCIL-3D	1.78E+08	18.43	97.47	2.58
BACKPROP	5.18E+04	1.00	100	7.71
KMEANS	1.67E+06	1.01	100	35.10
KNN	1.03E+05	1.90	73.64	6.99
PATHFINDER	1.66E+04	1.42	98.70	0.02
CONV	1.50E+28	2.89	48.62	28.49
Geometric Mean	6.80E+10	1.79	79.87	11.81

space partition.

Third, the v2 framework terminates the DSE process faster (∼2.68 hours on average) than the v1 (4 hours) due to the early stopping criteria. As a result, even the v1 framework is able to realize the same design point as v2 such as `KMP`, it still terminates the process after 4 hours due to the lack of an effective early stopping criteria. In addition, we also analyze the effectiveness of one straightforward stopping criteria that stops the DSE process if no better result were found for consecutive 10

Table 3.6: FPGA Resource Utilization

Benchmark	BRAM (%)	LUT (%)	FF (%)	DSP (%)
AES	72	28	4	0
NW	78	52	30	0
KMP	48	17	2	0
GEMM	51	33	6	30
SPMV	11	9	1	4
STENCIL-2D	13	7	2	8
STENCIL-3D	20	5	2	3
BACKPROP	47	22	5	5
KMEANS	49	30	10	26
KNN	43	45	1	41
PATHFINDER	4	7	2	1
CONV	70	34	22	30

iterations. It turns out that compared to the Shannon entropy criteria, the trivial stopping criteria terminates the process one hour later ($\sim$3.72 hours) with the similar performance.

3.3.6 Insights and Summary

Although the experimental results have demonstrated that the improved framework with several optimization strategies can find a much better design configuration compared to v1, the vanilla OpenTuner, it sill has a main challenge: sometimes the hyper-heuristic approach cannot find a design point with an acceptable performance

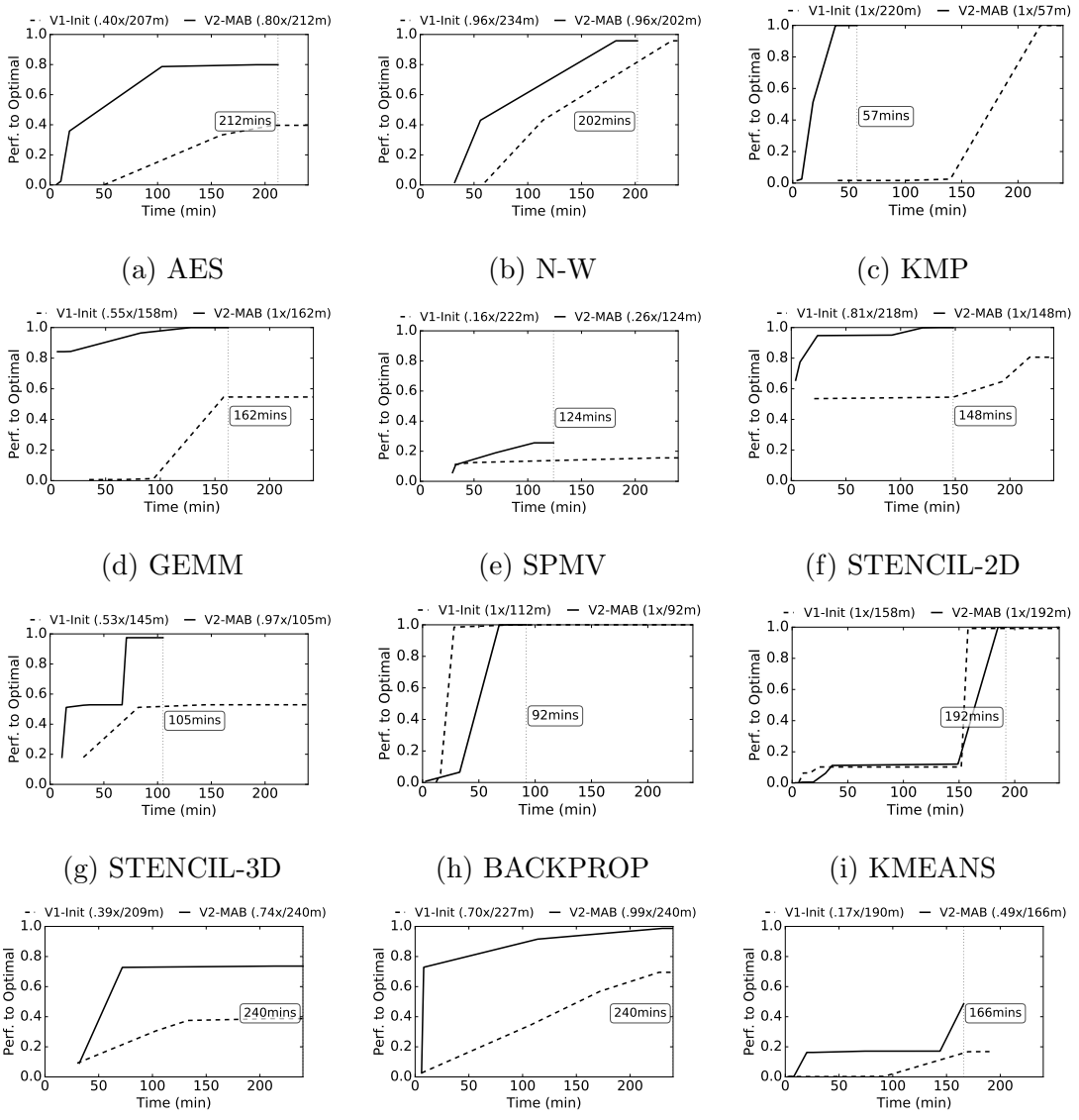


Figure 3.6: Design Space Exploration of the Improved Framework. The legend notes the best speedup over the manual design and the time to achieve it.

in the given time, as the search process is non-deterministic. Once the search algorithm fails to realize the best design point, it is hard for designers to reason the

performance bottleneck such as `SPMV` and `CONV` we discussed in the previous section, because 1) the explored design points are based on meta-heuristic algorithms that do not leverage any domain knowledge, and 2) the behavior of vendor HLS tools is unpredictable (e.g., Figure 3.1).

To seek for other opportunities of improving the search algorithm, we further analyze the exploration process as well as the manual optimal designs. We find that most designs have an obvious performance bottleneck (e.g., effective external memory bandwidth, insufficient parallel factors, etc.) which usually dominates more than a half of the overall execution cycle and is controlled by only one or two design parameters. It implies that the performance gain of tuning other parameters is often very limited. The meta-heuristic algorithm needs many iterations to identify the killer parameter and tune it to resolve the performance bottleneck. After that, it has to spend another large number of iterations again to find the next killer parameter. This phenomenon motivates us to develop a new search algorithm in the next section that is guaranteed to optimize the killer parameter prior to others.

3.4 Version 3: Stability Optimization

In order to make a systematic search algorithm for better stability and reasoning, we attempt to leverage the concept of gradient descent, because it always towards to the direction with the best gradient value so that we can easily track the process and make sure the performance could be improved from time to time. In this section, we first introduce the gradient descent with finite difference method in Section 3.4.1 that systematically finds a better design point in the design space. On the other hand, as we will illustrate in Section 3.4.1, simply adopting gradient approach causes a serious

local optimal problem and does not generate high-performance accelerator designs. As a result, we present several strategies from Section 3.4.2 to Section 3.4.5. Finally, we integrate all strategies in Section 3.4.6 and build the v3 framework.

3.4.1 Gradient Descent with Finite Difference Method

Gradient descent is a well-known iterative optimization algorithm for finding a local minimum point in a differentiable objective function, and it has also been successfully applied to solve large scale non-linear physical design problems with a smooth analytical approximation such as multi-level circuit placement [CCS05]. Formally, gradient descent is used to find a configuration θ with the minimal objective value $J(\theta)$ in a solution space $\mathbb{R}^K_{\mathcal{P}}$:

$$\underset{\theta_i \in \mathbb{R}^K_{\mathcal{P}}}{argmin} \mathbf{J}(\theta_i) \tag{3.8}$$

To achieve the goal, we start from an initial configuration θ_0, and iteratively update the configuration by following the steepest descent, the negative gradient $-\nabla \mathbf{J}$:

$$\theta_{i+1} = \theta_i - \alpha \nabla \mathbf{J}(\theta_i) \tag{3.9}$$

where α is the step size.

One of the most important limitations in gradient descent approach is that it requires the objective function to be differentiable in order to find the next steepest descent. This limitation, however, makes it impractical in many real-world applications, as the system may be too complicate to be modeled as partially observable

Markov decision problems. To avoid the potential problems (e.g., accuracy and portability) of modeling HLS tools, we leverage finite difference method in derivative-free optimization [CSV09] to approximate the gradient value by treating the HLS tool as a black-box. That is, given a candidate configuration θ_j perturbed from the current configuration θ_i, we use finite difference method to approximate the gradient as follows:

$$g(\theta_j, \theta_i) \sim \frac{Cycle(\mathbf{H}, \mathcal{P}(\theta_j)) - Cycle(\mathbf{H}, \mathcal{P}(\theta_i))}{Util(\mathbf{H}, \mathcal{P}(\theta_j)) - Util(\mathbf{H}, \mathcal{P}(\theta_i))} \tag{3.10}$$

Note that Equation 3.10 considers not only performance gain but resource efficiency so it could reduce the possibility of being trapped in a local optimal. For example, we may reduce 10% execution cycle by spending 30% more area if we increase the parallel factor of a loop (configuration θ_1); we can also reduce the 5% execution cycle by spending 10% more area if we enlarge the bit-width of a certain buffer (configuration θ_2). Although θ_1 seems better in terms of the execution cycle, it may be trapped by a local optimal point easier because it has a relatively limited resource to be further improved. On the other hand, the finite difference values for the two configurations are $g(\theta_1, \theta_0) = \frac{-10\%}{30\%} = -0.3$ and $g(\theta_2, \theta_0) = \frac{-5\%}{10\%} = -0.5$, so the system prioritizes the second configuration for a better long-term performance.

Since finite difference method selects the best candidates as the next configuration, we need to generate a set of candidates, Θ_{cand}, at each iteration. Specifically, we generate candidates by advancing the value of each parameter in the current configuration by one step. Formally, the c-th candidate generated from θ_i is:

$$\theta_c = [p_0, p_1, ..., p_c + 1, ..., p_k] \tag{3.11}$$

where p_c is the value of c-th parameter in θ_i. Accordingly, we will generate K candidates at each iteration, which means we use K HLS runs to determine the next configuration:

$$\theta_{i+1} = \underset{\theta_j \in \Theta_{cand}}{argmin}\ g(\theta_j, \theta_i) \tag{3.12}$$

By leveraging the gradient descent with a finite difference method, we expect to find a better design point every K HLS runs. Unfortunately, as we have illustrated in Figure 3.1, the performance trend is not always smooth, so the gradient process is easily trapped by a low-quality local optimal design point. Taking Figure 3.1 again as an example, the gradient approach will stop at factor 2 for `FG-loop-1` because factor 3 has worse performance but costs more resources. Actually, the gradient approach proposed in this section only achieves 0.86× speedup on the geometric mean of our benchmark, which is even worse than results from v1. Consequently, we propose several strategies in the remainder of this section to improve the efficiency.

3.4.2 Graph-based Design Space Pruning

One solution to facilitating the gradient process is to reduce ineffective parameters. A straightforward way to build a design space from Merlin pragmas is treating each Merlin pragma as a design parameter, but it is inefficient. For example, if we have determined the outermost loop in a loop nest that needs to be performed the memory

burst, then the Merlin compiler will tile that loop, create local buffers with the tile size and insert `memcpy` to enable memory burst before the loop body. In this case, the physical meaning of the tiled outermost loop is to transfer a batch set of data from DRAM to BRAM, which cannot be executed in parallel. As a result, memory burst and parallel pragmas are mutually exclusive in a loop nest. To avoid this inefficient design points, we propose a graph-based algorithm to create a design space that is capable of reflecting such characteristics.

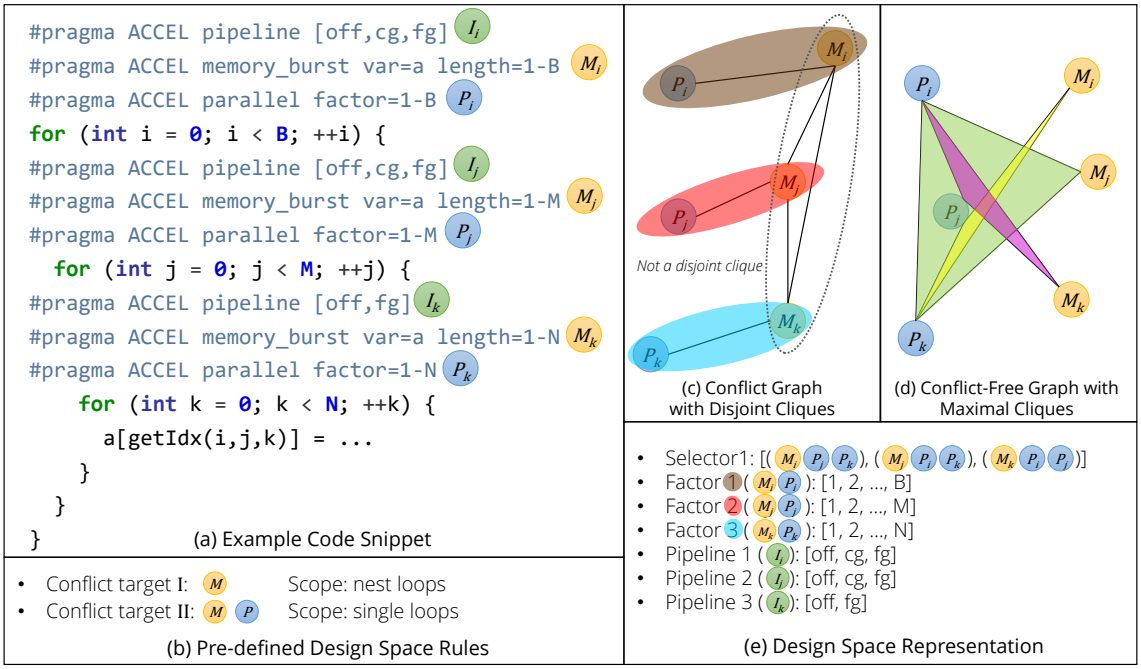


Figure 3.7: Graph-based Design Space Building Approach

Our proposed approach is illustrated in Figure 3.7. Figure 3.7a is a loop nest example with all possible Merlin pragmas (each pragma exists an option to remove itself). In Figure 3.7b, we specify the rules mentioned above. By accepting both Figure 3.7ab as inputs, our approach first builds a conflict graph of which the vertices

are pragmas and the edges mean the two pragmas cannot be existed at the same scope (either a single level loop or a loop nest). Since we only need to explore one instead of N parameters if N pragmas in a scope are mutually exclusive, we find the minimum number of maximum disjoint cliques in the graph to realize what pragmas can be explored using one uniform parameter. For example in Figure 3.7c, the factor of pragma P_i and M_i can always be the same because of the conflict target II defined in Figure 3.7b. After this step, we create three factor parameters as shown in Figure 3.7e for this loop nest.

Subsequently, we find the minimum number of options of selecting exclusive pragmas. To do so, we build a conflict-free graph by complementing the conflict graph, as shown in Figure 3.7d by complementing Figure 3.7c. As a result, we create a selector parameter for DSE in Figure 3.7e by finding all maximum cliques, which means we select as many pragmas as we can. As can be seen in this example, with the graph-based approach, we greatly reduce the design space from $3 \times B^2 \times 3 \times M^2 \times 2 \times N^2 = \mathcal{O}(B^2M^2N^2)$ in Figure 3.7a to $3 \times B \times 3 \times M \times 2 \times N \times 3 = \mathcal{O}(BMN)$ in Figure 3.7e. According to our evaluation result, this approach reduces on average $\sim$24.65$\times$ design space.

3.4.3 Design Space Partition

Another solution to address the local optimal issue caused by the non-smooth performance gain is partitioning the design space based on likely distribution of local optimal points and exploring each partition independently. Since we already have design space partition mechanism in v2, we integrate the observations found in this chapter and improve the partition rule that partitions the design space based on the

pipeline mode, as pipeline mode `fg` unrolls all sub-loops to achieve fine-grained while the mode `cg` uses double buffers to implement coarse-grained pipeline. These two modes apparently have the most significant different influence on the generated architecture and are expected to have non-related performance and resource utilization. According to the pipeline modes in each loop, we use tree partition and generate 2^N partitions from a design space with N non-innermost loops.

Supposing we use use t working threads to perform at most h hours DSE for 2^N design space partitions, we need $\frac{2^N}{t} \times h$ hours to finish the entire process. On the other hand, some partitions that are based on an insignificant pipeline pragma may have the similar performance, so it is more efficient to only explore one of them. As a result, we profile each partition by running HLS with minimized parameter values to obtain the minimum area and performance, and use K-means clustering with performance and area as features to identify t representative partitions among all 2^N partitions.

3.4.4 Adaptive Line Search

After partitioning the design space, we are able to avoid the gradient process to be terminated at the early stage due to dramatic performance difference between pipeline modes. On the other hand, the performance trap of consecutive parameter values caused by vendor HLS tools, as shown in Figure 3.1, is still an impediment of finding a better result.

By observing Figure 3.1, we realize that the relationship between factors and execution cycle is a negative correlation when we only consider the power-of-two numbers. This is reasonable because the vendor HLS tools usually apply many

heuristics to synthesize and schedule loops when the parallel factor is not power-of-two. As a result, we prefer to let the gradient process explore power-of-two values prior to others so that the objective function could be smooth in the beginning. This idea is inspired by the concept of line search strategy [BER99] which uses an adaptive step size during the search process and has also been adopted for VLSI circuit placement (e.g. [KW05]), so the Equation 3.11 can be refined as follows:

$$\theta_c = [p_0, p_1, ..., p_c + s, ..., p_k] \tag{3.13}$$

where s is an adaptive step-size. In detail, we first set s as a large number to make a large step to the power-of-two factors of `parallel`, `memory burst` and `memory coalescing`. When there is no valid candidate to be selected by the finite difference method, we reduce the step size by 2 and re-generate candidates.

3.4.5 Multiscale V-Cycles

Since our gradient approach tries one adaptive step on every parameter and changes one parameter at a time, it assumes every parameter can be tuned individually. On the other hand, some parameters in our design space may have a strong dependency to others, so changing one of them would not be effective. For instance in Code 3.1, since both `loop-i` and `loop-j` access array `A`, the performance of both loops are affected by array `A`'s partition factor, which is inferred automatically by the Merlin compiler according to their parallel factors. It is obvious that array `A` should be partitioned cyclically by 4 in Code 3.1, because both loops are partially unrolled by 4 times. When we increase the parallel factor of `loop-i` to 5, on the other hand, this problem becomes nontrivial. Theoretically, cyclically partitioning array `A`

Code 3.1: Example of Dependent Loop Parallel Factors

```
1 #pragma ACCEL parallel factor=4
2 for (i = 0; i < N; ++i)
3   A[i] = ...;
4 #pragma ACCEL parallel factor=4
5 for (j = 0; j < N; ++j)
6   ... = A[j];
```

by 5 to match `loop-i`'s parallel factor will not encounter bank conflict at `loop-j`, since the bank access order is `0,1,2,3` when $j = 0$ and `4,0,1,2` when $j = 1$. However, it is possible that the vendor HLS tool fails to recognize this pattern and considers that there has a bank conflict at `loop-j`. In this case, the HLS tool sacrifices the performance to guarantee the functionality by increasing the loop II. Since this phenomena is case by case, the most promising solution is making sure the parallel factors of `loop-i` and `loop-j` are always the same, but it may ignore some corner cases and loses the optimality.

The optimization strategy we applied for solving this problem is multiscale V-cycles [CS13], which was widely used in VLSI physical design problems such as partitioning (e.g., [HME]) and placement (e.g., [CCK03]). The idea is that we first coarsen the loop parallel parameters that access the same arrays as a parameter cluster, which reduces the number of tuning parameters from 2 (parallel factors of `loop-i` and `loop-j`) to 1 in the above example. Then we process the gradient with parameter clusters. When the gradient approach is trapped due to local optimal, we relax the clusters and continue the process. The process of coarsening and relaxing forms a V-cycle. Note that although relaxing parameter clusters increases the parameter number and slows the gradient process in later iterations, we already stand

on a decent solution.

3.4.6 Putting It All Together

We summarize the above strategies and present an optimized gradient descent with finite difference method in Algorithm 3. The algorithm takes a program as well as its design space partitions as inputs. For each partition, it first sets a large initial step (line 4) and coarsens design space parameters by analyzing data access patterns (line 5). In each gradient iteration, the algorithm moves all parameters in the same cluster to generate candidate points (line 8-19), and evaluate the points using a vendor HLS tool in parallel (line 20). After that, it checks the results and commits the move with the best finite difference value (line 21-29). When there is no valid finite difference value, we relax the cluster (line 23) and reduce the step size (line 24) to further refine the solution. The algorithm finally outputs the overall best design point among all partitions.

Algorithm 3 Optimized Gradient Descent with Finite Difference

Require: A C program $\mathcal{P}$ and a set of design space partitions $\mathbb{S}$.

Ensure: A design configuration θ with the best QoR.

1: $bestPoints \leftarrow \varnothing$
2: **for all** $S \in \mathbb{S}$ **do**
3: $\quad currPoint \leftarrow Evaluate(GetDefaultPoint(S))$
4: $\quad stepSize \leftarrow GetInitStep()$
5: $\quad paramClusters \leftarrow CoarsenParams(BuildAST(\mathcal{P}), S)$
6: $\quad$ **while do**
7: $\quad\quad pendingQueue \leftarrow \varnothing$
8: $\quad\quad$ **for** $cluster \in paramClusters$ **do**

```
 9:          cfg ← CopyConfig(currPoint.cfg)
10:          move ← false
11:          for param ∈ cluster do
12:            if MoveSteps(cfg, param, stepSize) = true then
13:              move ← true
14:            end if
15:          end for
16:          if move = true then
17:            pendingQueue.append(cfg)
18:          end if
19:        end for
20:        pointSet ← ParallelEvaluate(pendingQueue)
21:        bestPoint ← GetBestMove(pointSet)
22:        if bestPoint = ∅ then
23:          paramClusters ← RelaxClusters(paramClusters)
24:          if ReduceStepSize(stepSize) = false then
25:            break
26:          end if
27:        else
28:          currPoint ← bestPoint
29:        end if
30:      end while
31:      bestPoints.append(currPoint)
32:    end for
33:    return GetBest(bestPoints).cfg
```

Finally, we integrate the proposed method to the DSE framework in Figure 3.8. The framework first builds a design space according to Table 3.1. Then, it profiles and selects representative partitions using K-Means. For each partition, the explorer performs DSE using the proposed gradient approach. Note that we could configure the framework to leverage only the gradient approach or other meta-heuristic algorithms. When the explorer finishes exploring a partition, it stores the best configuration found by that partition and reallocates the working threads to other partitions to keep the high resource utilization. Finally, when all partitions are finished, the framework outputs the design configuration with the best performance among all partitions.

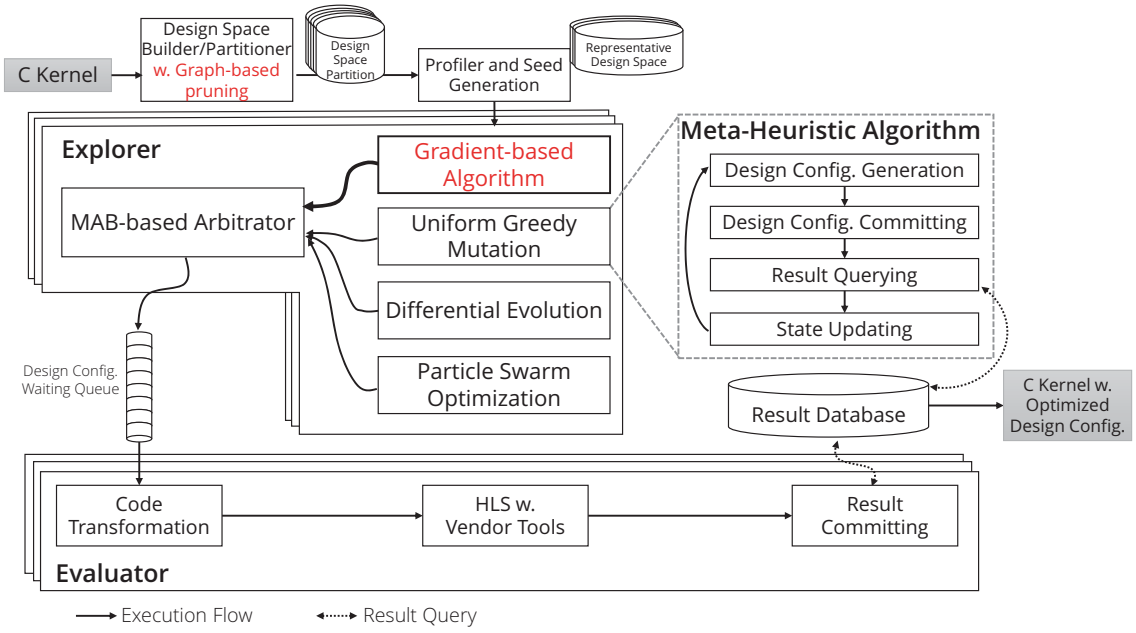


Figure 3.8: The Framework with Gradient-based Approach

3.4.7 Experimental Results

We again use the same experimental setup in Section 3.2.4.1. In this experiment, we first evaluate the effectiveness of the proposed optimization strategies for the gradient descent algorithm, followed by the overall evaluation of the performance and DSE process comparing to previous versions.

3.4.7.1 Evaluation of Gradient Approach Optimization

Figure 3.9 presents the accumulated performance improvement by applying the optimization strategies step by step. The leftmost bar is the vanilla gradient descent with finite difference method. After applying the graph-based design space pruning, the DSE only explores the effective memory burst candidates (single memory burst per loop nest) so the number of memory burst candidates are greatly reduced. In fact, the overall design space is reduced by 24.65× on average among all cases after applying the proposed algorithm, which results in 1.3× performance improvement on geometric mean, as shown in the second from the left bar.

We then evaluate the gradient descent with a finite difference method and proposed optimization strategies in 3^{rd} to 5^{th} bars of Figure 3.9. We can see that each of the proposed strategies benefits at least one case in our benchmark. After applying design space partition, the geometric mean speedup is improved by 2.1×. In particular, design space partition benefits the designs with many nest loops in which the gradient process is easily trapped by the local optimal when changing pipeline modes—such as `AES`, `GEMM`, `N-W`, `STENCIL-2D`, and `STENCIL-3D`.

In addition, after applying adaptive line search (ALS), the performance is further improved by 1.9×, especially for `AES`, `N-W`, `SPMV`, `STENCIL-2D`, `STENCIL-3D`, `KMEANS`,

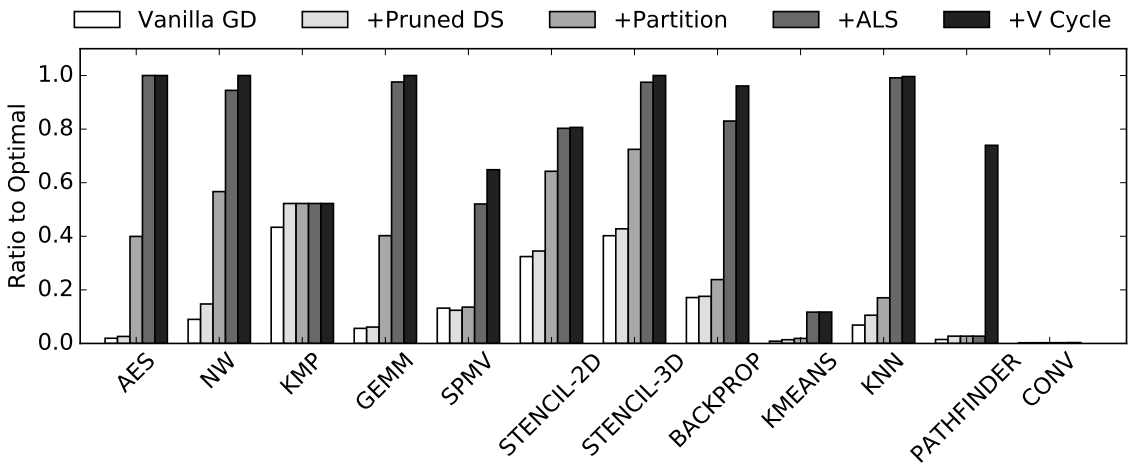


Figure 3.9: Step-by-Step Performance Improvement with Gradient Approach

and `KNN`. As we have illustrated in the beginning of this chapter, adaptive line search helps the gradient process avoid irregular finite difference values and results in a better performance. Finally, we can see from Figure 3.9 that the multiscale V-cycle significantly improves the performance of `PATHFINDER`, because this design use multiple loops to process the same array buffer and result in a high impact of array partition factor on performance. In other words, changing one factor at a single step will not affect the overall performance.

3.4.7.2 Overall Evaluation and Analysis

The overall performance and resource utilization are shown in Table 3.7 and Table 3.8, respectively. We can see that most the gradient approach results match the manual design performance. For `KMP`, the optimal is achieved by 32 process units (PEs) as well as 512-bit memory coalescing. Although the gradient approach does coalesce the off-chip buffers to improve the bandwidth, it fails to achieve 32 PEs due

to the local optimal, because we find that the efficiency of 16 PEs is worse than 8 PEs which results in a low gradient value. Therefore, the gradient approach is trapped by 8 PEs and cannot achieve the optimal performance.

Table 3.7: Overall Comparison to CPU and Manual Designs

Benchmark	Design Space	Speedup over v2	Ratio to Manual (%)	Speedup over CPU
AES	3.11E+09	1.25	100	3774.69
NW	1.51E+09	1.04	100	3468.11
KMP	5.76E+03	0.52	52.24	5.04
GEMM	1.26E+09	1.00	100	16.25
SPMV	5.76E+03	2.98	76.29	1.32
STENCIL-2D	9.70E+09	0.81	80.64	0.34
STENCIL-3D	1.94E+06	1.03	100	2.65
BACKPROP	1.15E+04	1.00	100	7.71
KMEANS	2.49E+05	0.11	11.72	4.12
KNN	1.90E+04	1.35	99.61	9.46
PATHFINDER	5.18E+03	1.01	100	0.18
CONV	1.50E+28	0.01	0.32	0.19
Geometric Mean	1.26E+08	0.59	47.16	6.97

For `SPMV`, although the gradient approach dose not achieve the optimal performance, it has already achieved 77% that is much better than v3 (26%). This is because the gradient approach is able to identify the killer parameter, the data tiling size, and optimize this parameter prior to others. On the other hand, the reason of

Table 3.8: FPGA Resource Utilization

Benchmark	BRAM (%)	LUT (%)	FF (%)	DSP (%)
AES	31	11	2	0
NW	40	18	12	0
KMP	72	24	5	0
GEMM	52	33	6	30
SPMV	72	35	4	19
STENCIL-2D	4	2	1	2
STENCIL-3D	9	5	4	7
BACKPROP	47	24	5	6
KMEANS	50	31	3	27
KNN	72	64	17	22
PATHFINDER	7	7	2	1
CONV	50	2	1	2

the rest 23% gap comes from the resource allocation. Since the optimal design uses only 8 PEs but improves the PE throughput by having 16 as the unroll factor of the innermost loop with reduction enabled. However, duplicating 16 PEs has higher finite difference value than unrolling the inner loop by 16 times when reduction is disabled. As a result, the gradient towards to duplicating more PEs and is trapped by the local optimal, because our V-cycle analysis does not group the unroll factor and the reduction flag when coarsening the design space. For `KMEANS`, although it is not a memory-bounded design like `SPMV`, it also requires the data tiling size to be optimized before improving the design throughput. However, the gradient approach

has to run one design point for every parameter but update only one parameter at each iteration. Consequently, it only performs 8 iterations in the 4 hour time limit and fails to achieve the large enough data tiling size. This problem is more critical for `CONV`, since it has 40 design parameters. Since the gradient approach needs to evaluate 40 design points to make a move, it only performs 4 iterations in 4 hours and results in an even worse performance than v2.

We finally evaluate the DSE process in Figure 3.10. The gradient approach outperforms the previous two version in almost all cases in turns of the achieved performance and DSE time. This proves the conclusion we made in the previous chapter about the search algorithm would be effective if we focus on identifying the killer parameter.

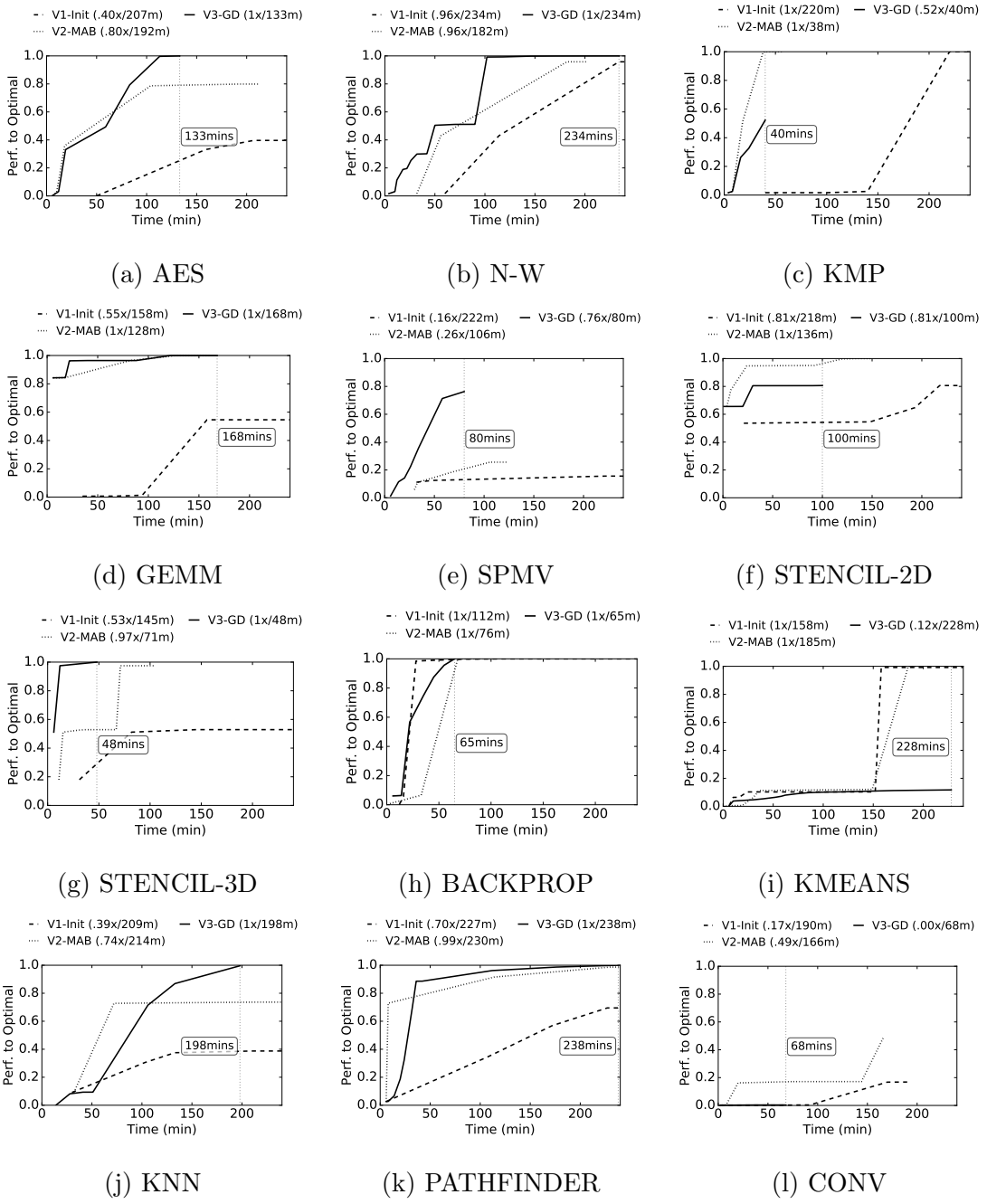


Figure 3.10: Design Space Exploration with Gradient-based Algorithm. The legend notes the best speedup over the manual design and the time to achieve it.

3.4.8 Insights and Summary

In this section, we demonstrate that we could leverage gradient descent with several HLS-specific optimization strategies to perform a more systematic DSE than the hyper-heuristic algorithm, multi-armed bandit approach. For the designs with a tremendous design space, MAB may fall into random search since the path to a better performance is unclear. It is challenging especially when the DSE fails to achieve high performance, because designers will have no clue about where can be further improved. On the other hand, as we have illustrated in the experimental result, we can easily reason the root cause of failing to achieve the optimal performance for each design by tracing the gradient process. This helps us not only manually bridge the performance gap for our designs, but also identify the opportunities to further improve the search algorithm. On the other hand, we summarize the remaining challenges of the gradient approach as follows, and the refined DSE framework is presented in the next section by addressing those challenges.

Challenge 1: Non-smooth design space: Although we apply graph-based pruning algorithm to reduce the design space, the reduced design space is not smooth enough. In particular, the generated selector parameters may result in dramatically changing of the design, since they invalid exclusive parameters.

Challenge 2: High evaluation cost for each iteration: Since we move one step of every parameter to approximate the gradient value with finite difference method, we have to evaluate N design points in each iteration, where N is the total number of design parameters. As a result, the evaluation cost for each iteration is proportional to the parameter number and it is not scalable (e.g., `CONV`).

Challenge 3: Potential local optimal: Although adaptive line search resolves

significantly improves the local optimal problem, it may still happen (e.g., `SPMV`) especially between different factors of a design parameter.

3.5 Version 4: Scalability Optimization

We address the challenges summarized in the previous section by improving the design space representation as well as the search algorithm. For *Challenge 1*, we improve design space representation in Section 3.5.1 by preserving all design space dimensions but invaliding infeasible design points so that the exploration process can be smooth. For *Challenge 2*, we improve the design point evaluator to support cycle breakdown analysis and performance bottleneck analysis in Section 3.5.2. Finally, we refine the search algorithm in Section 3.5.3 to focus on high impact parameters while avoiding potential local optimal issues.

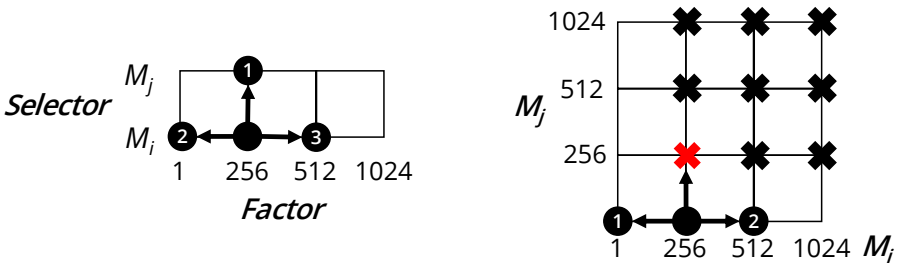
3.5.1 Comprehensive Design Space Representation

One major problem of the graph-based algorithm proposed in Section 3.4.2 is that it prunes the design space according to a predefined constraint by creating a selector to eliminate infeasible parameter combinations. We use Figure 3.11 to further illustrate this problem. Figure 3.11a is an example code snippet. In this example loop nest, we attempt to explore the best position of a memory burst pragma and its value, so the pragma M_i and M_j are exclusive, and only one of them should be inserted at a time. With the graph-based algorithm, we create a design space in Figure 3.11b. As can be seen, we merge the factor of M_i and M_j and create a selector to indicate the targeting memory burst pragma. Assume that we are at the configure $(Selector, Factor) = (M_i, 256)$ and have three candidates to be explored by the gradient-based approach.

Among all candidates, candidate 2 and 3 are expected to have continuous result qualities that are suitable for gradient, but this is not the case for candidate 1, since it changes the memory burst position and usually has a high impact on performance and resource utilization.

```
#pragma ACCEL memory_burst var=a length=Mi // Options: 1,256,512,1024
for (int i = 0; i < 1024; ++i) {
#pragma ACCEL memory_burst var=a length=Mj // Options: 1,256,512,1024
  for (int j = 0; j < 1024; ++j) {
    a[getIdx(i,j)] = ...
  }
}
```

(a) An Example Code Snippet



(b) Graph-based Approach

(c) Proposed Approach

Figure 3.11: Different Design Space Representations and Their Impact on DSE

A better design space representation, on the other hand, preserves the original design space dimensions but invalids infeasible points. An example is presented in Figure 3.11c. Again, we are at the configure $(M_i, M_j) = (256, 1)$, but we only have two candidates this time because configure $(M_i, M_j) = (256, 256)$ is invalid. In summary, although the feasible design space in Figure 3.11b and Figure 3.11c are equivalent, the representation in Figure 3.11c is more exploration friendly and easier to reason.

We borrow the syntax of Python list comprehensions to represent the design space and achieve the above goal. Python list comprehensions are a concise approach for creating lists. They have the following syntax:

```
list_name = [expression for item in list if condition]
```

and this representation is equivalent to:

```
for item in list:
    if condition is True:
        list_name.append(expression)
```

Formally, we define the design space representation for Merlin pragmas with list comprehensions as follows:

```
#pragma ACCEL <pragma-type> <attribute-key>=auto{
  options: parameter_name=list-comprehension-expression;
  default: default-value
}
```

Taking Figure 3.11a as an example, the design space can be represented using list comprehensions as follows:

```
1  #pragma ACCEL memory_burst var=a length=auto{
2    options: Mi = [x for x in [1,256,512,1024] if Mj==1];
3    default: 1
4  }
5  for (int i = 0; i < 1024; ++i) {
6  #pragma ACCEL memory_burst var=a length=auto{
7    options: Mj = [x for x in [1,256,512,1024] if Mi==1];
8    default: 1
9    }
10   for (int j = 0; j < 1024; ++j) {
11     a[getIdx(i,j)] = ...
12   }
13 }
```

where lines 2 and 7 indicate that the two memory burst pragmas are exclusive. In

other words, when we set $M_i = 256$, the available option for M_j is only the default value, which is 1 in this case.

There are two main advantages to adopting list comprehension-based design space representations. First, the Python list comprehension is general and can represent any list. It provides a friendly and comprehensive interface with higher levels such as polyhedral analysis [ZLC13] and domain-specific languages to generate an effective design space. Second, the syntax of this representation is Python compatible. It means we can directly leverage the Python interpreter to evaluate the design space and improve overall stability of the DSE framework.

3.5.2 Performance Bottleneck Analysis

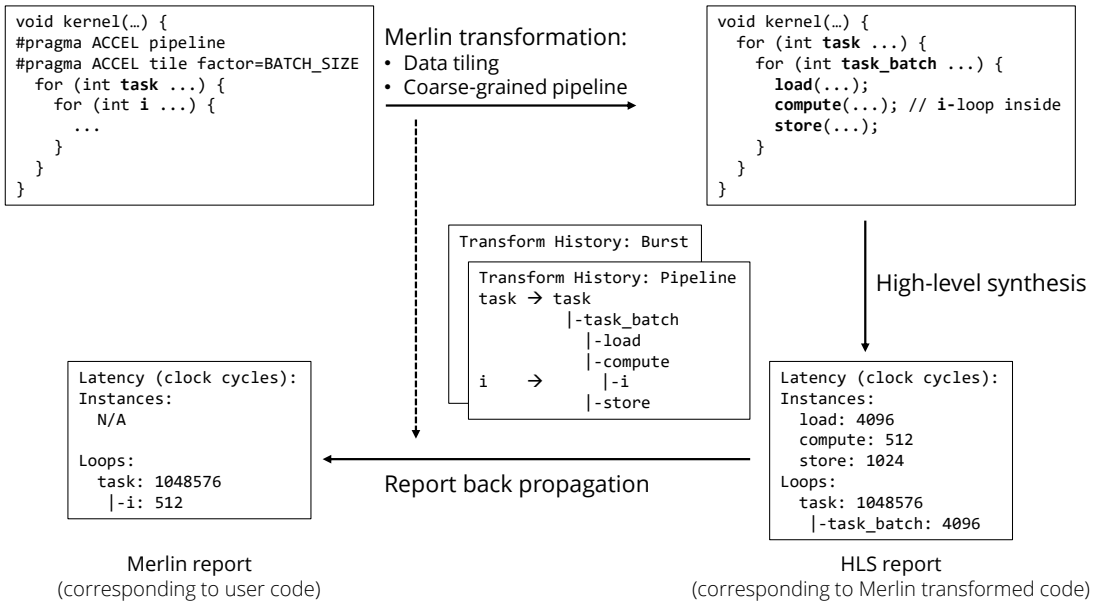


Figure 3.12: Merlin Compiler Report Generation

As we pointed out in *Challenge 2*, the efficiency of using the gradient-based

approach for DSE is limited by the process of approximating gradient value. Specifically, at each iteration, the gradient approach has to evaluate N design points, where N is the total number of tuning parameters, to determine the next step, because we treat the HLS tool as a black-box and only fetch the overall design latency. In fact, the Merlin compiler [CHP16a] includes a feature that performs back propagation to propagate the performance breakdown reported by the HLS tool to the user input code. Figure 3.12 illustrates its process. When performing code transformation, the Merlin compiler records the code change step by step so that it is able to propagate the latency estimated by the HLS tool back to the user input code. This feature is helpful for the DSE framework to analyze the performance bottleneck and identify the killer design parameter by running HLS for only one design point.

Specifically, we identify the performance bottleneck by traversing the Merlin report using depth-first search (DFS). Details of the algorithm are shown in Algorithm 4. The algorithm starts with the kernel top function statement. We first check to see if the current statement has child loop statements (line 1). For the function call statements, we dive into the function implementation to further check its child statements (line 3). Then we traverse each of them and create hierarchy paths (lines 10-12). Note that since we sort all loop statements according to their latency by checking the Merlin report (lines 8-9), the hierarchy paths we created will also be sorted by their latency. Subsequently, we check the Merlin report again to realize whether the performance bottleneck of the current statement is memory transfer or computation (lines 14-18). The Merlin compiler obtains this information by analyzing the transformed kernel code along with the HLS report. A cycle is considered to be a memory transfer cycle if it is consumed by communicating to off-chip memory. Finally, we append the current statement to the end of each path (lines 22-24) and

return a list of paths in order. With Algorithm 4, we can not only figure out the performance bottleneck for each design point, but we can also identify a small set of effective design parameters to focus on. As a result, we are able to significantly improve the efficiency of our searching algorithm in the next section.

Algorithm 4 Depth-First Search of Design Bottleneck Analysis

Require: A Merlin performance report Rpt, loop hierarchy $Hier$, and current statement $currStmt$.

Ensure: An ordered list of critical paths CP and bottleneck (memory or compute).

if $!Hier(currStmt).hasChild()$ **then**
  if $currStmt.isFuncCall()$ **then**
    $CP \leftarrow DFS(Rpt, Hier, Hier(currStmt).getFuncDecl())$
  else
    $CP \leftarrow \varnothing$
  end if
else
  $child \leftarrow Hier(currStmt).getChild()$
  $child \leftarrow sortByLatency(child, RptRp)$
  for all $c \in child$ **do**
    $CP.append(DFS(Rpt, Hier, c))$
  end for
end if
if $Rpt(currStmt).memoryCycle() > Rpt(currStmt).computeCycle()$ **then**
  $bottleneck \leftarrow MEMORY$
else
  $bottleneck \leftarrow COMPUTE$

18: **end if**

19: **if** $CP = \varnothing$ **then**

20: $CP \leftarrow List((currStmt, bottleneck))$

21: **else**

22: **for all** $path \in CP$ **do**

23: $path.append((currStmt, bottleneck))$

24: **end for**

25: **end if**

26: **return** CP

3.5.3 Bottleneck Optimization Approach

We summarize again the inefficiencies of the gradient-based DSE approach proposed in the previous section (Algorithm 3) by comparing its behavior with human design experts:

1. The gradient-based approach has to evaluate many design points to identify the performance bottleneck. An expert could directly acquire this information by analyzing the cycle break.

2. The gradient-based approach has no knowledge about parameters, so it has no way to prioritize important parameters. An expert may know which parameter has a high potential of being the killer parameter.

3. The gradient-based approach may stop exploring the options of a parameter due to local optimal, An expert may know whether other options are worthwhile to explore or not.

The first two inefficiencies can resolved by leveraging the bottleneck analysis. We first build a map from loop or function statements in the user input code to design parameters so that we know which parameters should be focused for a particular statement. When we obtain an ordered list of critical hierarchy paths from the bottleneck analysis, we start from the most critical innermost loop statement and identify its corresponding parameters. Note that since the bottleneck analysis also provides the bottleneck type information (i.e., memory transfer or computation), we may identify a subset of the parameters mapped to that statement. For example, we may have design parameters of `PARALLEL`, `PIPELINE`, and `MEMORY_BURST` at the same loop level. When the bottleneck type of the loop is memory transfer, we focus on the `MEMORY_BURST` parameter for the loop; otherwise we focus on `PARALLEL` and `PIPELINE` parameters. In other words, we reduce the number of candidate design parameters not only by the bottleneck statement but the bottleneck type.

For the third inefficiency, we cannot identify whether the current option of a parameter is local or global optimal, so the most promising solution is breaking the dependency between options and searching a set of them in parallel. In this way, although we still need to evaluate multiple design points at every iteration, we guarantee that each design point can provide the maximum information for improving the performance because we always evaluate the options of the parameter that has the largest impact on the performance bottleneck. The refined algorithm is presented in Algorithm 5 with the following descriptions of the data structures used, and the complete v4 DSE framework is presented in Figure 3.13.

Algorithm 5 Refined Search Approach for Bottleneck Optimization

---

Require: A C program $\mathcal{P}$ and a set of design space partitions $\mathbb{S}$.

Ensure: A design configuration θ with the best QoR.

1: $topFunc \leftarrow GetTopFunction(\mathcal{P})$
2: **for all** $S \in \mathbb{S}$ **do**
3: $cfg, report, hier \leftarrow Evaluate(GetDefaultPoint(S))$
4: $FocusParamsWOptions \leftarrow BottleneckAnalysis(report, hier, topFunc)$
5: $LevelHeap \leftarrow \varnothing$
6: $LevelHeap.append(\varnothing)$
7: $LevelHeap[0].push(DesignPoint(0, cfg, FocusParams, \varnothing))$
8: **while** $LevelHeap \notin \varnothing$ **do**
9: $CurrLevel = GetLastLevel(LevelHeap)$
10: $CurrPoint \leftarrow LevelHeap[curr_level].peek()$
11: $CurrParamWOptions \leftarrow CurrPoint.popParam()$
12: $candidates \leftarrow \varnothing$
13: **for all** $option \in CurrParamWOptions$ **do**
14: $NewCfg \leftarrow Manipulate(CurrPoint, CurrParamWOptions, option)$
15: $candidates.append(NewCfg)$
16: **end for**
17: $ParallelEvaluate(candidates)$
18: **for all** $cfg, report, hier \in candidates$ **do**
19: $FocusParamsWOptions \leftarrow BottleneckAnalysis(report, hier, topFunc)$
20: $FD \leftarrow CalFiniteDifference(report)$
21: $NewPoint \leftarrow DesignPoint(FD, cfg, FocusParamsWOptions)$
22: $LevelHeap[curr_level + 1].push(NewPoint)$
23: **end for**
24: **if** $LevelHeap[currLevel].FocusParamNum() = 0$ **then**
25: $LevelHeap[currLevel].pop()$

26: **end if**

27: **end while**

28: **end for**

29: **return** $GetBestCfg()$

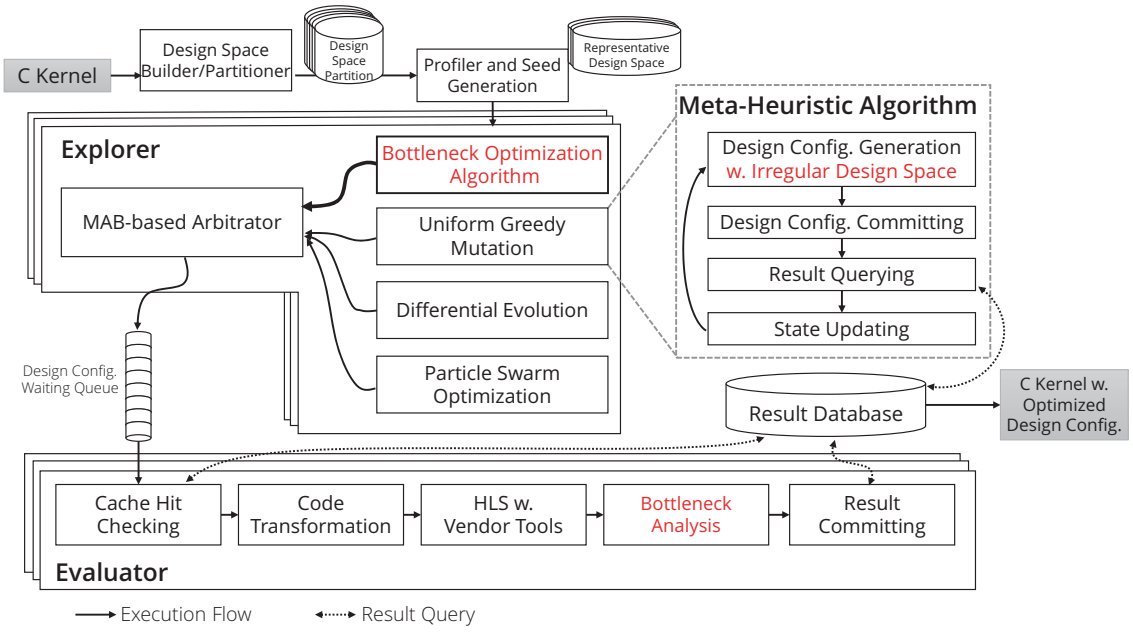


Figure 3.13: The Framework with Hotspot Optimization Approach

- ***LevelHeap***: A heap for each level of pending design points that can be further explored. Note that level n means we have fixed the value of n parameters, so the maximum level in this algorithm is equal to the total number of parameters. Since new design points are sorted by their finite difference values when they were pushed into the heap, the design point with a better finite difference value will be explored prior to other points.

- ***DesignPoint***: The data structure of a design point includes its 1) finite difference value, 2) configuration, 3) focused parameters, and 4) fixed parameters.

- ***ParamWOptions***: The data structure of a set of design points that are generated from a reference design point by mutating a certain design parameter. Note that the available options of a parameter is determined based on the reference point as we have illustrated in Figure 3.11c.

3.5.4 Experimental Results

With the same experimental setup as previous versions, the overall performance and resource utilization are presented in Table 3.9 and Table 3.10, respectively. With the same design space and exploration time, we can seen that the refined bottleneck optimization algorithm is able to realize the design points that match the optimal performance. In particular, `CONV` achieves 94% performance compared to the manual design. This is the best performance among all versions of the DSE frameworks that we have proposed in this chapter. Since `CONV` has the largest design space, this result proves the practicability of bottleneck analysis and the scalability of the improved algorithm. In fact, we used `CONV` as the class project of CS133 in winter 2019 for undergraduate and graduate students of UCLA's Computer Science Department. The students were asked to improve the performance of `CONV` with the Merlin compiler [CHP16a] within the period of one week. Our DSE result ran second place among 84 student submissions.

On the other hand, `KMP` can only achieve 52% compared to the manual design. We find that the reason lies in the accuracy of Merlin report analysis. When the kernels contain many unbounded loops or while-loops, the HLS report may not reflect

Table 3.9: Overall Comparison to CPU and Manual Designs

Benchmark	Design Space	Speedup over v3	Ratio to Manual (%)	Speedup over CPU
AES	3.11E+09	1.00	100	3774.69
NW	1.51E+09	0.98	97.67	3387.46
KMP	5.76E+03	1.00	52.24	5.04
GEMM	1.26E+09	1.00	100	16.25
SPMV	5.76E+03	1.31	100	1.73
STENCIL-2D	9.70E+09	1.17	94.00	0.39
STENCIL-3D	1.94E+06	1.00	100	2.65
BACKPROP	1.15E+04	1.00	100	7.71
KMEANS	2.49E+05	8.46	99.18	34.82
KNN	1.90E+04	1.00	99.84	9.48
PATHFINDER	5.18E+03	0.89	88.62	0.16
CONV	1.50E+28	291.49	93.96	55.06
Geometric Mean	1.26E+08	1.96	93.78	13.69

the accurate computation cycles. This affects the bottleneck type analysis of the Merlin report. In the case of `KMP`, the Merlin report shows that the bottleneck type of design point is computation, but it is actually memory transfer. Once the Merlin report provides the wrong information, our search algorithm will identify unimportant design parameters to focus on; therefore the performance bottleneck cannot be resolved. Future work will study the Merlin report analysis and identify the situations that may cause inaccurate analysis so that we can try to avoid them.

Table 3.10: FPGA Resource Utilization

Benchmark	BRAM (%)	LUT (%)	FF (%)	DSP (%)
AES	31	11	2	0
NW	52	42	12	0
KMP	37	21	5	0
GEMM	52	33	6	30
SPMV	74	30	4	20
STENCIL-2D	5	5	3	8
STENCIL-3D	9	7	5	7
BACKPROP	72	37	8	10
KMEANS	50	30	10	26
KNN	23	23	1	18
PATHFINDER	7	2	1	1
CONV	69	59	42	60

We then analyze the DSE process of all four versions in Figure 3.14. The most important message behind Figure 3.14 is that v4 has the overall fastest performance growth due to the identification of bottleneck parameters and bottleneck type. We note that this is important to DSE for HLS. The reason is that although it is common for hardware designers to spend months exploring the best architecture configuration, they will spend most of their time analyzing the workload and narrowing it down to a suitable architecture before performing design space exploration. On the other hand, the HLS designers usually have an intuitive C-based implementation to start with, so they often perform HLS without any optimization to analyze the performance

bottleneck. Based on the performance bottleneck, which may be caused by data dependency, program structure, memory bandwidth, or underutilized resource, the designer may need to reconstruct the program to complement uncovered design space, such as loop splitting or interchange. During this process, it is better for designers to obtain the best performance of the current implementation in a short time so that they could further improve it accordingly. As a result, a reasonable exploration time should be within hours. According to Figure 3.14, our v4 framework can rapidly achieve high performance, and we believe this is helpful for HLS designers to refine their designs.

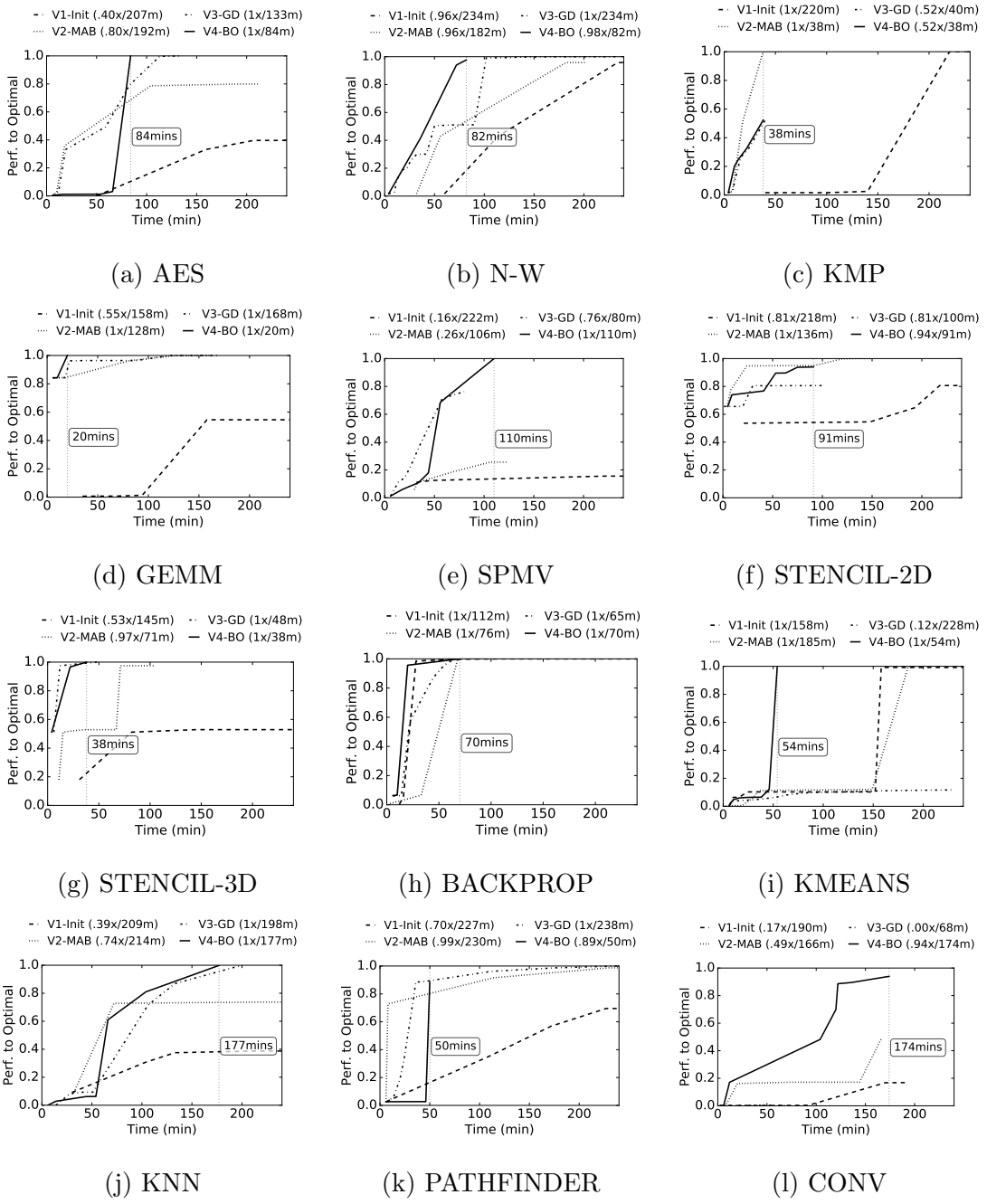


Figure 3.14: DSE with Bottleneck Optimization Algorithm. The legend notes the best speedup over the manual design and the time to achieve it.

3.5.5 Experimental Results on a Different Platform

We finally explore the design space on the Intel FPGA using the optimized version of the DSE framework to demonstrate its adaptability in another dimension. The platform as well as the device we use in this experiment are the Intel AOCL 19.1 [INT] and an Intel Arria 10 FPGA [ARR]. The experimental results of performance comparison to manual design and CPU baseline are shown in Table 3.11, while the resource utilization is reported in Table 3.12. For each benchmark case, we manually implement an optimal version on the Intel FPGA using Merlin compiler pragmas to evaluate the result quality achieved by the DSE framework. Note that the Merlin manual designs of some cases, such as `KMP` and `SPMV`, fail to achieve speedup over the CPU baseline because the performance bottleneck of those kernels are highly bounded by the external memory bandwidth and would require a specialized architecture. Again, our analysis will focus on the optimality achievement by the framework. We can see from Table 3.11 that our DSE framework successfully identifies the best design point which matches the manual design performance for all cases with the same design space as on the Xilinx platform and achieves a geometric mean 15.46× speedup over the CPU baseline. This result on the Intel platform illustrates that the proposed framework is capable of finding the best point on a different device and design flow within the same design space.

In addition, Table 3.14 and Table 3.13 list the number of used pragmas in the best design point for the benchmark case on two platforms. Note that *"CG"* in the table stands for coarse-grained while *"FG"* stands for fine-grained. It is obvious that the best design points are quite different on each platform. In particular, a coarse-grained pipeline is one of the most important optimizations for most cases on Xilinx, so almost

Table 3.11: Overall Comparison to CPU and Manual Designs on Intel FPGA

Benchmark	Design Space	Explored Points	Ratio to Manual (%)	Speedup over CPU
AES	3.11E+09	81	100	1875.45
NW	1.51E+09	149	100	3213.97
KMP	5.76E+03	129	100	0.29
GEMM	1.26E+09	95	100	9.97
SPMV	5.76E+03	242	100	0.66
STENCIL-2D	9.70E+09	405	100	1.38
STENCIL-3D	1.94E+06	193	100	22.44
BACKPROP	1.15E+04	185	100	8.07
KMEANS	2.49E+05	251	100	35.50
KNN	1.90E+04	263	100	9.54
PATHFINDER	5.18E+03	73	100	12.37
Geometric Mean	1.26E+08	165	100.0	15.46

all cases are applied, since Vivado HLS optimizes external memory access mostly relying on user code structure. Intel AOCL, however, attempts to generate dataflow-like architecture, which naturally pipelines all module executions using FIFOs, for user applications. As a result, Merlin coarse-grained pipeline transformation with double buffering is less effective on the Intel platform and should be avoided in some cases to save resources.

Similarity, coarse-grained parallelism is also widely applied on the Xilinx platform but not on Intel. This is also mainly because of the dataflow architecture Intel tool

Table 3.12: FPGA Resource Utilization

Benchmark	BRAM (%)	LUT (%)	FF (%)	DSP (%)
AES	14	2	4	0
NW	75	48	34	0
KMP	21	12	11	0
GEMM	21	79	28	34
SPMV	50	22	11	4
STENCIL-2D	10	3	2	1
STENCIL-3D	18	3	2	3
BACKPROP	54	44	16	27
KMEANS	7	5	4	21
KNN	31	21	11	72
PATHFINDER	34	9	5	0

adopted. The Merlin coarse-grained parallelism transformation not only generates multiple processing elements (PEs) but a number of buffers to deal with data transfer from external memory to each PE. The Xilinx Vivado HLS is able to use the generated buffers to infer memory burst that transfers a whole chunk of data to on-chip BRAM at once and optimizes the bandwidth. On the other hand, although the Intel AOCL already uses FIFO channels to transfer data between modules in the pipeline manner, additional buffers are still required to achieve full pipelining if the PEs access external memory out of order. In this case, the Merlin transformation reorders the data and puts it to the generated buffers, but the extra overhead introduced by data reordering moderates the benefit of massive parallel execution and may not be adopted in the

Table 3.13: Best Design Point on Xilinx FPGA

	Memory coalesce	FG parallel	CG parallel	CG pipeline	Data tiling
AES	1	6	1	1	0
NW	0	5	1	1	0
KMP	3	2	1	1	0
GEMM	3	2	1	2	0
SPMV	4	1	1	1	1
STENCIL-2D	2	4	0	1	0
STENCIL-3D	2	2	1	1	0
BACKPROP	3	1	1	0	1
KMEANS	1	1	0	0	1
KNN	3	1	1	1	0
PATHFINDER	2	3	0	1	0

best design point.

In summary, the best design points identified by the DSE framework are capable of reflecting the philosophy of different underlying vendor tool implementations, especially from the same Merlin compiler formed design space. The Intel AOCL puts more efforts on computation optimization and channel-based module communication. This means users do not have to worry too much about high-level architecture but can focus on computation optimization inside modules; but it also implies that the developers or optimization tools such as the Merlin compiler will find it hard to remedy poor performance if the AOCL fails to optimize certain designs. In con-

Table 3.14: Best Design Point on Intel FPGA

	Memory coalesce	FG parallel	CG parallel	CG pipeline	Data tiling
AES	1	6	1	1	0
NW	4	2	1	0	0
KMP	2	0	0	0	0
GEMM	0	2	1	0	0
SPMV	0	1	0	0	0
STENCIL-2D	0	0	0	0	0
STENCIL-3D	0	2	0	0	0
BACKPROP	4	1	0	1	1
KMEANS	1	1	0	0	0
KNN	2	1	0	0	1
PATHFINDER	1	3	0	0	0

trast, optimization of the Xilinx Vivado HLS is mostly triggered by user pragmas and specific code patterns. In other words, users have to spend more efforts on carefully implementing an entire architecture to achieve high performance on the Xilinx platform, but it also provides a relatively clear direction to iteratively improve the performance.

3.6 Conclusion

In this chapter, we design and implement an efficient design space exploration framework with the Merlin compiler for HLS on FPGAs. We start from OpenTuner [AKV14], an open source auto-tuning framework with multi-armed bandit approach to explore Merlin pragma formed design space (v1). Although the execution flow is working well, it only finds the high quality design point for a half of our benchmark designs and only achieves on geometric mean 44.6% performance to the manual designs. To improve the efficiency of the DSE process, we propose several optimization strategies. We partition the design space to enable parallel searching; we generate effective seeds as promising start points; we leverage the concept of Shannon entropy to terminate the DSE process earlier. With all those strategies applied, the DSE framework v2 reduces the execution time to only 2.68 hours on average while achieving on geometric mean 79.87% performance to the manual designs.

In addition, to easily reason the DSE process, we attempt to use a deterministic approach as a search algorithm. Specifically, we use gradient descent with finite difference met hoed to explore the design space. Unfortunately, the gradient approach encounters a serious local optimal issue, so we dive into the reasons behind the local optimal and propose multiple optimizations. We first propose a graph-based algorithm to prune the design space by 24.65×. Then we leverage design space partition and adaptive line search to alleviate the local optimal issue. Finally, we use multiscale V-cycle, which is inspired from VLSI physical design [CS13], to temporary group some design parameters and explore them together. Although the DSE framework v3 only achieves 47.16% performance on geometric mean even we have integrated all proposed optimizations, the gradient process provides a clear direction

for improvement. Accordingly, in the DSE framework v4, we propose a comprehensive design space representation to preserve the dimension and shape so that the exploration process could be smooth. We also leverage Merlin report analysis to identify the design bottleneck, and let the search algorithm focus on a small set of design parameters to improve the efficiency. Consequently, the DSE framework v4 achieves 93.78% performance to manual designs with an even shorter exploration time. Moreover, we also perform DSE on Intel platform to further demonstrate the generalization. The results show that with the same design space, the DSE framework v4 can identify the best design points, which are different from the best one on Xilinx platform, that achieve manual design performance on Intel platform.

Since the DSE framework we developed in this chapter adopts general Python compatible list comprehensions to specify design space, it is suitable to be a performance optimizer of higher level domain specific languages (DSLs) to FPGAs. The frontend compiler of a high-level DSL is able to reflect the application characteristics to the design space and further facilitate the DSE process. In the next chapter, we use two high level DSLs, Spark in Scala and HeteroCL in Python, to illustrate this idea.

CHAPTER 4

Raising Design Abstraction for Domain Specific Frameworks

In this chapter, we provide supports to high-level domain specific languages to expend the usability of the automated design space exploration framework proposed in Chapter 3. We first introduce S2FA, a Spark-to-FPGA compilation framework in Section 4.1. As Apache Spark [ZCD12], S2FA leverages MapReduce programming model to manipulate resilient distributed datasets (RDDs). Users are allowed to use almost arbitrary functional programming and object-orient constructs to describe their applications. In this work, we focus on reducing the semantic gap between Scala and Merlin C, and leverage the parallel patterns of MapReduce programming model to help the DSE framework reduce the design space.

In addition to MapReduce parallel patterns, we also provide the support to HeteroCL [LCH19], a programming infrastructure with a Python-based domain-specific language (DSL) for FPGAs. Like Halide [RBA13] and Tensor Virtual Machine (TVM) [CMJ18], HeteroCL programming model fully decouples functional description and scheduling so that the application and platform-dependent scheduling function can be developed separately. As a result, we build a framework that compiles HeteroCL DSL to Merlin C and leverage the DSE framework to demonstrate that the

DSE for HLS could be more effective and promising with a clean scheduling interface and user interactions.

4.1 S2FA: A Spark-to-FPGA Accelerator Framework

4.1.1 Overview

Cloud computing has recently become a popular solution to the growth of dataset sizes for data analytics. Many widely-used open-source big data analytics frameworks, such as Apache Hadoop [HAD] and Spark [ZCF10], have been made to scale to large numbers of datacenter machines. However, as energy efficiency becomes a larger problem for datacenter operators, the adoption of energy-efficient devices such as GPUs and FPGAs becomes attractive. In particular, to use FPGAs in datacenter, we need to provide support for mapping applications written for big data analytics frameworks down to FPGAs easily and efficiently. While several existing works have built a path for GPUs [SSE15, HCC10, GBS13, GS16], the research into FPGAs is limited. The primary challenge that must be addressed to adopt FPGAs in datacenters is the programmability. The most broadly used open-source frameworks for datacenters, Hadoop and Spark, are implemented in either Java or Scala. Unfortunately, these high-level programming languages are not supported by FPGA design flows directly. When offloading a kernel written in Java or Scala, a significant amount of developer effort is required to manually design and implement an FPGA accelerator with the same functionality. Moreover, the developer also must deal with system integration to access the designed accelerators from the bit data application.

To facilitate the ease of use of FPGA for big-data computations, Blaze [HWY16]

made efforts to integrate FPGAs into Spark and allow programmers to offload computational kernels to FPGAs easily. However, Blaze leaves to the programmers the responsibility for developing FPGA accelerators for the offloaded kernels. Therefore, a significant amount of human efforts is still required for users to manually produce accelerator designs. Worse still, Blaze only supports primitive types as accelerator interfaces, indicating that additional programming effort may be needed to serialize/deserialize composite data types such as structures and classes in Java/Scala.

Table 4.1: Development Time of FPGA Accelerators from Scala and Speedup Comparison with One Spark Executor

Kernel	Time-to-FPGA	Time-to-Speedup
PageRank	<0.01× (3 hrs)	0.2× (2 days)
KMeans	0.18× (5 hrs)	51.6× (4 days)
Logistic Regression	<0.01× (6 hrs)	40.8× (8 days)
K-Nearest Neighbor	<0.01× (4 hrs)	26.0× (4 days)
Support Vector Machine	<0.01× (5 hrs)	9.1× (7 days)
Least Linear Square	<0.01× (5 hrs)	10.8× (7 days)
Smith-Waterman	<0.01× (12 hrs)	204.7× (6 days)
AES encryption	<0.01× (8 hrs)	1278.3× (3 days)

As further motivation, Table 4.1 illustrates the effort required to manually rewrite Scala functions as FPGA kernels. The effort is quantified as human hours by a HLS expert. Since we need to not only translate the syntax from Scala to an FPGA kernel friendly language, but also bridge the semantic gap between an object-oriented

language and a C-based language used by HLS tools, it usually requires a few hours[1] to generate a working accelerator kernel. However, additional days are required to achieve an acceptable amount of performance improvement because we need to analyze the nature of the design and apply suitable FPGA-specific optimizations such as pipelining, parallelism, memory bursting, double buffering, etc. As a result, automated code generation with FPGA-aware optimization from JVM-based programming languages play an important role in supporting big data applications on FPGAs.

In this chapter, we present S2FA, a Spark-to-FPGA accelerator framework. S2FA is an automated compilation framework which automatically offloads user-written Spark applications to FPGAs by generating optimized accelerator kernels and data (de)serialization methods. The user is able to use Scala to implement Spark transformations without considering the underlying hardware architecture. S2FA's programmability is improved over past work by supporting several commonly used object-oriented programming constructs. It also leverages the design space exploration framework presented in Chapter 3 to optimize the accelerator performance. Since the user-written computational kernels in Scala for Spark and Blaze contain the semantic information of RDD transformation operators such as `map`. This information is capable of being used to prune the design space and facilitate the design space exploration process. To summarize, this work makes the following contributions:

- An automated framework which performs offline compilation of the user-given transforms in a Spark application to an FPGA accelerator, while generating

[1]The time reported here includes human hours as well as the execution time of FPGA design flow tools for kernel generation.

corresponding functions for software-hardware system integration. Only kernels that use a subset of Scala/JVM functionality (e.g. no dynamic class loading) are eligible for offline compilation.

- Support of several high-level Scala programming and data constructs through Scala-to-C code transformations and class object serialization.
- An integration to the design space exploration framework with Spark-specific pruning strategies to effectively organize a given set of optimization strategies to produce high-performance designs.
- Detailed evaluation of S2FA on a variety of computational kernels on the Amazon EC2 F1 instance, and insights to the impact of DSE optimization strategies on the quality of results.

The evaluation results demonstrate that S2FA is able to generate FPGA accelerator designs from Spark applications applications with correct functionality. Table 4.2 and Table 4.3 summarizes the characteristics of our framework when compared with other automated accelerator-offload big data frameworks. To our best knowledge, S2FA is the first big data framework for FPGAs that supports object-oriented constructs.

4.1.2 Preliminary: Blaze Runtime System

Blaze [HWY16] is a Spark-based runtime system which provides programming and runtime support for easy and efficient deployments of FPGA accelerators in datacenters. Blaze abstracts FPGA accelerators as a service (FaaS) by decoupling the FPGA accelerator and Spark application developments. FPGA accelerators can be

Table 4.2: Big Data Frameworks with Code Generation for Heterogeneous Platforms

Framework	**Base Language**	**Accelerator Platform**	**Programming Model**	**Runtime**	**Kernel Optimization**
SparkCL [SCN15]	Java	FPGA	Override methods	Spark (SparkCL)	Manual
Melia [WZH16]	C	FPGA	Override methods	Melia	Performance cost model
HeteroDoop [SSE15]	C	GPU	Directives	Hadoop (HeteroDoop)	User specified directives
MapCG [HCC10]	C	CPU/GPU	Override methods	MapCG	N/A
HadoopCL [GBS13]	Java	GPU	Override methods	Hadoop (HadoopCL)	N/A
SWAT [GS16]	Scala	GPU	Wrapper APIs	Spark (SWAT)	N/A
S2FA	Scala	FPGA	Annotations	Spark (Blaze [HWY16])	DSE

Table 4.3: Programmability Summary of the Heterogeneous Frameworks

Framework	Objects	Type Parameter	Class Inheritance	Method Overriding
SparkCL [SCN15]	No	No	No	No
Melia [WZH16]	No	No	No	No
HeteroDoop [SSE15]	No	No	No	No
MapCG [HCC10]	No	No	No	No
HadoopCL [GBS13]	Yes	No	No	No
SWAT [GS16]	Yes	Lmited	No	No
S2FA	Yes	Limited	Yes	Yes

Code 4.1: Blaze Application Code Snippet

```
1 val vecs: RDD[Vector] = // read input
2 val blaze_vecs = blaze.wrap(vecs)
3 val maxIndices = blaze_vecs.map(new FindMaxAcc)
```

customized and registered to the Blaze accelerator manager. Spark application developers can access FPGA accelerators using provided APIs that hide the details of JVM-to-FPGA data communication.

The Blaze programming model requires only a few code changes from Spark applications to support FPGA accelerators. Code 4.1 illustrates the Blaze programming model using the previous maximum-index example. To accelerate an RDD transformation, we use the Blaze API to wrap that RDD so that Blaze can extract that RDDs metadata. To specify the accelerator design to be used for this RDD transformation, we write another Scala class with the corresponding accelerator design

Code 4.2: Blaze Accelerable Class

```
class FindMaxAcc()
    extends Accelerator[Vector, Int] {
  val id: String = "FindMax"
  def call(in: Vector) = in.argmax
}
```

ID in Code 4.2, and put the original map function into it. While Blaze streamlines accessing FPGAs from Spark applications, Blaze still requires that users manually implement their own FPGA kernels and data (de)serialization methods for specialized input/output data processing.

In this work, we develop a framework for automatically generating 1) an FPGA accelerator design, and 2) data (de)serialization methods for the Blaze runtime system in datacenters. We choose Blaze because it is the only Spark-based runtime system that efficiently integrates with FPGAs. However, we note that the framework we present is able to compile any Java/Scala method that satisfies the constraints listed in the next section to an FPGA kernel, so we can easily integrate this framework with other JVM-based runtime systems in the future (e.g. Java parallel streams, Hadoop MapReduce).

4.1.3 S2FA Framework

In this section, we start with an example to motivate our work. Then, we introduce each component of the S2FA framework and explain how they address challenges demonstrated by the example. Finally, we summarize the Java/Scala constructs currently supported by the S2FA framework.

4.1.3.1 Motivating Example

We continue to use the maximum-index example from the previous section here. To accelerate the process of finding the maximum index in a vector using Blaze, we must first implement an FPGA kernel in HLS C with the same functionality as the RDD transformation (`def call(in: Vector) = in.argmax`). However, the following challenges make the implementation challenging.

Challenge 1: Static compilation flow. Implementing FPGA designs offline implies that we lack dynamic, runtime information, such as the length of vector for each input item. Thus, it is difficult to generate a dataset-optimized and semantically-equivalent accelerator design.

Challenge 2: The semantic gap. Since OpenCL is a C-based programming language, it does not support any object-oriented language constructs. In this example, the class `Vector` and virtual method call `argmax` are not directly supportable in FPGA kernels. Instead, we must use an FPGA-compatible data representation. In this example, we use a double array to store values in a vector and an integer to store the length of the array.

Challenge 3: Poor performance of generated FPGA designs. A high-performance FPGA accelerator design requires the designer to understand the underlying FPGA architecture, which can be impractical for domain experts or application developers. For this example, a naive implementation using HLS C results in a more than 10× slow down relative to single-threaded JVM execution. Only with hours of developer time spent on architecture-aware optimizations can we finally achieve an acceptable speedup.

Challenge 4: System integration. After implementing an efficient FPGA kernel,

we also need bridge code to tie that kernel into the host JVM application. In our case, that requires the implementation of a Scala method for processing a `Vector` object into a double array. Requiring the user to do this manually would also impose heavy programmer burdens.

As can be seen, these challenges are common and inevitable for a programmer who wants to accelerate a Spark program using FPGAs. As a result, it is worthwhile to develop an automated framework to address them.

4.1.3.2 Programming Model

The S2FA framework leverages an annotation-based programming model to preserve the programmability of Spark and hide all details of the hardware. Annotations in the Scala source code start with `@` followed by the constructor of an annotation class.

Code 4.3: S2FA Programming Model

```
1  @S2FA_Kernel(Vector.values:1024)
2  def call(in: Vector) = in.argmax
```

Annotations can be applied to any Scala declaration. Code 4.3 presents a code snippet of the previous maximum-index example written in Scala with S2FA annotations. In order to accelerate the `call` method using FPGAs, we simply put the annotation `@S2FA_Kernel` on the top of the method declaration so that the framework can identify and compile this method to an FPGA kernel. Because current FPGA programming models do not support dynamic memory allocation and all arrays in Scala are dynamically allocated, the user must also specify the maximal lengths for any arrays with non-constant lengths at compilation time. The S2FA framework will compile a dynamic array allocation with user-provided length to a static array

declaration.

4.1.3.3 S2FA Framework Overview

To prevent multi-hour FPGA synthesis times from interfering with runtime application performance, the Blaze runtime system [HWY16] decouples FPGA accelerator design and software development. To eliminate additional FPGA design burdens, the proposed S2FA framework compiles FPGA kernels offline, and cleanly integrates with the Blaze runtime system. Figure 4.1 presents the design of the S2FA framework. We introduce each component as follows along with the maximum-index example in Code 4.4 to illustrate the execution flow. In Code 4.4 and Code 4.5, we show the application with an S2FA annotation and the corresponding declaration of ML-lib's [MLL] `Vector` classes. The annotation indicates that the user would like to offload the method `call` to the FPGA.

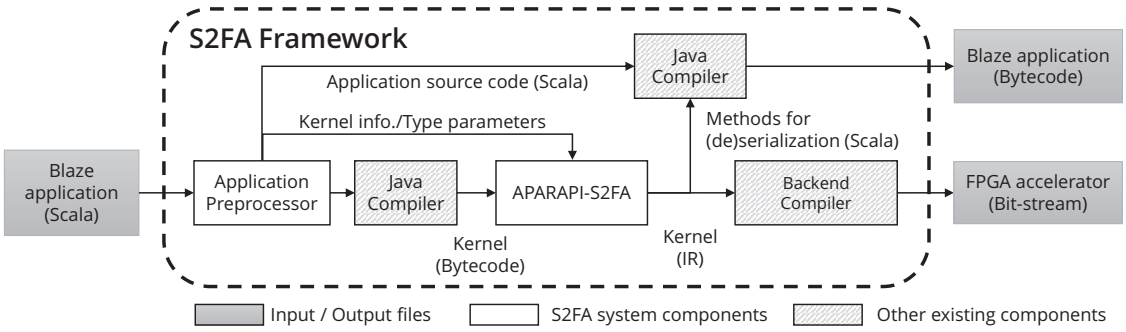


Figure 4.1: Framework Overview

Our flow is designed to address the challenges we presented above.

Stage 1: Preprocessor. Unlike most existing work in the area of accelerated data analytics using GPUs [SCN15, GBS13, GS16] which can leverage runtime informa-

Code 4.4: A Running Example: User-written Scala Method Code

```
1  @S2FA_Kernel(Vector.values:1024)
2  def call(in: Vector) = in.argmax
```

tion, S2FA preprocessor takes only Scala source code and user annotations as input. By analyzing the abstract syntax tree (AST) of the Scala source code, the S2FA preprocessor is able to infer type parameters that cannot be inferred from bytecode alone due to type erasure. Moreover, user-provided annotations provide other useful hints, such as the maximum length of dynamically allocated arrays. As a result, the S2FA preprocessor can recover the necessary information that would normally only be available at runtime. As shown in Code 4.6, the preprocessor combines the user-written `call` method and MLlib bytecode to generate a C kernel, which includes all necessary class and function declarations. Note that we ignore SparseVector in the C kernel in this example for concise.

Stage 2: APARAPI-S2FA. After parsing the Scala source code, S2FA compiles the target method to Java bytecode and feeds it to APARAPI-S2FA. APARAPI-S2FA is a tool developed based on the open-source AMD APARAPI [APA] framework which performs bytecode-to-OpenCL translation. In this work, we heavily modify the APARAPI code generator to generate efficient FPGA kernels from Scala.

Note that FPGA kernel languages today do not support object-oriented constructs. we therefore must convert the Java bytecode into an FPGA-compatible, syntactically-C language. To avoid computationally heavy code analysis at runtime, the original APARAPI code generator limits users to primitive types, and does not support any automatic code optimization. As a result, we use a transformation approach inspired by [KP10] to convert Scala classes and objects to FPGA-compatible

Code 4.5: A Running Example: MLlib Vector Class Code Snippet

```
trait Vector {
  def size: Int
  def argmax: Int
}
class DenseVector(val values:
  Array[Double]) extends Vector {
  def size: Int = values.length
  def argmax: Int = {
    if (size == 0) -1
    var maxIdx = 0
    var maxValue = values(0)
    var i = 1
    while (i < size) {
      if (values(i) > maxValue) {
        maxIdx = i
        maxValue = values(i)
      }
      i += 1
    }
    maxIdx
  }
}
class SparseVector(
  // Skip due to page limit
}
```

Code 4.6: A Running Example: Generated C Code Snippet

```
1 int Vector_size(long *this) { ... }
2 int Vector_argmax(long *this) { ... }
3 int DenseVector_size(long *this) {
4   return (int) this[1];
5 }
6 int DenseVector_argmax(long *this) {
7   if (DenseVector_size(this) == 0)
8       return -1;
9   int maxIdx = 0;
10  double maxVal = (double) this[2+0];
11  int i = 1;
12  while (i < DenseVector_size(this)) {
13    if ((double) this[2+i] > maxVal) {
14      maxIdx = i;
15      maxVal = (double) this[2+i];
16    }
17    i += 1;
18  }
19  return maxIdx;
20 }
21
22 int call(long *in) {
23  return Vector_argmax(in);
24 }
25 void kernel(int N, long *in, int *out) {
26  for (int i = 0; i < N; i++)
27    out[i] = call(&in[i * VECTOR_SIZE]);
28 }
```

C code by converting objects into an array of fields. Similarly, field accesses are transformed to array access expressions. As can be seen in Code 4.6, the expression `values.length` in the function `DenseVector_size` has been transformed to `this[1]`. The mapping of object fields to array indices is determined by the S2FA class transformer. Additionally, class member functions are transformed to explicitly accept `this` as their first argument. To deal with function overriding, virtual functions such as `Vector.argmax` are transformed to dispatch functions. The details of dispatch functions are described in the next section.

Stage 3: Design space exploration. To guarantee high performance of generated designs, we leverage the DSE framework presented in Chapter 3. On the other hand, instead of exploring the full design space created by the DSE framework, we apply a parallel pattern specific pruning strategy to facilitate the searching efficiency. For example, the parallel pattern `map` indicates that we must have a single outermost loop in a kernel and it can be fully executed in parallel. As a result, we are able to limit scheduling options of the outermost loop and reduce the design space size by orders of magnitude.

Stage 4: Data processing method generator. To bridge the gap between the Spark-based Blaze runtime and the automatically generated and synthesized FPGA kernel, the S2FA class transformer can auto-generate Scala methods based on static analysis which the Blaze runtime system uses to serialize and deserialize input and output data. These methods use Java reflection to access object fields and convert them to a format that matches the accelerator interface.

With each of these challenges solved, the final step is the broadcast of a synthesized bit-stream and accelerator information to each worker node in a datacenter.

4.1.3.4 Supported Object-Oriented Constructs

The S2FA framework generates FPGA kernel code in C from a user-written Scala lambda passed to the Spark RDD `map` transformation. Similar RDD transformations such as `flatMap` and `mapPartition` are also possible through slight modifications of the `map` implementation. Other transformations such as `sample` and `filter` are not suitable for FPGA acceleration as their kernels are not computationally intensive and cannot achieve sufficient computational speedups to hide CPU-FPGA data transfer overhead.

The S2FA compatible user-written lambda for RDD `map` transformation should only use the following supported object-oriented features:

Classes: Simple classes, interface classes, and single inheritance classes that are provided by either a user or a library are supported. For objects created with type parameters such as `Tuple2[Int, Float]`, the Scala source code must be accessible for S2FA to extract the type information. In other words, we currently do not support objects with type parameters declared in third-party libraries.

Methods: S2FA supports Scala methods either provided by a user or a third-party library. However, S2FA again needs to extract type information from Scala source code, which is usually unavailable at compile-time for libraries compiled to bytecode. Thus, the library methods used in the kernel may not be parameterized by types. In addition, methods that accept lambda-typed arguments (e.g. `foreach`) are not supported.

Dynamic memory allocation: S2FA supports the JVM's `new` operation with either a constant memory size, or a dynamic memory size when a method annotation provides a maximal number of elements that will be allocated by any dynamic

memory allocation in that method. All `new` operations will be compiled to static variable declarations in C, and no dynamic memory allocation will be performed on the FPGA.

Exceptions: Neither throwing nor catching exceptions are supported currently. Past work has explored supporting exceptions on GPUs [HGZ13]. Future work would integrate similar techniques into the S2FA framework.

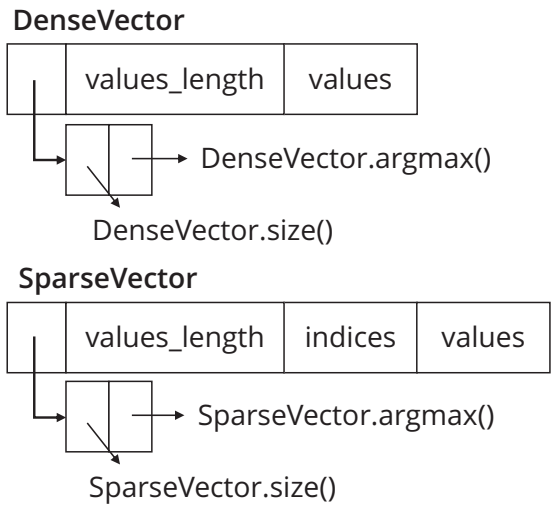
We mention that above restrictions do not affect design scopes, e.g. users are still able to leverage S2FA to accelerate a kernel with any functionality. On the other hand, the restriction only affects the way of implementation.

4.1.4 Class/Object Transformation

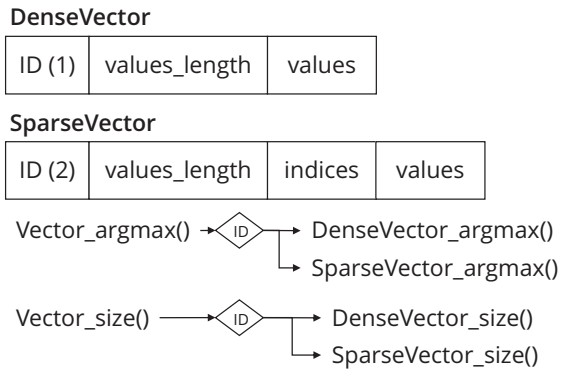
Objects are instances of classes and are widely used in object-oriented programming. It is worthwhile to support objects for the improvements in programmability they offer. Typically, a Java or Scala object contains member variables (fields) and functions (methods). In addition, an object may also contain fields and methods that it inherits from base classes.

The traditional approach compiles each class into a `C struct` which is composed of 1) a pointer to a virtual method table, 2) the variables of the base classes, and 3) the variables declared in the class itself. Figure 4.2a illustrates this traditional approach. Polymorphism in Java/Scala is supported in C by calling the proper function pointer in the virtual method table during execution. As a result, code must be generated to place function pointers in this lookup table based on the actual data types at runtime.

Unfortunately, this approach is not applicable to FPGA kernels, since 1) C structs



(a) Traditional Data Layout



(b) S2FA Data Layout

Figure 4.2: Data Layouts for Class Serialization

are not permitted as arguments to FPGA kernels, and 2) function pointers are not supported. To address these issues, we use unified arrays instead of structs to store the data of each class object. An example of this is shown in Figure 4.2b. The first element of each unified array stores the class ID in order to support dynamic casting. It is followed by the values stored in each class field. All class field accesses

are compiled to element accesses in the unified array with type casting. In addition, we generate dispatch functions to deal with polymorphism. Our complete approach is illustrated in Algorithm 6.

Code 4.7: Dispatch Function of Vector.argmax

```
1 int Vector_argmax(long *this) {
2   switch ((int) this[0]) {
3     case 1: // DenseVector class ID
4       return DenseVector_argmax(this);
5     case 2: // SparseVector class ID
6       return SparseVector_argmax(this);
7   }
8 }
```

In Algorithm 6, we first parse the class declaration to build a class model. A class model contains information on the class hierarchy and fields. Subsequently, class methods are compiled to standalone functions with appropriately mangled function names. The first argument is always the unified array representation of the `this` object. To dispatch a virtual method call based on the type of the target, we generate a dispatch function. A dispatch function is composed of a switch statement which calls the appropriate function based on the class ID of the target object (stored in `this[0]`). For instance, the dispatch function of `Vector.argmax` is presented in Code 4.7. Finally, we generate customized object (de)serialization methods in Scala for Blaze system integration based on the constructed class model.

Algorithm 6 Class Serialization

Require: A class declaration C.

Ensure: 1) Semantic equivalent C code with transformed member functions, and 2) Class (de)serialization code in Scala.

1: $M \leftarrow CreateClassModel(C)$

```
2: for all F ∈ M.GetMemberFunctions() do
3:    F' ← Clone(F)
4:    F'.SetName(M.GetName() + "_" + F.GetName())
5:    F'.InsertArgument(0, M.GetTypeName(), "this")
6:    for all Arg ∈ F'.GetArgs do
7:       if Arg ∈ Object then
8:          Arg.SetType(UnifiedType)
9:       end if
10:   end for
11:   for all A ∈ F'.GetFieldAccesses() do
12:      FieldIdx ← M.GetFieldIdx(A.GetField())
13:      NewAccessExp ← new ArrayAccessExp("this", FieldIdx)
14:      NewAccessExpWithCast ← newCastExp(A.GetType, NewAccessExp)
15:      F'.Replace(A, NewAccessExpWithCast)
16:   end for
17:   Write F' to output code
18:   if F.hasDerivedClassImplementation() then
19:      Write dispather function to output code
20:   end if
21: end for
22: Write (de)serialization Scala code from M
```

4.1.5 Experimental Evaluation

While the application-level speedup and system-level overhead are transparent to Blaze runtime system [HWY16], this evaluation focuses on the performance eval-

uation of S2-FA generated accelerators. Our evaluation of S2FA is performed on Amazon EC2 F1 instance [AWS]. The instance type is `f1.2xlarge`, which includes an 8-core CPU with 122GB of main memory and one Xilinx Virtex UltraScale+^{TM} VU9P FPGA with three separated dies. In addition, we select a set of common Spark applications to evaluate S2FA. We also select two string processing applications in our evaluation since they are classic applications for FPGA acceleration. All applications are built with the software environment that consists of JDK 1.7.0_79, Scala 2.11.4 and Spark 1.5.1.

Table 4.4: Scala Application I/O Types for the Experiments

Application	Input Type	Output Type
PageRank (PR)	(Int, Int)	(Int, Float)
KMeans (KMeans)	(Vector, Double)	(Vector, Array[Int])
Logistic Regression (LR)	Vector	Vector
K-Nearest Neighbor (KNN)	Vector	Vector
Support Vector Machine (SVM)	Vector	Vector
Least Linear Square (LLS)	Vector	Vector
Smith-Waterman (S-W)	(String, String)	(String, String)
AES (AES)	String	String

We use eight applications to evaluate S2FA. The input and output Scala data types of each application are shown in Table 4.4. In addition, we adopt real-world data sets in this evaluation for each application. For `PageRank`, we use the Hyper-Link Graph dataset from Web Data Commons [WDC] with ~120M nodes for this application. For `KMeans`, we use the Record Linkage Comparison dataset from the

UCI Machine Learning Repository [MLR07] with ∼20K data points and 12 features to evaluate this application. We classify the data set to 3 clusters in our experiments. For `KNN`, `LR`, `SVM` and `LLC`, we use a variant of the MNIST data set of handwritten digits [MNI] that contains 8 million 28×28 serialized gray images. As a result, each image has 784 features (pixels) and 10 labels, so the problem we target is a multi-class classification problem. For `S-W`, we use the S-W algorithm to align 256K reads from a real individual human genome sample, HCC1954. Finally, for `AES`, we use a 256-bit key size and 128-bit block size. The dataset we use for evaluating the AES cipher is a random 250MB text file.

Based on the best configurations from the DSE framework in Chapter 3, Table 4.5 lists the resource utilization and working frequency of each generated design. Since the performance of `AES` and `PR` are bounded by external memory bandwidth, they do not fully utilize hardware resources. On the other hand, other cases fully utilize at least one kind of resource, meaning that those three designs are computationally bounded and their performance can be potentially improved if a larger FPGA is provided. Note that we set the maximum resource utilization to 75% since the rest of them were used by the vendor-provided control logic. In addition, since we perform place and route using the default setting of Xilinx SDx [SDX], the frequency fails to achieve the target (250MHz) for satisfying timing constraints for some cases. The impact of design parameters on frequency during the DSE process is also a worth topic to be investigated.

Figure 4.3 shows the speedup of manual and S2FA-generated FPGA designs with and without the help from the proposed DSE framework over the original Spark transformation methods running on a JVM. The x-axis lists all designs while the y-axis illustrates the speedup in logarithm scale. We use a single-threaded Spark

Table 4.5: Resource Utilization (%) and Clock Frequency (MHz)

Kernel	Type	BRAM	DSP	FF	LUT	Freq.
PageRank	graph proc.	25	2	16	18	250
K-Means	classification	73	6	10	14	230
K-Nearest Neighbor	classification	75	6	50	50	240
Logistic Regression	regression	74	3	49	74	220
Support Vector Machine	regression	74	4	48	72	250
Least Linear Square	regression	74	3	45	21	230
AES	string proc.	36	0	3	6	250
Smith-Waterman	string proc.	33	30	54	75	100

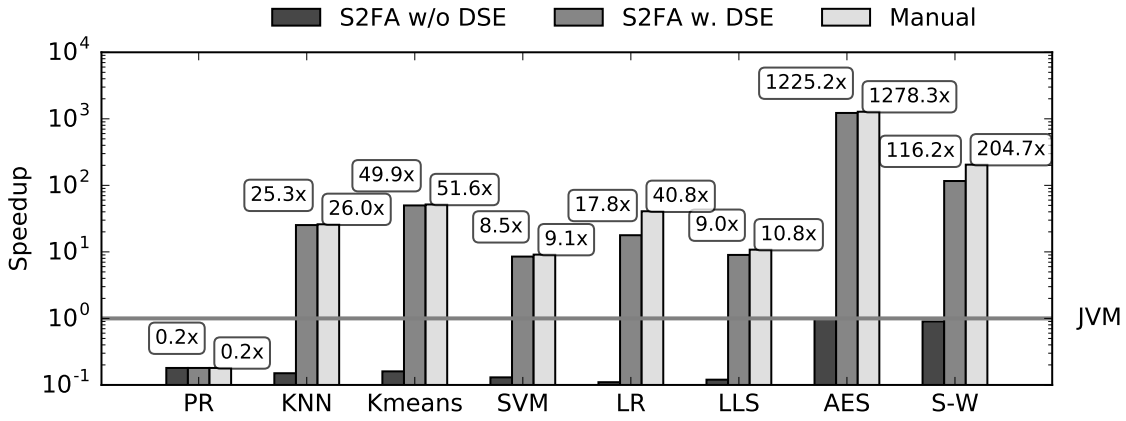


Figure 4.3: Speedup of S2FA Designs over JVMs

executor on the JVM as a baseline because only one thread is necessary for launching FPGA and other threads are able to perform other tasks simultaneously. The manual design for each application is also implemented in HLS C. Both manual and S2FA-generated designs use Xilinx SDx 2018.2 [SDX] as the design flow. However, S2FA

only requires a few hours including the design flow execution time to generate a FPGA design when compared with manual designs, greatly reduced the development time listed in Table 4.1.

As can be seen in Figure 4.3, most S2FA-generated designs with the proposed DSE framework V4 achieve competitive speedups to the manual designs ($\sim$ 85% on average) and outperform the corresponding Scala implementations on the JVM by 181.5$\times$ on average. On the other hand, the core computation of `LR` is the regression model that involves floating point multiplication and exponential calculation so the minimal initial interval is still 13. The `LR` manual design splits the computation statement to multiple stages to form a highly efficient pipeline. Future work would try to solve this problem by analyzing such a performance bottleneck and perform automatic partitioning. In addition, since the computational pattern of `PR` is too simple to hide the communication latency, even the manual HLS implementation cannot achieve a high performance on the FPGA.

4.2 A Semi-Automatic DSE Support to HeteroCL

4.2.1 Overview

As we have illustrated in Chapter 1, although HLS facilitates the development of FPGA accelerators, it still requires designers to have hardware knowledge. One important reason is that current commercial HLS tools are based on C programming languages. C/C++/OpenCL programming languages, however, are hard to be analyzed for application semantic and suitable scheduling due to their ambiguity. As a result, the users have to reconstruct the program to guide the tool to

realize specific architectures. To address this challenge, a promising trend is to further abstract the accelerator design with domain specific languages (DSLs) to obtain more application specific user hints while hiding underlying architecture details. For example, Halide [RBA13] and TVM [CMJ18] are widely known programming infrastructures for image processing and deep learning applications, respectively. Both of them propose a DSL that decouples temporal scheduling functions from algorithm descriptions. It means that only the scheduling part of the application is hardware dependent and has to be adjusted when porting the design to difference platforms.

Similar to Halide and TVM, HeteroCL [LCH19] is a multi-paradigm programming infrastructure for FPGAs with a Python-based DSL that separates algorithm and scheduling functions. The algorithm part of the HeteroCL DSL supports both declarative and imperative programming models to achieve high programmability, and its scheduling part includes compute/data customization for designers to detail specify the underlying micro-architecture. In this section, we attempt to partially automate the HeteroCL scheduling function generation by the proposed DSE framework. Specifically, we automate the architecture related scheduling primitives for HeteroCL DSL so that programmers only have to focus on the program structure scheduling such as loop transformations and data dependencies.

In the rest of this section, we first introduce HeteroCL and its scheduling primitives in Section 4.2.2, followed by the DSE support in Section 4.2.3. Finally, the evaluation result is presented in Section 4.2.4.

4.2.2 Preliminary: HeteroCL

We illustrate the HeteroCL DSL with an example in Code 4.8. A HeteroCL program is composed of two parts. The algorithm parts (line 1-10) describes the function of the program. In this example, we implement a digit recognition using K-nearest neighbor algorithm with three major steps. First, since all input 7×7 images are serialized to a 49-bit unsigned integer, we use XOR to obtain the difference between training image (labeled) and test image. Then we pop count the number of 1s in the difference to get their distance. With the distance, we update the 10×3 matrix that outputs the shortest three distances of each digit. The voting part to determine the final recognition result will be performed on the host.

In addition, line 12-20 of Code 4.8 presents one possible scheduling function for this application. In order to improve the granularity of parallelism, we merge all three steps' innermost loop, which iterates over 1,800 training images, and reorder them to be the outer loop (line 13-16). Accordingly, we can achieve decent performance by generating 10 processing elements (PEs) and dataflow pipeline (line 19-20). The corresponding Merlin C code is shown in Code 4.9.

A HeteroCL program will first be compiled to an extended Halide intermediate representation (IR) [RBA13]. The Halide IR is an in-memory IR for dataflow representation. Each IR node represents a primitive operator for two or more tensor arrays. HeteroCL extends Halide IR to better support FPGA related scheduling, such as pipeline and data bit-width customization. From the extended Halide IR, HeteroCL backend generates low level kernel implementations for different design frameworks. It currently supports SODA [CCW18] for stencil computation, PolySA [CW18] for systolic array architecture, and Merlin C [CHP16a] for others.

Code 4.8: KNN in HeteroCL DSL

```
def knn(test_img, train_img):
  # Algorithm implementation
  diff = hcl.compute((10, 1800),
    lambda x, y: train_img[x][y] ^ test_img, "diff")
  dist = hcl.compute(diff.shape,
    lambda x, y: popcount(diff[x][y]), "dist")
  knn_mat = hcl.compute((10, 3), lambda x, y: 50, "init")
  hcl.mutate(dist.shape,
    lambda x, y: update_knn(dist, knn_mat, x, y), "update")
  return knn_mat

s = hcl.create_schedule([test_img, train_img], knn)
# Loop transformation (e.g., merge and interchange)
s[knn.diff].compute_at(s[knn.update], knn.update.axis[0])
s[knn.dist].compute_at(s[knn.update], knn.update.axis[0])
s[knn.update].reorder(knn.update.axis[1], knn.update.axis[0])

# Loop scheduling
s[knn.update].parallel(knn.update.axis[1])
s[knn.update].pipeline(knn.update.axis[0])
```

Code 4.9: Corresponding Merlin C Code from Code 4.8

```
for (int i = 0; i < 10; ++i)
  for (int j = 0; j < 3; ++j)
    knn_mat[i][j] = 50;
#pragma ACCEL pipeline
for (int j = 0; j < 1800; ++j) {
#pragma ACCEL parallel
  for (int i = 0; i < 10; ++i) {
    diff[i][j] = train_img[i][j] ^ test_img;
    dist[i][j] = popcount(diff[i][j]);
    update_knn(dist, knn_mat, i, j);
  }
}
return knn_mat
```

Table 4.6: HeteroCL Scheduling Primitives

Loop Transformations	
C.split(i ,v)	Split loop i of operations C into a two-level nest loop with v as the factor of the inner loop.
C.fuse(i, j)	Fuse two loop i and j of operation C in the same nest loop into one.
C.reorder(i, j)	Reorder the order of sub-loop i and j of operation C (sub-loop i becomes an outer loop).
P.compute_at(C, i)	Merge loop i of the operation P to the corresponding loop level in operation C.
Loop Scheduling	
C.unroll(i, v)	Unroll loop i of operation C by factor v.
C.parallel(i)	Schedule loop i of operation C in parallel.
C.pipeline(i, v)	Schedule loop i of operation C in pipeline with v as the target II.

In addition, Table 4.6 lists all available HeteroCL scheduling primitives. We summarize them to two categories. The loop transformation primitives affect the loop structures. These primitives will be processed when compiling a HeteroCL program to the IR. In other words, the IR accepted by backend code generators are already transformed accordingly and the backend code generators will not see the original IR. On the other hand, the loop scheduling primitives become annotations of IR nodes. For example, `C.parallel(i)` creates an annotation on the IR node of loop-i to indicate its scheduling. As a result, this category is platform-dependent

Code 4.10: KNN Code Snippet in HeteroCL DSL with a Partial Scheduling Function

```
1  s = hcl.create_schedule([test_img, train_img], knn)
2  # Loop transformation (e.g., merge and interchange)
3  s[knn.diff].compute_at(s[knn.update], knn.update.axis[0])
4  s[knn.dist].compute_at(s[knn.update], knn.update.axis[0])
5  s[knn.update].reorder(knn.update.axis[1], knn.update.axis[0])
6
7  # Loop scheduling
8  s[knn.update].pipeline(knn.update.axis[1],
9                         auto={'options': ['off', 'on']})
10 s[knn.update].parallel(knn.update.axis[0],
11                         auto={'options': ['factor','power-of-two']})
```

and should be the main focus when porting a HeteroCL program to a new device.

4.2.3 Semi-Automated Design Space Exploration

Our objective of support to HeteroCL is to ease human efforts when optimizing the HeteroCL scheduling functions. In particular, we attempt to automate the loop scheduling related primitives so that a HeteroCL program can be easily ported to a new platform. To this end, we improve the HeteroCL primitives to support fuzzy loop scheduling. We use the same KNN example from the previous section in Code 4.10 for demonstration.

We can see from line 8 and line 10 that users could specify a set of possible options for loop scheduling primitives in their HeteroCL program. For numerical options such as parallel factors, we predefined commonly used sets (e.g., power-of-two numbers and divisors of the loop trip-count) for users to specify. The corresponding Merlin C code with design space is shown in Code 4.11 and it can directly be an input of our DSE framework presented in Chapter 3. On the other hand, we automatically

Code 4.11: Corresponding Merlin C Code with Design Space from Code 4.10

```
for (int i = 0; i < 10; ++i)
  for (int j = 0; j < 3; ++j)
    knn_mat[i][j] = 50;
#pragma ACCEL pipeline auto{options: PIP1=["off", "on"]; default: "off"}
for (int j = 0; j < 1800; ++j) {
#pragma ACCEL parallel auto{options: PAR1=[1,2,4,5,8,10]; default: 1}
  for (int i = 0; i < 10; ++i) {
    diff[i][j] = train_img[i][j] ^ test_img;
    dist[i][j] = popcount(diff[i][j]);
    update_knn(dist, knn_mat, i, j);
  }
}
return knn_mat
```

create a design space as we did in Chapter 3 when user does not specify any loop scheduling primitives in the HeteroCL program. Consequently, our DSE support helps users rapidly explore a small design space with certain loop transformations to realize the best transformation for the design. Once the loop transformation has been determined, the user could enlarge the design space to perform a more sophisticate search in order to achieve the global best performance.

4.2.4 Experimental Evaluation

We use Amazon EC2 F1 instance [AWS], `f1.2xlarge` that includes an 8-core CPU with 122GB of main memory and one Xilinx Virtex UltraScale+^{TM} VU9P FPGA, in this evaluation. We select four applications to evaluate how the DSE framework can help improve the HeteroCL usability.

- `DIGITREC`: Digit recognition using K-nearest neighbor algorithm. The input is

18,000 7×7 training images grouped by their digits, as well as one test image to be recognized. All image pixels are binary and have been serialized to a 49-bit unsigned integer. The output is a 10×3 matrix to indicate the three shortest distances between the test image to each digit group.

- `KMEANS`: K-Means clustering algorithm that takes 320 data points with 32 dimensions each and clusters them to 16 groups. Our K-Means kernel performs 200 iterations and outputs the final coordinates of cluster centers.

- `S-W`: Smith-Waterman algorithm is a 2-D dynamic programming algorithm for inexact string matching. The kernel accepts a set of 128 character string pairs, and outputs the same number of 256 character aligned string pairs.

- `CONV`: A layer including convolution, ReLU and pooling operations, in AlexNet, a well-known convolutional neural networks [KSH12]. The input of the layer is 256 228×228 channels and the output is 256 114×114 features. The convolutional kernel size is 5×5.

Table 4.7: Step-by-Step Loop Transformation Application

	V1	V2
DIGITREC	+Loop Merging	+Loop interchange
KMEANS	+Loop interchange	N/A
S-W	N/A	N/A
CONV	+Loop Splitting	+Loop interchange

Figure 4.4 depicts the evaluation results while the applied loop transformations for each version are listed in Table 4.7 (note that the version here is different from

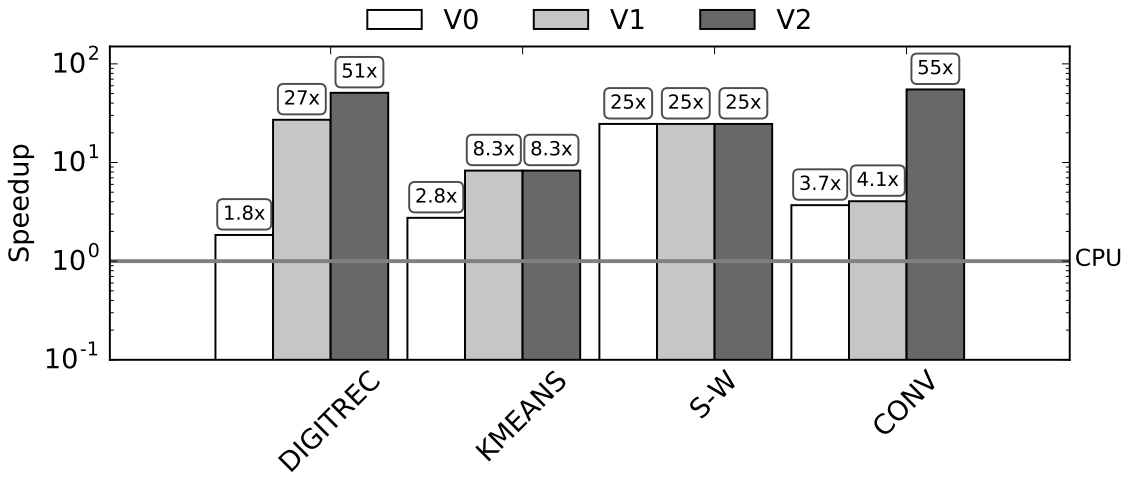


Figure 4.4: Overall Performance Comparison with CPU Baseline

the versions used in the DSE framework). As can be seen, most cases are benefit from loop transformations in addition to loop scheduling primitives. This again illustrates the importance of the design space optimality. For `DIGITREC`, since the performance of V0 is limited by fine-grained parallelism granularity, we apply loop merging to fuse three inner loops (`diff`, `dist` and `upate` in Code 4.8) at V1 and improve the performance by 14.7×. Subsequently, we find at the best design point of V1 creates 10 PEs at the outer loop to calculate the distance for each digit group simultaneously, but it fails to further improve the parallelism inside the PE because of the loop carried dependency. As a result, we further apply loop interchange at V2 so that the smaller loop with trip-count 10 can be fully unrolled and the larger loop with trip-count 18,000 can be tiled and processed in pipeline manner.

In addition, for `KMEANS`, although we have applied loop interchange at V1 to remove the main performance bottleneck of computing distances between points, the process of updating the center coordinates cannot be scheduled well due to random

access and true data dependency. Therefore, it only achieves 8.3× speedup. For `S-W`, on the other hand, dose not benefit from any supported loop transformations, because the dataflow of 2-D dynamic programming depends on the input data. As a result, the original version with proper parallelism and pipeline explored has already achieved decent performance. Finally, the intuitive convolutional layer implementation puts the loop of iterating output features to the outermost while the loop of performing kernel convolution to the innermost. This loop order cannot create sufficient fine-grained parallelism to maximize the throughput due to the lack of data reuse as well as the on-chip memory size limitation. Thus, we apply loop splitting to the outermost loop and reorder the split (tiled) loop to the innermost, so that the DSE is able to maximize the PE throughput and achieve 55× speedup over CPUs. It is worthwhile to mention that other ways of loop interchanges in this case cannot remove the carry dependency and largely limit the optimal performance (only ∼5-10× over the CPU) in the design space.

4.3 Conclusion

In this chapter, we present high-level domain specific language (DSL) supports (i.e., Spark [ZCF10] and HeteroCL [LCH19]) to expend the usability of the DSE framework. DSE can benefit from DSLs by their plenty semantic information. We use Spark-to-FPGA-Accelerator (S2FA) framework to demonstrate the design space pruning with parallel patterns. We also leverage HeteroCL to show that users directly perform platform-independent code change for optimizing the design space can be the key of the ultimate performance, and our DSE framework is able to reduce the effort of realizing the best design point.

On the other hand, even we could apply many strategies to prune the design space, the bottleneck of performing design space exploration for a design with arbitrary functionality is still the evaluation methodology that deals with a trade-off between accuracy and evaluation time. In the next chapter, we demonstrate that this trade-off could be alleviated by proposing a general-purpose micro-architecture template.

CHAPTER 5

Design Space Exploration with Architecture Templates

5.1 Overview

Since the DSE framework proposed in Chapter 3 supports arbitrary HLS designs, it has to use commercial HLS tool to evaluate the design quality and results in long exploration time. On the other hand, we observe that many computation kernels that are suitable for FPGA acceleration suffer similar performance bottlenecks, as we concluded in Table 1.1. In this chapter, we propose the composable, parallel and pipeline (CPP) micro-architecture, an accelerator design template with high flexibility to bound the design space by considering those reasons of poor performance. With the use of micro-architecture template, we can not only reduce the size of design space, but also derive an analytical model to analyze and evaluate the design space as well as the performance and resource consumption. Accordingly, we further propose a series of pruning strategies to prune the design space so that it can be exhaustively searched within an hour. In summary, this chapter makes the following contributions:

- **The CPP micro-architecture and the analytical model.** By introduc-

ing an accelerator design template like CPP, we are able to perform design space exploration efficiently using the corresponding performance and resource model.

- **The pruning strategies.** We propose a series of pruning strategies to reduce the design space, so that the optimal design configuration can be found exhaustively in one hour.
- **An automation framework.** We automate the accelerator generation and optimization process by implementing a framework and thus substantially improves the FPGA programmability.

Our experiments show that the generated accelerators outperform their corresponding software implementations by an average of 72× for the MachSuite [RAS14] computation kernels.

The rest of this chapter is organized as follows. Section 5.2 formulates the scope of the problem we target followed by the CPP micro-architecture. Section 5.3 derives the corresponding analytical model for performance and resource utilization. Section 5.4 illustrates the design space exploration to the proposed model with pruning strategies, followed by the experimental result in Section 5.5.

5.2 CPP Accelerator Design Template

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

5.2.1 Problem Formulation

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

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

We develop an algorithm based on the polyhedral analysis from [PZS13] to determine if an input program meets the constraints. Based on our problem formulation, computational kernels featuring extensive random accesses on a large memory footprint, e.g., PageRank [PBM99] and the breadth-first search (BFS) algorithm, will probably not meet the *Cacheable* constraint. On the contrary, computational kernels that process input data block by block generally meet these constraints. In fact, almost all streaming and batch processing kernels with regular data-level parallelism fall into this category. These kernels are also well-known to potentially benefit from FPGA acceleration. For the kernel that satisfies the above constraints, we implement it using our proposed micro-architecture, which we will discuss in the following

section, to bound the design space.

5.2.2 CPP Micro-architecture

The composable, parallel and pipeline (CPP) micro-architecture is proposed as a template of accelerator designs. For an input kernel that meets the above constraints, our approach first fits the kernel into the CPP micro-architecture, then performs design space exploration to identify the optimal parameter configuration, and finally transforms the input kernel code to the CPP micro-architecture description code. The CPP micro-architecture guarantees the quality of the output accelerator design by providing a series of features to realize the transformations we summarized in Table 1.1. In the remainder of this section, we introduce the key features of the CPP micro-architecture, along with the N-W motivating example in Code 1.2.

Feature #1: Coarse-grained pipeline with data caching. Figure 5.1 illustrates the N-W accelerator design under the CPP micro-architecture. The overall CPP micro-architecture consists of three stages: `load`, `compute` and `store`. The `kernel` function in the NW source code only corresponds to the `compute` module instead of defining the entire accelerator. The input sequence pairs are processed block by block, i.e., iteratively loading a certain number of sequence pairs into on-chip buffers (Stage `load`), aligning these pairs (Stage `compute`), and storing the post-aligned pairs back to DRAM (Stage `store`). This feature guarantees off-chip data movement only happens in the `load` and `store` stages, leaving data accesses of computation completely on chip. In general, as far as the input kernel meets the *Cacheable* constraint, it is able to fit into this load-compute-store execution process.

The `load` and `store` modules connect to two input and output DRAM buffers,

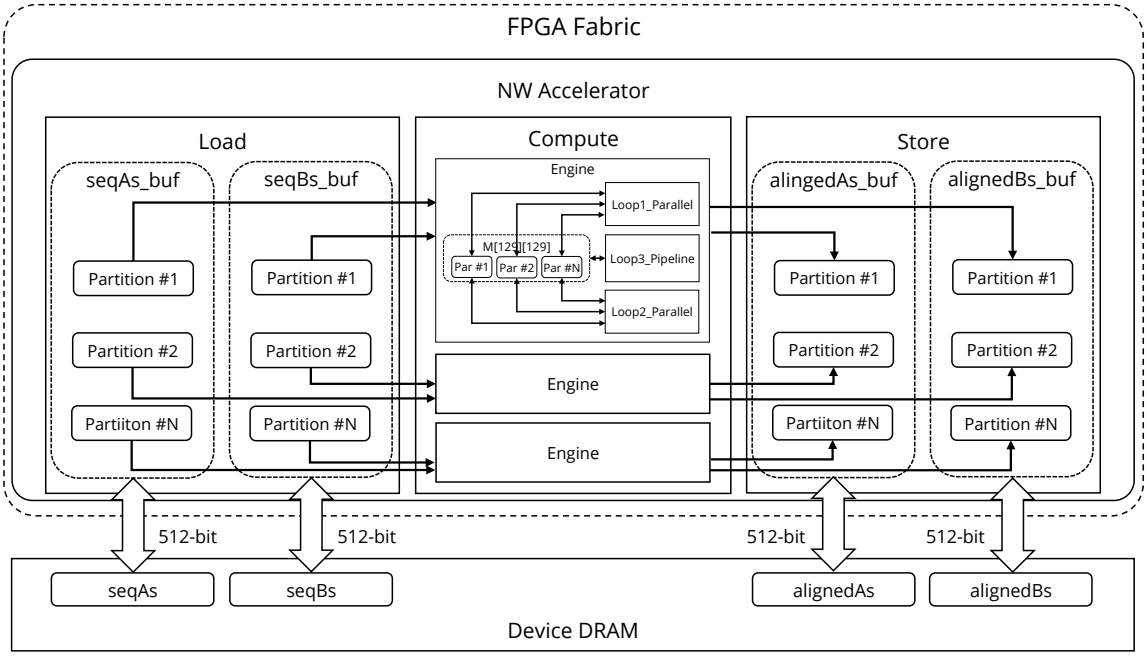


Figure 5.1: N-W Accelerator under CPP Micro-architecture

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

Feature #2: Loop scheduling. The CPP micro-architecture tries to map every loop statement presented in the computational kernel function to either 1) a circuit

that processes different loop iterations in parallel, 2) a pipeline where the loop body corresponds to the pipeline stages, or 3) a combination of both. As for the N-W example, the loop statement in the `kernel` function is mapped to a set of `NW` modules to process the sequence pairs in parallel. The loop statements in the `NW` function are mapped to parallel and pipeline circuits as well.

Feature #3: On-chip buffer reorganization. In the CPP micro-architecture, all the on-chip BRAM buffers are partitioned to meet the port requirement of parallel circuits, where the number of partitions of each buffer is determined by the duplication factor of the parallel circuit that connects to the buffer. In the N-W example, the on-chip buffers that cache the input and output sequence pairs are partitioned into multiple segments, each segment feeding one `NW` module. The local buffer `M` that stores the score matrix is also partitioned to allow parallel read and write transactions.

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

5.2.3 Design Space Analysis

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

$$\mathcal{A} = \{\mathcal{L}, \mathcal{B}\} \tag{5.1}$$

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

We then formulate the possible scheduling of loops as follows:

$$\forall L \in \mathcal{L}, L = \{(\alpha, \beta) \mid 1 < \alpha < L_{tc}, \beta = \{0, 1\}\} \tag{5.2}$$

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

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

$$\begin{gathered} \forall B \in \mathcal{B}, B = \{(\mu, \nu) \mid 8 \leq \mu \leq 512, 0 \leq \nu \leq C_{BRAM}\} \\ \sum_{B \in \mathcal{B}} B_\nu \leq C_{BRAM} \end{gathered} \tag{5.3}$$

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

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

5.3 Analytical Models

While a number of previous studies have attempted to model FPGA designs [KPZ16, WHZ16, ZMS16, ZPL16, ZPW17, ZFS17], our model[1] targets at a well-defined accelerator micro-architecture and thus features a highly accurate modeling of the utilization of the FPGA on-chip resources.

On the other hand, some of the existing models for general FPGA accelerator designs focus on only the performance estimation [ZMS16, WHZ16]. Although others also have the model for different kind of resources [KPZ16, ZPL16, ZPW17, ZFS17], their LUT models are either based on machine learning [KPZ16, ZPW17] or even missing [ZPL16, ZFS17].

5.3.1 Performance Modeling

The performance model estimates an accelerator's overall execution cycle (C):

$$C = max(C_l + C_s, C_c) \tag{5.4}$$

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

The execution cycles of the load, compute and store modules, as well as all of their submodules, can be quantified as the total cycles of all the loops (C_{loop}), submodules (C_{mod}) and standalone logic (C_r):

[1]The development of analytical models in this section was done jointly with Peng Wei [WEI18].

$$C_{mod}(M) = \sum_{i \in M.loops} C_{loop}(i) + \sum_{m \in M.mods} C_{mod}(m) + C_r(M) \tag{5.5}$$

where M denotes an arbitrary hardware module.

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

$$C_{loop}(L) = C_{iter}(L) + II(L) \times \frac{TC(L)}{UF(L)} \tag{5.6}$$

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

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

$$C_{iter}(L) = \sum_{i \in L.loops} C_{loop}(i) + \sum_{m \in L.mods} C_{mod}(m) + C_r(L) \tag{5.7}$$

Equation 5.6 and Equation 5.7 reflect the architecture hierarchy with nested modules and loops. The proposed model recursively traverses all the loops and modules until a loop or module does not contain any sub-structures. In addition, we can find that Equation 5.6 and Equation 5.7 are almost identical. This is because the loop iteration can be treated as a special "module" and modeled in the same way for both performance and resource. Hence, we omit the loop iteration breakdowns in the following resource models.

5.3.2 Resource Modeling

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

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

$$R^{mem}_{mod}(M) = \sum_{b \in M} R^{mem}_{buf}(b) + \sum_{m \in M.mods} R^{mem}_{mod}(m) \times DF(m) \tag{5.8}$$

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

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

$$R^{mem}_{buf}(B) = \prod_{d \in dim(B)} PF(d) \times V\left(\lceil \frac{S(B)}{\prod_d PF(d)} \rceil, bw(B) \right) \tag{5.9}$$

The function $V(s, bw)$ calculates the BRAM consumption of a single partition:

$$V(s, bw) = \lceil \frac{s}{N_{blk}(bw) \times S_{unit}} \rceil \times N_{blk}(bw) \tag{5.10}$$

where S_{unit} denotes the size of a BRAM block that is a platform-dependent constant, and $N_{blk}(b)$ is another function to calculate the minimum number of BRAM blocks needed to compose a buffer with bit-width b:

$$N_{blk}(bw) = \lceil \frac{bw}{bw_{phy}} \rceil \tag{5.11}$$

where b_{phy} is a platform-dependent constant that represents the largest supported bit-width of a BRAM building block.

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

$$\begin{aligned} R^{lut}_{mod}(M) = & \sum_{l \in M.loops} R^{lut}_{iter}(l) \times UF(l) + \sum_{b \in M.bufs} R^{lut}_{buf}(b) \\ & + \sum_{m \in M.mods} R^{lut}_{mod}(m) \times DF(m) + R^{lut}_{r}(M) \end{aligned} \tag{5.12}$$

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

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

$$R^{lut}_{loop}(L) = \sum_{l \in L.loops} R^{lut}_{iter}(l) \times UF(l) + \sum_{m \in L.mods} R^{lut}_{mod}(m) \times DF(m) + R^{lut}_{r}(L) \tag{5.13}$$

Note that since we always perform loop-invariant code motion in advance, we guarantee that there has no BRAM used in the loop body.

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

$$R^{lut}_{buf}(B) = R^{mem}_{buf}(B) \times (R^{lut}_{ctrl} + R^{lut}_{data}) + R^{lut}_{mux} \left(\prod_{d \in dim(B)} PF(d) \right) \times bw(B) \tag{5.14}$$

where $R^{lut}_{mux}(k)$ can be calculated using Eq. 7 in [CWY17].

These equations quantify the relationship between a buffer's LUT consumption and its BRAM usage.

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

5.4 Design Space Exploration

Figure 5.2 illustrates our design space exploration (DSE) flow. The DSE flow first initializes the analytical model by performing HLS synthesis instances and parsing the generated reports, and then fetch the set of design parameters from the C kernel code. As we pointed out in the previous section, exhaustively searching in such a tremendous design space is impractical. As a result, we propose the following strategies to prune the design space:

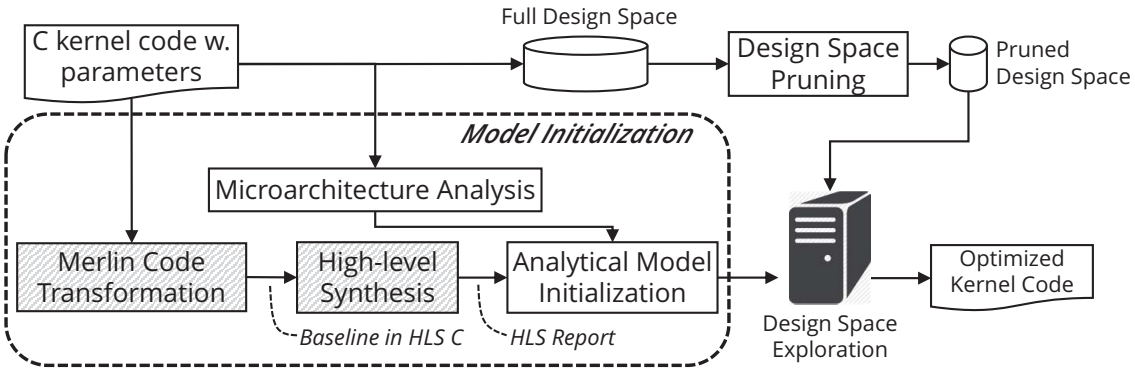


Figure 5.2: Design Space Exploration Flow

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

Loop unroll factor pruning: Loop unroll factors determine the number of on-chip BRAM partitions. This number is bounded by the total number of BRAM blocks available for user-defined accelerators, which is approximately a few thousand. This

pruning strategy is particularly beneficial for programs with deep, complicated loop hierarchy.

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

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

By applying the Saddleback search algorithm [BIR06] to the formulated problem, we can reduce the time complexity of searching all loop unroll factors from $O(\prod_{L\in\mathcal{L}} L_{tc})$ to $O(\prod_{L\in\mathcal{L}\wedge L\notin\{L_p,L_q\}} L_{tc} \times L_p \times \log\frac{L_q}{L_p})$, where L_q and L_p denote the unroll factors of the two loops with the largest trip counts. This strategy works very well for programs with shallow loop hierarchies.

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

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

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

Power-of-two buffer bit-widths and capacities: We prune the design space by only searching the power-of-two bit-width and capacity values for each buffer. We note that this pruning strategy covers the optimal design point because 1) the BRAM utilization would be the same for all bit-width values that round up to the same power-of-two value, and 2) setting the capacity to be a power-of-two value achieves the highest efficiency for the buffer control logic and is enabled in commercial HLS tools by default.

Taking the N-W example as an instance, the design space is reduced from 1.5×10^9 to only 3.2×10^6 by applying the above strategies. The scale of reduced design space is sufficient to be searched within an hour even using a single modern CPU core.

5.5 Evaluation Results

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

5.5.1 The Framework

As shown in Figure 5.3, we implement a push-button framework that takes a nested loop in C as input and performs a series of transformations to produce a high-quality FPGA accelerator under the CPP micro-architecture. Like the DSE framework in Chapter 3, this framework is implemented on top of the Merlin compiler [MER, CHP16a]. We leverage the code transformation primitives provided by the Merlin compiler to agilely construct the CPP micro-architecture. On the other hand, without the need for users to manually insert directives in the input code, the CPP micro-architecture provides an automated way to organize these primitives to come up with high-quality designs. Subsequently, we use static analysis to extract the necessary information (e.g., loop trip count) to form the design space. Then the design space exploration flow we introduced in the previous section is adopted to realize the best design specification in minutes. This design can be directly fed into Xilinx SDAccel to produce a high-quality accelerator bit-stream.

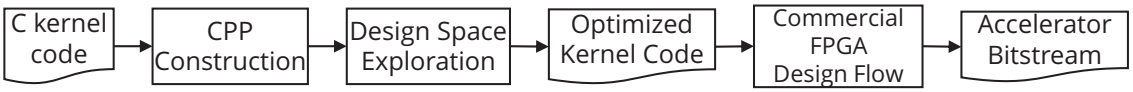


Figure 5.3: Framework Overview

5.5.2 Experimental Setup

The evaluation is performed on the mainstream PCIe-based CPU-FPGA platform with the Xilinx SDAccel design flow. We use the MachSuite [RAS14] benchmark suite that contains a broad class of computational kernels programmed as C functions for accelerator study. For each kernel, MachSuite provides at least one implementation

that is programmed without the consideration of FPGA acceleration, which makes it a natural fit for this evaluation.

Table 5.1: Configuration of Hardware and Software

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

5.5.3 Evaluation Results

We first evaluate the error rate between the model-generated result and the HLS report. The average error rate for cycle count, BRAMs, DSPs, LUTs and FFs are only ¡1%, ¡1%, ¡1%, 6.5% and 4.3%, indicating that the proposed model aligns to the HLS report accurately. We then compare this result with the actual on-board result, and list the error rate for each benchmark in Table 5.2. We can see that the average error rate among all the benchmarks is only 6.2%. We further analyze the benchmarks with over 10% error rate, i.e., `AES` and `KMP`. We find that such a relatively large error rate is mainly because the accelerator designs for these benchmarks have a very small execution time (∼10 ms). For these time frames, the start-up and end overhead bias the time significantly. On the contrary, we also observe that the error rate of the model to on-board execution is always less than 5% when a design has an over 100-millisecond execution time. Hence, the proposed model is able to accurately

predict the on-board execution time of a design given that its execution time is tens of milliseconds or larger.

Table 5.2: Error Rates Between Model and On-board Results

Case	AES	SPMV	KMP	FFT	VITERBI	NW	STENCIL	GEMM
Error	13.5%	9.5%	12.2%	0.1%	2.1%	1.1%	7.7%	3.3%

We then evaluate the performance improvement of the generated FPGA accelerator designs. Figure 5.4 compares the performances between the naive implementation of MachSuite, generated accelerator designs and manual HLS designs, all of which are normalized to the performances of the corresponding software implementations. We can clearly see that generated accelerators outperform the naive implementations by 27,000×, indicating that the framework dramatically improves the quality of accelerator designs without manual programming effort. Meanwhile, the generated accelerators also outperform the software implementations by 72×, indicating that our approach does lead to competitive accelerator designs.

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

Although we are able to further improve the performance of other kernels by manually applying very specialized circuit designs not covered by the CPP micro-architecture, e.g., Race Logic [MSS14] for the N-W kernel, we still preserve a high

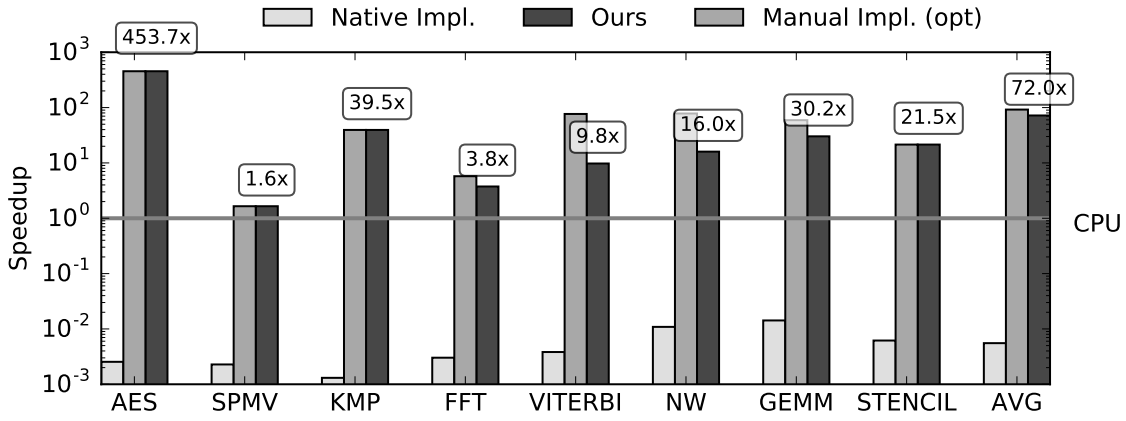


Figure 5.4: Speedup over an Intel Xeon CPU Core

quality of results while substantially reducing the programming effort.

5.6 Conclusion

We propose composable, parallel and pipeline (CPP) micro-architecture template in this chapter to facilitate the DSE process for HLS on FPGAs. Featuring the CPP micro-architecture, analytical-based design space exploration and automatic code transformation, we are able to achieve 72× speedup for a broad class of computation kernels within an hour of exploration time. Furthermore, we believe that the design principles CPP can be further generalized to stimulate more research on the adoption of FPGAs in datacenters. The CPP micro-architecture serves as a proof-of-concept that using accelerator design templates as specifications of the program-to-behavioral-description transformation fundamentally reduces the design space while preserving the accelerator quality. Future work would support more micro-architectural templates and more sophisticated code transformation techniques

to improve the coverage of computation kernels.

While the DSE with CPP micro-architecture still requires a few number of HLS runs for analytical model initialization in order to capture the unpredictable heuristic optimization in the commercial HLS tools; this issue could be identified and reflected to the analytical model in advance when we limit the user applications to a specific domain. It means we do not require HLS run during the DSE process anymore but can still achieve the best performance. In the next chapter, we use convolutional neural network (CNN) as the target domain to demonstrate this idea.

CHAPTER 6

Design Optimization for Systolic Array Template

6.1 Overview

Convolutional neural network (CNN) is one of the key algorithms for the deep learning applications, ranging from image/video classification, recognition, and analysis to natural language understanding, advances in medicine, and more. The core computation in the algorithm can be summarized as a convolution operation on the multiple dimensional arrays. Although the algorithm requires computation power and communication bandwidth, it also offers significant potential for massive parallelization and extensive data reuse. Hence, FPGA implementations of CNN have seen an increased amount of interest from academia [CMB10, SJC09, CSJ10, FPH09, PSM13, ZLS15, SCD16, VB16, QWY16, ZFZ16] due to the customizability of FPGAs.

Some existing CNN designs on FPGAs mainly focus on on-chip computation engine optimization by exploiting different parallel strategies [SJC09, CSJ10, CMB10, FPH09]. The studies explore parallelism opportunities in input feature maps [SJC09] and convolution kernels [CMB10, FPH09]; while the work in [CSJ10] chooses to parallelize output feature maps. These implementations customize massively parallel processing elements (PEs) on FPGAs according to specific computation types; they achieve a high performance that exceeds modern CPUs, thanks to FPGA's abundant

logic resources and reconfigurability.

On the other hand, some designs take external memory communication into consideration to achieve high throughput at system-level [PSM13, ZLS15, SCD16, VB16, MCV17]. The study in [PSM13] develops a memory-centric design method to maximize data reuse for memory bandwidth optimization. Meanwhile, to balance computation to communication ratio, the study in [ZLS15] leverages a roofline model to identify the optimal design option from a large design space, while the authors in [SCD16, VB16] propose analytical models to realize this goal. In addition, The authors in [MCV17] quantitatively analyze different optimization objects, and then propose a specific dataflow architecture to minimize data movements and memory accesses.

Although those implementations utilize FPGA resources well to achieve high throughput, the capacity of hardware resources in the FPGA increases continuously, which provides more than a thousand floating compute units in one FPGA chip—such as the Intel Arria 10 [ARR] and Xilinx Virtex UltraScale+ [XUL]. Once the existing customized designs of CNNs are applied to the latest device, the existing optimization approaches need to deal with the trade-off between high resource utilization and clock frequency, which leads to dramatic performance degradation.

To address such challenges, a suitable architecture for FPGAs plays an important role in developing a scalable CNN implementation. In particular, a systolic array architecture [KUN88] is a specialized form of parallel computing with a deep pipeline network of PEs. With regular layout and local communication, the systolic array features with low global data transfer and high clock frequency, which is suitable for large scale parallel design on FPGAs. Systolic array architecture has been

widely used in difference kinds of applications on the FPGAs, such as matrix multiplication [WC15] and bioinformatics [JBC10]. As a result, researchers attempt to map CNN inference to systolic array architecture [ZFZ16, AOC17] in recent years. Specifically, Caffeine [ZFZ16] implements the massive parallelism for CNN inference on Xilinx Kintex Untrascale device. The design in [ZFZ16] adopts a systolic-like architecture to mitigate the timing issue for the large design, but it still directly connects all PEs to the on-chip memory and results in not fully local interconnects. This is the reason that the design in [ZFZ16] is outperformed by a later work [AOC17] that adopts a complete systolic array architecture. The authors in [AOC17] propose a 1-D systolic array design in OpenCL for AlexNet [KSH12] CNN model with the help of an analytical model to realize the best design point and result in a high throughput design that outperforms all previous designs. However, this design only supports small models such as AlexNet as it assumes that all input feature maps reside in on-chip memory for computation. Moreover, applying the methodology in [AOC17] to other CNN models is not straightforward due to the lack of an automated design space exploration approach.

We note that most previous designs on systolic arrays are implemented in RTL, which is not only time-consuming but requires hardware design expertise. Moreover, the fine-grained systolic architecture requires careful attention to resource usage and timing. Hence, an automated design flow from a pure algorithmic software program to an efficient systolic array is essential for the software designers and data scientists to benefit from FPGA acceleration. In this chapter, we investigate the challenges in systolic array implementations from a nested loop construct, and propose an automated methodology to optimize the design on systolic arrays.

6.2 Systolic Array Architecture

We adopt a 3-D systolic array architecture on FPGA in Figure 6.1. The architecture can only input two data buffers (IN and W) and output one buffer (OUT). As shown in this figure, each PE shifts the data of W and IN horizontally and vertically to the neighboring PEs at each cycle. This 2-D topology matches the 2-D structure in the FPGA layout so that it can achieve timing constraint easily because of low routing complexity. In addition, the third dimension represents the SIMD vector accumulation inside each PE. The parallelization factor of the SIMD factor is usually power of two due to the dedicated inter-DSP accumulation interconnect in modern FPGAs.

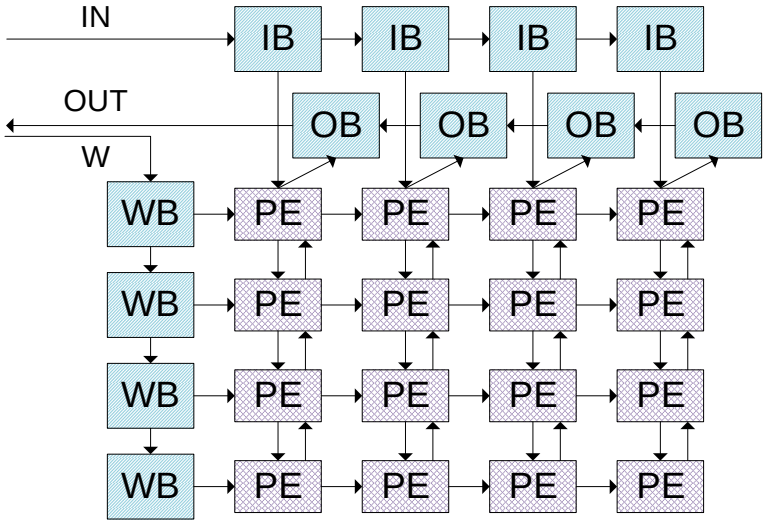


Figure 6.1: Systolic Array Architecture

This architecture is able to tackle the timing issue for massive parallelization within a design. Its key features can be summarized as 1) local interconnect and 2) shifting data transfer. As shown in Figure 6.1, the global and large fan-out inter-

connect is split into local interconnects between neighboring PEs. In addition, the input/output data are shifted into/from the PE array and between the neighboring PEs so that the multiplexes are eliminated. With the local, short, peer-to-peer interconnects, systolic array architecture can achieve high frequency even in the case of massive parallelization with over a thousand PEs.

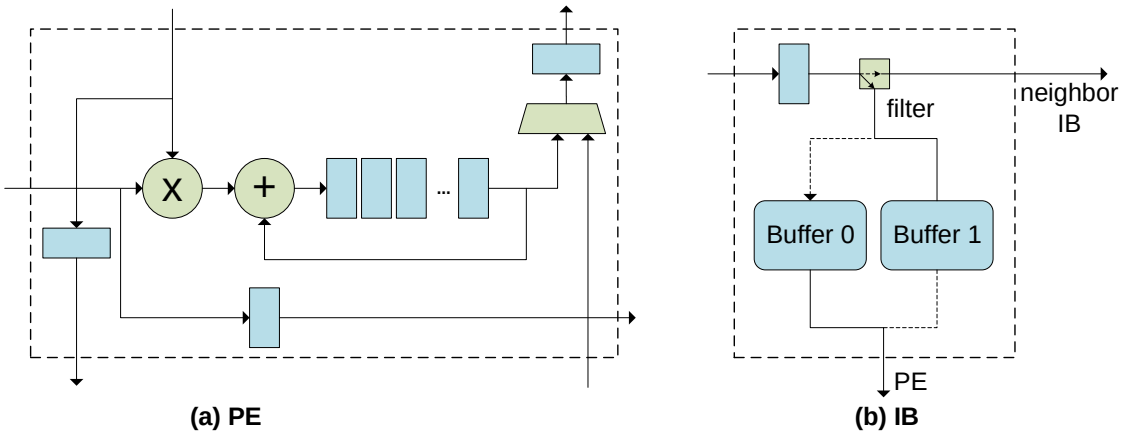


Figure 6.2: Structure of PE and Input Buffer (IB)

The architecture of PE and buffer structure to store input feature maps (IB) are presented in Figure 6.2. A PE passes input data to its neighboring PE in vertical direction and passes W to neighboring PE in horizontal direction every cycle, and it also accumulates the multiplier of the IN and W. The output data are shifted out across the PE array as well when they are ready after multiple rounds of accumulation. Each input buffer contains a double buffer – one buffer is used to store the data fetched from external memory, and the other is used to feed the data into the PE. All the input data IN are shifted across the buffers in the horizontal direction; while the input buffer will selectively store the data belonging to the corresponding column

of PEs by a filter. The similar double buffer structure is applied to the buffers for weight (WB) and output feature map (OB) as well.

The local interconnect introduces several outstanding features of a systolic array execution. First, the data required for the computation of the PEs have to be transferred from the PE boundary and across multiple PEs. Since only the boundary PE has the access to input data, data reuse between each row and column of PEs is required. More importantly, a systolic array runs in a regular and synchronized way such that fine-grained pipelining is performed between every neighboring PE. With these features, a suitable scheduling of the PE executions is essential for systolic array design, especially the synchronization of the data on each PE from different directions.

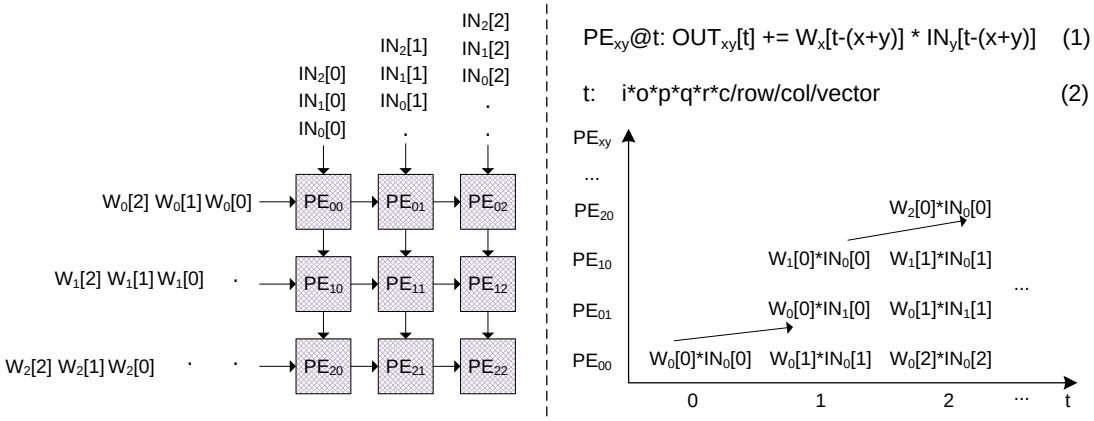


Figure 6.3: Cycle-Level Schedule of Systolic Array

Figure 6.3 shows one possible scheduling of PE execution after performing mapping feasibility check and address mapping, which will be discussed in the next section. $PE_{xy}@t$ illustrates the mapping of cycle number, PE indexes onto data access indexes, where (x, y) is PE index, and t is cycle number. It means at cycle t,

the multiplication of W and IN accumulates on OUT. Cycle number t is computed by number of total loop number being divided by $row \times col \times vector$, where row and col are the number of PEs in row and column, and $vector$ is the length of SIMD computation for the third dimension of the 3-D systolic array architecture. At cycle t, all PEs implement a multiplication and accumulation in parallel. In Figure 6.3, PE_{00} gets weight data W from vertical buffer WB and input feature map data IN from horizontal buffer IB at the first cycle. PE_{00} performs the multiplication of the two inputs and accumulates the result OUT in the register within PE where previous partial accumulation result is stored.

Meanwhile, the other PEs are stalled because no data is being received from at least one of its inputs. At cycle 1, the vertical data from PE_{00} (W) is passed to PE_{01}, and horizontal data (IN) to PE_{10}; both PE_{01} and PE_{10} have the required data to perform an execution. Meanwhile, PE_{00} performs execution with new data coming from the input buffers simultaneously. Taking 3×3 PE array in Figure 6.3 for example, after 5 cycles, all PEs will be active and synchronously read data from previous neighbor PEs, perform computation, and pass data to next PEs in each cycle. After accumulation within a PE ends, OUT in the shift register is shifted across vertical PEs until it is stored in OB.

6.3 Challenges

Although the systolic array architecture is able to significantly benefit designs on the FPGA, mapping a nest loop computation onto a systolic array structure is not straightforward. We summarize the mapping process in three steps and describe their challenges along with examples with a six-level nested loop from the CNN

configuration of AlexNet [KSH12] layer 5, $(I, O, R, C, P, Q) = (192, 128, 13, 13, 3, 3)$, as follows.

Code 6.1: A Motivating Example

```
1 L1: for(o = 0; o < O; o++)
2   L2: for(i = 0; i < I; i++)
3     L3: for(c = 0; c < C; c++)
4       L4: for(r = 0; r < R; r++)
5         L5: for(p = 0; p < K; p++)
6           L6: for(q = 0; q < K; q++)
7             OUT[o][r][c] += W[o][i][p][q] * IN[i][r+p][c+q];
```

1. Find a feasible mapping. We need to first find a feasible mapping in the systolic array to guarantee that the proper data is available at specific locations in the PE array at every cycle. Specifically, we attempt to select three of six loops to represent the 3-D systolic array: PE row, PE column and the SIMD vector inside a PE. As mentioned in the previous section, systolic array requires data reuse in both directions, so the corresponding loops need to carry the data reuse of two different arrays (W and IN), while the third loop needs to carry the accumulation of the output (OUT). Failing to satisfy this rule will cause a non-feasible mapping. For example, mapping loop *L3* and *L4* into a PE row and column is not feasible, because data reuse does not happen on array W which does not relate to neither loop *L3* nor *L4*.

2. Select a PE array shape. Next, we select the PE array shape by determining the size of each dimension, which impacts the performance in terms of 1) the required DSP number, 2) the clock frequency, and 3) the DSP efficiency. The DSP efficiency is defined as the effective computation ratio performed by DSPs, as shown in Equation 6.1.

$$Eff = \frac{effective\_operation}{total\_operation} \tag{6.1}$$

We use an example in Table 6.1 to illustrate the impact of systolic array shape. Both configurations map loop $(L1, L3, L2)$ to systolic arrays $(row, column, vector)$ but with different shapes. As can be seen, assuming the clock frequency for both configures are the same (280 MHz), *sys2* has a higher DSP utilization but a relatively lower DSP efficiency compared with *sys1*. This is because *sys2*'s shape $(16, 10, 8)$ does not match the mapped trip counts of the mapped loops $(128, 13, 192)$.

Table 6.1: Impact of Systolic Array Shape to Performance

	ROW	COL.	VEC.	DSP Util.	DSP Eff.	Peak Thrpt.
sys1	11 (L1)	13 (L3)	8 (L2)	71.5%	96.97%	621 GFlops
sys2	16 (L1)	10 (L3)	8 (L2)	80.0%	60.00%	466 GFlops

3. Determine the data reuse strategy. After we identify the systolic array mapping and shape, we determine the data reuse strategy. As mentioned previously, the data reuse for the PE array is not sufficient for achieving high throughput, so we need to determine proper tiling sizes to add several orders of magnitude of data reuse. In other words, it requires exploiting the data reuse carried on multiple for-loops with long reuse distance, which in turn leads to the large reuse buffers. However, there are more than ten thousand design options in the trade-off between the on-chip memory utilization and off-chip bandwidth saving, including selection of the arrays to be reused, the loops that carry the data reuse, and tiling sizes for the selected loops carrying data reuse.

Taking *sys1* in Table 6.1 again as an example. Since the systolic array de-

sign we used is fully pipelined, the theoretical peak throughput is $96.97\% \times 2 \times 11 \times 13 \times 8 \times 280 \simeq 621$ GFlops. This can be achieved by choosing proper tiling sizes for each loop (e.g., $Tile(I, O, R, C, P, Q) = (4, 4, 13, 1, 3, 3)$) to balance data reuse and memory bandwidth. However, if we use inappropriate tiling sizes such as $Tile(I, O, R, C, P, Q) = (2, 2, 2, 2, 2, 2)$, then we require around 67 GB/s memory bandwidth to achieve the peak throughput (the analytical model is described in the next section. In fact, we only get 162 GFlops on Intel's Arria 10 with 19 GB/s bandwidth for this low QoR configuration.

6.4 Analytical Models

All these design challenges and their interplay need to be considered in a unified way with high-level modeling. In this section, we formulate the overall optimization problem as maximizing the system throughput under the systolic mapping feasibility condition and resource constraints.

6.4.1 Architecture Abstraction

Before we can perform the detailed modeling, an abstraction of the architecture is necessary. A loop tiling representation is proposed in Code 6.2 for this purpose, which establishes the link between the architecture and high level program code. The tiled loops in the intermediate representation contains all the architecture considerations in the systolic array, such as PE array mapping, PE array shape, data reuse strategy, etc. The corresponding systolic array architecture is shown in Figure 6.4. Since this representation itself is a sequential program, it enables us to perform the modeling in a general way using program analysis techniques and tools such as polyhedral model.

Code 6.2: Loop Tiling Representation for Systolic Array Mapping

```
// outer loops
L0: for (Lo = 0; Lo < l_0; Lo++)
  ...
  Ln: for (Ln = 0; Ln < l_n; Ln++)
    // middle loops
    S0: for (So = 0; So < s_0; So++)
      ...
      Sn: for (Sn = 0; Sn < s_n; Sn++)
        // inner loops
        T0: for (T0 = 0; T0 < t_0; T0++)
          T1: for (T1 = 0; T1 < t_1; T1++)
            T2: for (T2 = 0; T2 < t_2; T2++)
              // Orignal loop body
```

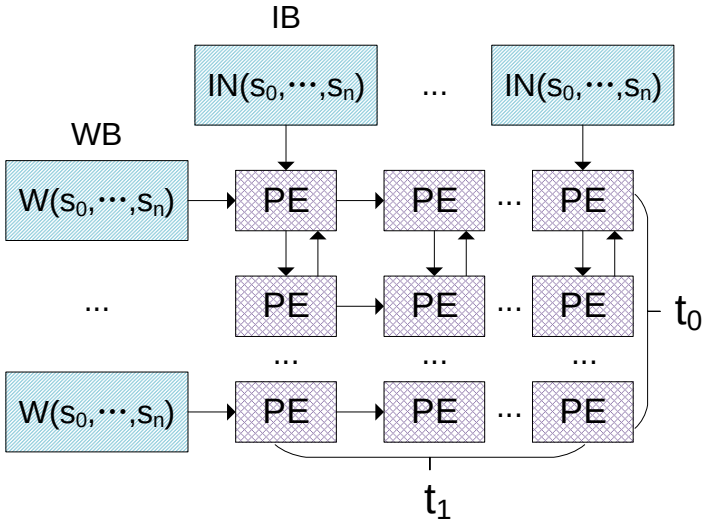


Figure 6.4: The Mapped Systolic Array Architecture

The program in Code 6.2 is transformed from the original code in Code 6.1 by loop tiling. The semantic of the program is preserved by the transformation if we ignore the precision error of reordering the floating point accumulation. Then the

tiled loops are associated with the architecture consideration as the following. The overall computation is performed block by block sequentially, where each block is corresponding to an iteration of the ***outer loops*** (L_0-L_n). Since the blocks are calculated independently, the outer loops do not impact the throughput.

Once a data block is fetched from off-chip memory, it is stored in the input buffers (IB and WB) for date reuse. The ***middle loops*** (S_0-S_n) represent the sequential processing of feeding data from input buffers to the PE array. The bounds of the middle loops $\vec{s} = (s_0, ..., s_n)$ determine the sizes of the reuse buffer. The data accessed in the reuse data by the PEs are represented by the array access addresses which are indexed by iterators of the middle loops.

Parallel execution is performed in the PE array in the fine-grained pipeline. The ***inner loops*** (T_0-T_2) represent the parallelism in the PE array where each iteration of the inner loops is corresponding to a parallel DSP unit in the array. The shape of the systolic array is determined by the bounds of inner loops ($\vec{t} = (t_0, t_1, t_2)$); while the systolic array mapping feasibility is determined by the relation between the inner loop iterators and the array access addresses in the loop body.

In addition, when we perform the loop tiling, the original loop bounds may not be divisible by the tiling sizes ($\vec{s}$, $\vec{t}$) we select. This leads to a waste of computation that determines the DSP efficiency.

6.4.2 Feasible Mapping to Systolic Array

As demonstrated in the previous section, the architecture of the systolic array is determined by the three inner loops that are selected to map to PE row, PE column and SIMD vector inside the PE. There are many alternatives for this loop-to-architecture

mapping, but not every one of them can finally have a feasible mapping in the systolic fashion. The condition of the feasible systolic mapping can be summarized as: *each of the three array variables (W, IN, and OUT) has to have fine-grained data reuse carried on at least one of the three inner loops.*

By introducing the binary variables k_l to indicate loop-to-architecture mapping ($k_l = 1$ if the loop l is selected as one of the inner loops, otherwise $k_l = 0$), the feasibility condition for the mapping is formulated as

$$\sum k_l = 3, \forall r, \sum k_l \times c_{rl} > 0 \tag{6.2}$$

where c_{rl} indicates data reuse of array r on loop l: $c_{rl} = 1$ if loop l carries the fine-grained data reuse of array r, otherwise $c_{rl} = 0$

All the possible fine-grained data reuse in the program can be analyzed in advance. We use polyhedral model to represent the program [KUC78]. The program can be summarized as three aspects: an iterator vector $\vec{i}$ which lists loop iterators from the outermost loop to the inner loop in the loop nest; an iteration domain $\mathcal{D}$ which defines the range of the loop iterators; and an access function F_r which maps the loop iterators into the access indexes of array r.

The fine-grained data reuse for array r on loop l requires the data accessed on array r in different loop l iterations have to be the same, which can be formulated as the following condition:

$$\forall \vec{i} \in \mathcal{D}, F_r(...i_{l-1}, i_l, i_{l+1}, ...) = F_r(i_0, ..., i_{l-1}, i_l + 1, i_{l+1}, ...) \tag{6.3}$$

6.4.3 Resource Utilization Modeling

Since the computation is mainly floating-point multiplication and accumulation, the DSP and on-chip block RAM (BRAM) are the two critical resource types. The DSP utilization is simply determined by the product of the inner loop bounds $\vec{t}$:

$$D(\vec{t}) = \prod t_l \tag{6.4}$$

The modeling of BRAM utilization needs to consider the data reuse in the input and output buffers. Due to the double buffering mechanism for hiding data transfer overhead, the buffer size is equal to two times the data block size of the array. The block size can be modeled as the total amount of data that is accessed in the middle and inner loops in Code 6.2.

$$DA_r(\vec{s}, \vec{t}) = \left| \left\{ \vec{a} | \vec{a} = F_r(\vec{i}) \wedge \vec{i} \in \mathcal{D}_{\vec{s},\vec{t}} \right\} \right| \tag{6.5}$$

where $\vec{a}$ is the access index vector of multi-dimensional array and $\mathcal{D}_{\vec{s},\vec{t}}$ is the iteration domain of the middle and inner loops. Counting an integer set with linear constraints can be solved by the polyhedral library [VER10], but it has high computational complexity. As a result, we simplify the model by counting the range of each dimension of the array access index, so that the total size is the product of range size of each dimension. It implies that we only support two kinds of index patterns in the program: the one consists of only one iterator, such as *out[o][r][c]* and *w[i][o][p][q]*; and the other is the sum of two iterators, such as *r+p* in the access *in[i][r+p][c+q]*. For the first case, the range for the dimension is the bound of the corresponding middle and inner loops. For second case, the range can be calculated

as the sum of the bound of the iterators, e.g. if the bound of r is t_0 and bound of p is 3, the range size of $r + p$ is $(t_0 + 3) - 1$.

To simplify the address generation complexity of multiple dimensional arrays, the OpenCL design flow tool will allocate the actual memory size as the rounding up power of two value. Finally, the total BRAM utilization is formulated as follows:

$$B(\vec{s}, \vec{t}) = \sum_{r} (c_b + 2^{\lceil \log_2 DA_r(\vec{s},\vec{t}) \rceil}) + (c_p \times \prod(\vec{t})) \tag{6.6}$$

where c_b is a constant BRAM cost for the IBs and OBs, c_p is the BRAM cost for each PE, and $\vec{t}$ is the bound vector of inner loops.

6.4.4 Performance Modeling

In a systolic design, both computation and data transfer may be the performance bottleneck for different design options. The adoption of double buffering in the input and output enables us to model the throughput in a decoupled way, so the overall throughput T is dominated by the lower one of computation throughput (PT) and external memory transfer throughput (MT).

$$T(\vec{s}, \vec{t}) = min(PT(\vec{s}, \vec{t}), MT(\vec{s}, \vec{t})) \tag{6.7}$$

Since the systolic array is executed in the fully pipelined way, each PE will complete two floating point operations (multiplication and accumulation) in each cycle. However, the quantization effect may lead to wasted computation on the incomplete data blocks on the boundaries of the original loops. By defining the clock frequency as F, the computational throughput is modeled as the number of effective floating

operations in the original code performed every second:

$$PT(\vec{s},\vec{t}) = Eff(\vec{s},\vec{t}) \times \prod \vec{t} \times 2 \times F \tag{6.8}$$

where $Eff(\vec{s},\vec{t})$ is the DSP efficiency defined in Equation 6.1.

In addition, external memory transfer throughput (MT) is defined as the number of effective floating point operations performed in each block divided by the time it takes to transfer the data required by the these operations. Due to the hardware feature, the memory bandwidth limitation is not only for overall memory access BW_{total}, but for each memory access port BW_{port} (array in, w, and out). The transferred data amount and bandwidth determines the data transfer time, so MT can be modeled as follows:

$$MT(\vec{s},\vec{t}) = min(MT_t(\vec{s},\vec{t}), MT_r(\vec{s},\vec{t})), r \in R \tag{6.9}$$

$$\begin{aligned} MT_t(\vec{s},\vec{t}) &= \frac{Eff(\vec{s},\vec{t}) \times 2 \times \prod (\vec{s} \times \vec{t})}{\sum DA_r(\vec{s},\vec{t})/BW_{total}} \\ MT_r(\vec{s},\vec{t}) &= \frac{Eff(\vec{s},\vec{t}) \times 2 \times \prod (\vec{s} \times \vec{t})}{DA_r(\vec{s},\vec{t})/BW_{port}} \end{aligned} \tag{6.10}$$

6.4.5 Putting It All Together

Finally, the overall optimization problem can be formulated as the combination of the following two subproblems.

Problem 1: Given a nested loop L that has 1) two inputs and one output and 2) supported indexing patterns, finding a set S that contains all feasible systolic array configurations:

$$\mathcal{S}_L = \left\{ (\vec{k}, \vec{t}) \mid \sum \vec{k} = 3, \prod \vec{t} \leq D_{total}, \forall r, \sum k_l \times c_{rl} = 1 \right\} \tag{6.11}$$

where $\vec{k}$ is the mapping vector, and $\vec{t}$ is the bounds of the inner loops.

Problem 2: Given a systolic array configuration $(\vec{k}, \vec{t})$, finding the optimal bounds of the middle loops $\vec{s}$ so that the overall design throughput is maximized:

$$\text{maximizing } T(\vec{s}, \vec{t}), \text{ s.t. } B(\vec{s}, \vec{t}) < B_{total}, D(\vec{t}) < D_{total}$$

where T, B, and D have been defined in Equation 6.7, Equation 6.6, and Equation 6.4, respectively.

The complex calculation of $B(\vec{s}, \vec{t})$ and $H(\vec{s}, \vec{t})$ makes the problem neither linear nor convex, which in turn leads to the difficulty in analytical solving. On the other hand, the entire design space of the two problems is tremendously large, which makes brute-force search impractical. For example, our implementation spends roughly 311 hours on traversing every design option for one of the convolutional layers from the AlexNet [KSH12] CNN model on Intel's Xeon E5-2667 CPU with 3.2 GHz frequency. In the next section, we will show that the size of design space can be reduced significantly when taking practical hardware architecture into consideration.

6.5 Design Space Exploration

Under the performance and resource modeling, our design space exploration can identify a valid design option with the highest throughput. However, the working frequency for a design is hard to model. As a result, we develop a two-phase process in Figure 6.5 which first filters the design space into a small set of candidates using

the proposed analytical model in the previous section with a given clock frequency, and then goes through the hardware generation flow for the selected designs to obtain the one that has the best on-board performance.

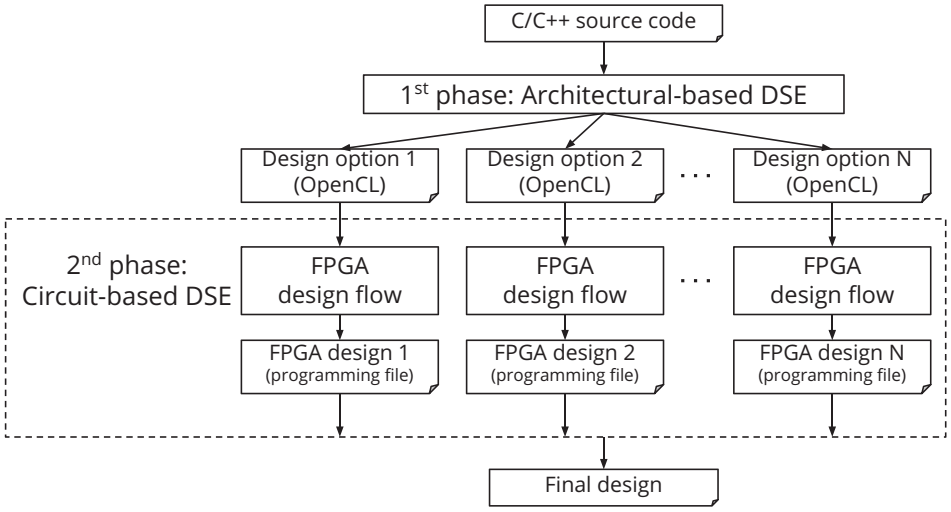


Figure 6.5: Two-Phase Design Space Exploration

In the architectural-based phase, we reduce the design space by considering resource utilization and on-chip BRAM features. Due to the scalability of the systolic PE array architecture we adopted, the clock frequency will not drop significantly with low DSP utilization, so we can prune the design options with low DSP utilization by adding the following constraint into Problem 1.

$$\prod \vec{t} \geq c_s \times D_{total} \tag{6.12}$$

where c_s is a constant to set a lower bound of DSP utilization defined by a user. The value of c_s determines the design space of the rest of the process. For example, by applying Eq. 6.12 with $c_s = 80\%$, the number of available systolic PE array

mappings is reduced from 160K to 64K for one of the convolutional layers from AlexNet [KSH12].

In addition, we also reduce the design space of data reuse strategies in terms of value of $\vec{s}$ by leveraging the fact that BRAM sizes in the implementation are always rounded up to a power of two. In details, we prune the design space by only exploring the candidates of $\vec{s}$ whose values are power of two. The pruned design space of data reuse strategies can cover the optimal solution in the original design space because 1) our throughput object function is a monotonic non-decreasing function of $\vec{s}$, and 2) BRAM utilization is the same for the options of $\vec{s}$ whose values have the same rounding up power of two. By applying the pruning on the data reuse strategies, the design space reduces logarithmically so that we are able to perform an exhaustive search to find the best strategy and result in an additional 17.5× saving on the average search time for AlexNet convolutional layers.

Consequently, the first phase of our design space exploration process takes less than 30 seconds to identify a set of high throughput design options instead of hundreds of hours. In the second phase, designs in the set are then synthesized using an Intel SDK for OpenCL Application [INT] to realize the clock frequency. We use the actual frequency to refine the performance estimation for deciding the best systolic array design.

6.6 Implementation and Experiments

6.6.1 An Automation Flow

Code 6.3: Programming Model

```
#pragma ACCEL systolic auto
for(o = 0; o < O; o++)
  for(i = 0; i < I; i++)
    for(c = 0; c < C; c++)
      for(r = 0; r < R; r++)
        for(p = 0; p < K; p++)
          for(q = 0; q < K; q++)
            out[o][r][c] += w[o][i][p][q] * in[i][r+p][c+q];
```

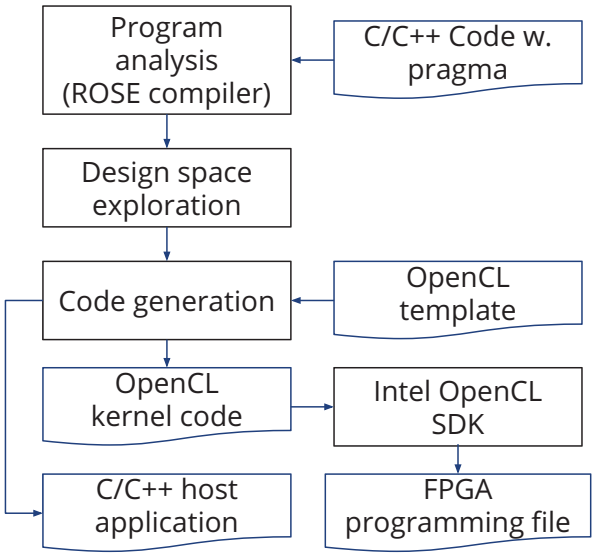


Figure 6.6: The Execution flow

We implement a push-button design flow framework to generate an executable system on FPGAs from a user-written intuitive nested loop in Code 6.3. A user only needs to specify the nested loop with a pragma (we adopt CNN models in this experiment as a case study). Our automation flow shown in Figure 6.6 first analyzes the user program using the ROSE compiler infrastructure [ROS] to obtain necessary information such as iteration domains and data access patterns. Subsequently, we perform design space exploration to identify multiple valid design options with the

highest estimated throughput. The design options are parameterized to instantiate template files, including OpenCL systolic array implementation (kernel), as well as the C/C++ software program (host). Finally, the instantiated OpenCL kernel is synthesized by the Intel FPGA SDK for OpenCL [INT] for the physical implementation.

6.6.2 Experimental Setup

We evaluate our model and systolic array architecture design in Intel's Arria 10 GT 1150 board which contains 1518 hard floating point DSPs. The underlying OpenCL implementation of the systolic array design is synthesized using the Intel SDK 16.0 for OpenCL application [INT]. We only evaluate the single precision floating point data type since the half-precision floating point is not yet supported by current version of the tool set.

We use all convolutional layers from two widely used real-life CNN models, AlexNet [KSH12] and VGG [SZ14], as benchmarks in this experiment. For each model, we generate the design with the optimal performance for all layers according to our two-phase DSE process.

6.6.3 Results and Analysis

In this experiment we use a unified systolic array design configuration for all the layers in each CNN model instead of making an optimal design for each layer, because it has big performance overhead to reprogram the FPGA for different layers. To realize the design with the highest throughput for all layers, we perform the proposed two-phase DSE process to every layer and select the best one to generate the programming file

for on-board evaluation. For example, Figure 6.7 depicts all valid design options of AlexNet layer 5 with a given clock frequency (280 MHz) reported by our framework. The density for each design point represents the throughput where darker means higher. As can be seen in Figure 6.7, high throughput design options may cost moderate BRAM blocks and DSPs due to lower design overhead. This motivates the first phase of our design space exploration. In addition, since the frequency is a given constant value, Figure 6.7 is not able to reflect the impact of different clock frequency.

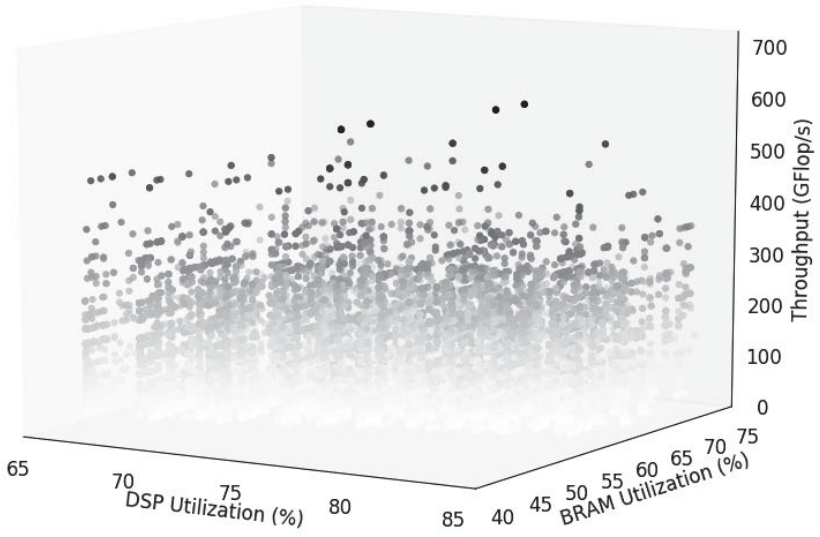


Figure 6.7: Pruned Design Space for AlexNet Layer 5

To deal with the impact of frequency variant, we use the top 14 design options from Figure 6.7 and perform P&R at the same time to realize the actual clock frequency. Figure 6.8 shows a comparison of on-board results against the analytical model of all 14 designs sorted by estimated throughput. As can be seen, our design space exploration identifies 6 designs with the highest estimated throughput. It means that those designs have the same, minimum computation overhead but adopt

Table 6.2: Frequency and Resource Utilization

Model	PE shape	Freq. (MHz)	LUT	DSP	BRAM	FF
AlexNet	(11,14,8)	270.8	57%	81%	45%	40%
VGG	(8,19,8)	252.6	59%	81%	47%	40%

different data reuse strategies. This difference results in different clock frequencies at the P&R stage of the design flow, and it is hard to be predicted in advance. Although the assumed frequency cannot be achieved finally by the design flow, our model perfectly matches the on-board results ($< 2\%$ error on average) by using the real working frequency. This illustrates the accuracy of our analytical model.

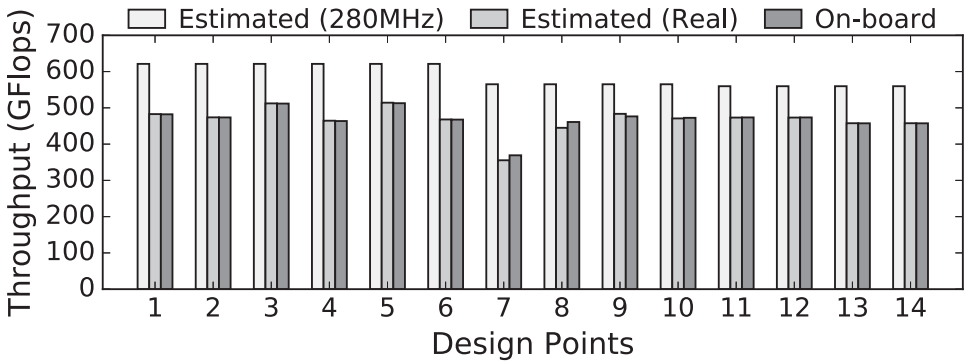


Figure 6.8: Comparison of On-board Data against Analytical Model

Table 6.2 shows the working frequency, resource utilization, and the systolic array design configuration we used for each CNN model as an order of PE row, column and vector. We can see that the designs generated by our framework have high resource utilization and suitable shapes that match most of layers in CNN models.

The performance of the two designs are shown in Table 6.3 and Table 6.4, respec-

Table 6.3: Results of AlexNet

Layer	**1**	**2**	**3**	**4**	**5**	**Avg.**
Thrpt.	123.5	225.0	541.7	541.6	600.0	406.1
DSP Eff.	18.51	33.70	81.03	81.03	90.00	40.32

Table 6.4: Results of VGG

Layer	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Thrpt.	223.86	450.11	600.27	601.69	601.57	602.44	602.44
DSP Eff.	36.36	72.73	96.97	96.97	96.97	96.97	96.97
Layer	**8**	**9**	**10**	**11**	**12**	**13**	**Avg.**
Thrpt.	602.42	602.83	602.83	602.49	602.49	602.49	561.38
DSP Eff.	96.97	96.97	96.97	96.97	96.97	96.97	89.11

tively. We can see that the overall performance for AlexNet [KSH12] and VGG [SZ14] could achieve 406 GFlops and 561 GFlops. Most of the layers of the two CNN models could achieve near-peak performance. For layer 5 of AlexNet and layer 3˜13 of VGG, we achieve more than 600 GFlops throughput as well as high DSP efficiency. However, the throughput and DSP efficiency of AlexNet's layer 1 is much lower than other layers. We conclude two main reasons. First, layer 1 has only 3 large input feature maps which makes the shape of layer 1 quite different from other layers so that a common design for all layers including layer 1 is hard to find. As a result, we folded layer 1 to have more small feature maps to make its configuration more consistent with others.

The second reason is that the kernel size (11) of layer 1 is much larger than other

layers (5 and 3). In order to obtain one design for all layers, our framework chose the data reuse strategy that benefit other layers more. Although the selected data reuse strategy is able to let other layers achieve high throughput, it causes the throughput of layer 1 to be bounded by memory bandwidth.

Table 6.5: Comparison to State-of-the-art Implementations

	[ZFZ16]	[ZFZ16]	[SCD16]	**Ours**	
Prec.	16-bit fixed	32-bit float	8-bit fixed	32-bit float	
FPGA	Xilinx VC709	Xilinx KU060	Altera Stratix-V	Intel Arria 10	
Freq.	150 MHz	200 MHz	120 MHz	270 MHz	253 MHz
CNN	VGG	VGG	AlexNet	AlexNet	VGG
Thrpt.	488 Gops	96 GFlops	137 Gops	406 GFlops	561 GFlops

In addition, the layer 1 of VGG has a lower performance than other layers as well. This is because the layer 1 image row number (16) is inconsistent with other layers, and lead to low DSP utilization of PEs' parallelism and pipelining. However, VGG still has a better overall performance than AlexNet since it has a more regular network shape that shows better scalability for its uniform hardware design.

We finally compare our optimal designs with state-of-the-art studies in Table 6.5. As show in the table, our high throughput designs outperforms all these previous results in convolutional layers. This is not only due to more available DSP resource on the adoption FPGA chip, but also the high frequency that is achieved by our scalable systolic array architecture. We note that the later work, PolySA [CW18], adopts the similar approach that maps a user application to a systolic array archi-

tecture and achieves the similar throughput on VGG [SZ14] (548 GFlops on Xilinx Virtex UltraScale+^{TM} VU9P FPGA). However, since PolySA leverages polyhedral analysis [ZLC13] to represent computation patterns, it covers a more comprehensive application domain than ours.

6.7 Conclusion

In this chapter, we demonstrate that the challenges of design space exploration automation could be easily resolved by sacrificing the generalization. Specifically, by limiting the application domain to convolutional neural networks (CNNs), we are able to design and implement a high-throughput systolic array architecture template on FPGAs. Accordingly, we propose a compiler that maps a user-given CNN in a nested loop to the architecture template to guarantee the throughput. Since we could analytically model the throughput and resource utilization of the architecture with high accuracy (∼95%), the design space exploration strategies are effective and efficient. In addition, we leverage source-to-source code transformation to automate the nested loop to the systolic array mapping process so we do not require human efforts during the DSE. Evaluation results show that our designs for AlexNet and VGG CNN models could on average 406 and 561 GFlops on Intel Arria 10 device.

CHAPTER 7

Conclusion

This dissertation is dedicated to simplify the design optimization process of using HLS for FPGAs. Based on the current challenges and limitations of commercial HLS tools, we design and implement an efficient design space exploration framework with the Merlin compiler [CHP16a]. The framework uses an effective, comprehensive design space representation to create a design space based on Merlin pragma combinations. It then searches for the best design point within the created design space using a proposed algorithm. The algorithm is inspired by gradient descent with several HLS-specific optimization. In addition, to facilitate the search efficiency, we design an algorithm to identify the design bottleneck with the help from Merlin HLS report. As a result, we can identify a subset of design parameters to focus on by running only one design point. The evaluation result shows that the complete DSE framework achieves 93.78% performance to the manual design on geometric mean.

Based on the complete DSE framework, we further support other domain-specific frameworks with domain specific languages. We integrate the DSE framework to S2FA, a Spark-to-FPGA Accelerator framework, to optimize the accelerator design generated from user-written Scale code for Spark [ZCD12]. By considering the parallel patterns from the Spark programming model, the design space is able to be reduced by orders of magnitude. In addition, we also use the DSE framework to

automate a part of HeteroCL [LCH19] scheduling primitives so that users can focus on platform independent loop transformations to eliminate data dependency. We believe that the two showcases demonstrate the usability of the proposed DSE framework for its flexibility and extensiblility, as it can be easily integrated to other compilation frameworks.

Subsequently, we study the trade-off between the generalization and DSE efficiency by proposing composable, parallel and pipeline (CPP) micro-architecture. CPP considers common optimization used by board classes of designs on FPGAs to be a general architecture template. We show in the experiment that we are able to find the best design point under the CPP micro-architecture within an hour using the derived analytical models.

Finally, we present a CNN compilation framework to demonstrate DSE in another architecture template. When the underlying micro-architecture is determined for a specific domain, the development of analytical models for performance and resource utilization is promising and efficient. The design space can also be pruned according to the characteristics of the domain. Combining the above two points, the exhaustive search may be possible and the optimal design point is guaranteed to be found.

[ABC10] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. "Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures." In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2010.

[AKV14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner: An Extensible Framework for Program Autotuning." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[AOC17] Utku Aydonat, Shane OConnell, Davor Capalija, Andrew Ling, and Gordon Chiu. "An OpenCL Deep Learning Accelerator on Arria 10." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[APA] "Aparapi in amd developer website." http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/.

[ARR] "Altera Arria 10 FPGA and SoC." https://www.altera.com/products/fpga/arria-series/arria-10/overview.html.

[ARV03] Arvind. "Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification Invited Talk." In *MEMOCODE*, 2003.

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

[BER99] Dimitri P Bertsekas. *Nonlinear programming*. Athena scientific Belmont, 1999.

[BIR06] Richard S. Bird. "Improving Saddleback Search: A Lesson in Algorithm Design." In *Mathematics of Program Construction*. Springer, 2006.

[BNK98] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.

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

[BVR12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avi?ienis, J. Wawrzynek, and K. Asanovi? "Chisel: Constructing hardware in a Scala embedded language." In *Design Automation Conference (DAC)*, 2012.

[CCK03] Tony F Chan, Jason Cong, Tim Kong, Joseph R Shinnerl, and Kenton Sze. "An enhanced multilevel algorithm for circuit placement." In *International Conference On Computer Aided Design (ICCAD)*, 2003.

[CCL15] Y. T. Chen, J. Cong, J. Lei, and P. Wei. "A Novel High-Throughput Acceleration Engine for Read Alignment." In *Field-Programmable Custom Computing Machines (FCCM)*, 2015.

[CCS05] Tony Chan, Jason Cong, and Kenton Sze. "Multilevel generalized force-directed method for circuit placement." In *International Symposium on Physical Design (ISPD)*, 2005.

[CCW18] Y. Chi, J. Cong, P. Wei, and P. Zhou. "SODA: Stencil with Optimized Dataflow Architecture." *International Conference On Computer Aided Design (ICCAD)*, 2018.

[CDL13] Jaewon Lee Eric Chung, John Davis. "LINQits: Big Data on Little Clients." In *International Symposium on Computer Architecture (ISCA)*, 2013.

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

[CFL18] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. "Understanding performance differences of FPGAs and GPUs." In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

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

[CHP16b] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. "Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers: Invited Paper." In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2016.

[CHZ14] Jason Cong, Muhuan Huang, and Peng Zhang. "Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2014.

[CJL11] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. "Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization." *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2011.

[CLN11] J. Cong, Bin Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment." *IEEE Transaction on Computer-Aided Design (TCAD)*, 2011.

[CMB10] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. "A Programmable Parallel Accelerator for Learning and Classification." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[CMJ18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning." In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[CS13] Jingsheng Jason Cong and Joseph R Shinnerl. *Multilevel optimization in VLSICAD*. Springer, 2013.

[CSJ10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. "A Dynamically Configurable Coprocessor for Convolutional Neural Networks." *ACM SIGARCH Computer Architecture News*, 2010.

[CSP15] Ren Chen, Sruja Siriyal, and Viktor Prasanna. "Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA." In *International Symposium on Field-Programmable Gate Arrays (FPGA*, 2015.

[CSV09] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to derivative-free optimization*, volume 8. Siam, 2009.

[CW18] J. Cong and J. Wang. "PolySA: Polyhedral-Based Systolic Array Auto Compilation." *International Conference On Computer Aided Design (ICCAD)*, 2018.

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

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

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[DGR74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." *IEEE Journal of Solid-State Circuits*, 1974.

[FAP18a] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. "Cluster-Based Heuristic for High Level Synthesis Design Space Exploration." *IEEE Transactions on Emerging Topics in Computing*, 2018.

[FAP18b] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. "Lattice-Traversing Design Space Exploration for High Level Synthesis." In *International Conference on Computer Design (ICCD)*. IEEE, 2018.

[FCS] "Falcon Computing Solutions, Inc." http://falcon-computing.com/.

[FDS10] Álvaro Fialho, Luis Da Costa, Marc Schoenauer, and Michèle Sebag. "Analyzing bandit-based adaptive operator selection mechanisms." In *Ann Math Artif Intell*, 2010.

[FPH09] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. "CNP: An FPGA-based processor for Convolutional Networks." In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2009.

[GBS13] Max Grossman, Mauricio Breternitz, and Vivek Sarkar. "HadoopCL: MapReduce on Distributed Heterogeneous Platforms Through Seamless Integration of Hadoop and OpenCL." In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[GS16] Max Grossman and Vivek Sarkar. "SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform." In *High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[GSZ18] Mohammad-Hossein Golbon-Haghighi, Hadi Saeidi-Manesh, Guifu Zhang, and Yan Zhang. "Pattern synthesis for the cylindrical polarimetric phased array radar (CPPAR)." *Progress In Electromagnetics Research*, 2018.

[HAD] "Apache Hadoop." http://hadoop.apache.org/.

[HCC10] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. "MapCG: Writing Parallel Program Portable Between CPU and GPU." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[HGZ13] Akihiro Hayashi, Max Grossman, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. "Speculative execution of parallel programs with precise exception semantics on gpus." In *International Workshop on Languages and Compilers for Parallel Computing*, 2013.

[HKR07] Martin Holzer, Bastian Knerr, and Markus Rupp. "Design space exploration with evolutionary multi-objective optimisation." In *International Symposium on Industrial Embedded Systems*. IEEE, 2007.

[HLC14] Muhuan Huang, Kevin Lim, and Jason Cong. "A scalable, high-performance customized priority queue." In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2014.

[HME] George Karypis. "hMetis: A Hypergraph Partitioning Package Version 1.5." http://www.cs.umn.edu/karypis.

[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 *ACM Symposium on Cloud Computing (SoCC)*, 2016.

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

[JBC10] A. C. Jacob, J. D. Buhler, and R. D. Chamberlain. "Design of throughput-optimized arrays from recurrence abstractions." In *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2010.

[KE95] J. Kennedy and R. Eberhart. "Particle swarm optimization." In *Proceedings of ICNN'95 - International Conference on Neural Networks*, 1995.

[KP10] Stephen Kou and Jens Palsberg. "From OO to FPGA: Fitting Round Objects into Square Hardware?" In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, 2010.

[KPZ16] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. "Automatic Generation of Efficient Accelerators for Reconfigurable Hardware." In *International Symposium on Computer Architecture (ISCA)*, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In *Advances in Neural Information Processing Systems*, 2012.

[KTR08] Ian Kuon, Russell Tessier, and Jonathan Rose. "FPGA Architecture: Survey and Challenges." *Found. Trends Electron. Des. Autom.*, 2008.

[KUC78] David L. Kuck. *Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.

[KUN88] S. Y. Kung. "VLSI Array Processors." In *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*, 1988.

[KW05] Andrew B Kahng and Qinke Wang. "Implementation and extensibility of an analytic placer." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2005.

[LBC15] J. Liu, S. Bayliss, and G. A. Constantinides. "Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS." In *Field-Programmable Custom Computing Machines (FCCM)*, 2015.

[LC13] Hung-Yi Liu and L. P. Carloni. "On learning-based methods for design-space exploration with High-Level Synthesis." In *Design Automation Conference (DAC)*, 2013.

[LC16] Charles Lo and Paul Chow. "Model-based optimization of high level synthesis directives." In *International Conference on Field-Programmable Logic and Applications (FPL)*, 2016.

[LCH19] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, FPGA '19, 2019.

[LLV07] "LLVM Language Reference Manual.", 2007. http://llvm.org/docs/LangRef.html.

[LWC16] J. Liu, J. Wickerson, and G. A. Constantinides. "Loop Splitting for Efficient Pipelining in High-Level Synthesis." In *Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[MCV17] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks." In *International Conference on Field Programmable Logic and Applications (FPGA)*, 2017.

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

[MLL] "Spark MLlib." http://spark.apache.org/mllib/.

[MLR07] D.J. Newman A. Asuncion. "UCI Machine Learning Repository.", 2007. http://www.ics.uci.edu/$\sim$mlearn/{MLR}epository.html.

[MNI] "The MNIST database." https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets.

[MPA16] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh. "TABLA: A unified template-based framework for accelerating statistical machine learning." In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[MPZ12] Giovanni Mariani, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. "OSCAR: An optimization methodology exploiting spatial correlation in multicore design spaces." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2012.

[MSS14] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. "Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms." In *International Symposium on Computer Architecture (ISCA)*, 2014.

[NW70] Saul B Needleman and Christian D Wunsch. "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of molecular biology*, 1970.

[OMC08] Berkin Ozisikyilmaz, Gokhan Memik, and Alok Choudhary. "Efficient system design space exploration using machine learning techniques." In *Design Automation Conference (DAC)*. ACM, 2008.

[OMP] "OpenMP." http://openmp.org/wp/.

[PAA12] Maciej Pacula, Jason Ansel, Saman Amarasinghe, and Una-May OReilly. "Hyperparameter tuning in bandit-based adaptive operator selection." In *Evostar*. Springer, 2012.

[PBD08] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. "CHiMPS: A C-level compilation flow for hybrid CPU-FPGA architectures." In *International Conference on Field Programmable Logic and Applications (FPGA)*, 2008.

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

[PCC14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." In *International Symposium on Computer Architecture (ISCA)*, 2014.

[PG02] Maurizio Palesi and Tony Givargis. "Multi-objective design space exploration using genetic algorithms." In *Proceedings of the tenth international symposium on Hardware/software codesign*. ACM, 2002.

[PKB16] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. "Generating Configurable Hardware from Parallel Patterns." *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

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

[PSM13] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal. "Memory-centric accelerator design for Convolutional Neural Networks." In *International Conference on Computer Design (ICCD)*, 2013.

[PZS13] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. "Polyhedral-based Data Reuse Optimization for Configurable Computing." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2013.

[QWY16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

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

[RBA13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines." *ACM SIGPLAN Notices*, 2013.

[RM05] Lior Rokach and Oded Maimon. "Top-down induction of decision trees classifiers-a survey." *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, **35**(4):476–487, 2005.

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

[RTS12] Roberto Rodríguez, Esleys Torre, and Juan H Sossa. "Image segmentation via an iterative algorithm of the mean shift filtering for different values of the stopping threshold." *IJIR*, 2012.

[SBC15] Adrian Sampson, James Bornholt, and Luis Ceze. "Hardware-Software Co-Design: Not Just a Cliché." In *Summit on Advances in Programming Languages (SNAPL)*, 2015.

[Keywords: approximation, co-design, architecture, verification.]

[SCD16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

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

[SDX] "Xilinx SDAccel." http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[SFP11] Cristina Silvano, William Fornaciari, Gianluca Palermo, Vittorio Zaccaria, Fabrizio Castro, Marcos Martinez, Sara Bocchio, Roberto Zafalon, Prabhat Avasare, Geert Vanmeerbeeck, et al. "Multicube: Multi-objective design space exploration of multi-core architectures." In *VLSI 2010 Annual Symposium*. Springer, 2011.

[SHA01] Claude E Shannon. "A mathematical theory of communication." In *ACM MC2R*, 2001.

[SJC09] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H. P. Graf. "A Massively Parallel Coprocessor for Convolutional Neural Networks." In *ASAP*, 2009.

[SMC14] Oren Segal, Martin Margala, Sai Rahul Chalamalasetti, and Mitch Wright. "High Level Programming for Heterogeneous Architectures." *ArXiv*, 2014.

[SP97] Rainer Storn and Kenneth Price. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces." *Journal of global optimization*, **11**(4):341–359, 1997.

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

[SSE15] Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann. "HeteroDoop: A MapReduce Programming System for Accelerator Clusters." In *High-Performance Parallel and Distributed Computing (HPDC)*, 2015.

[STW09] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. "Adaptive simulated annealer for high level synthesis design space exploration." In *International Symposium on VLSI Design, Automation and Test (DATE)*. IEEE, 2009.

[SW12a] B Carrion Schafer and Kazutoshi Wakabayashi. "Machine learning predictive modelling high-level synthesis design space exploration." *IET computers & digital techniques*, 2012.

[SW12b] Benjamin Carrion Schafer and Kazutoshi Wakabayashi. "Divide and conquer high-level synthesis design space exploration." *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2012.

[SYZ16] Jincheng Su, Fan Yang, Xuan Zeng, and Dian Zhou. "Efficient Memory Partitioning for Parallel Data Access via Data Reuse." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *arXiv*, 2014.

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

[VB16] S. I. Venieris and C. S. Bouganis. "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs." In *Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[VER10] Sven Verdoolaege. "Isl: An Integer Set Library for the Polyhedral Model." In *International Congress on Mathematical Software*, 2010.

[VIV] "Xilinx Vivado HLS." http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm.

[WC15] Jie Wang and Jason Cong. "Customizable and High Performance Matrix Multiplication Kernel on FPGA." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

[WDC] Hannes Muhleisen. "Web Data Commons Extracting Structured Data from Two Large Web Corpora.".

[WEI18] Peng Wei. *Enabling Customized Computing in Datacenters: from Accelerator Design to System Integration*. PhD thesis, UCLA, 2018.

[WHZ16] Z. Wang, B. He, W. Zhang, and S. Jiang. "A performance analysis framework for optimizing OpenCL applications on FPGAs." In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

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

[WZH16] Z. Wang, S. Zhang, B. He, and W. Zhang. "Melia: A MapReduce Framework on OpenCL-based FPGAs." *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2016.

[XLZ17] Chang Xu, Gai Liu, Ritchie Zhao, Stephen Yang, Guojie Luo, and Zhiru Zhang. "A Parallel Bandit-Based Approach for Autotuning FPGA Compilation." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[XPZ15] Sotirios Xydis, Gianluca Palermo, Vittorio Zaccaria, and Cristina Silvano. "SPIRIT: Spectral-Aware pareto iterative refinement optimization for supervised high-level synthesis." *IEEE Transaction on Computer-Aided Design (TCAD)*, 2015.

[XUL] "Xilinx Ultrascale Architecture." https://www.xilinx.com/products/technology/ultrascale.html.

[ZCD12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[ZCF10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[ZFS17] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. "COMBA: a comprehensive model-based analysis framework for high level synthesis of real applications." In *International Conference On Computer Aided Design (ICCAD)*, 2017.

[ZFZ16] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks." In *International Conference On Computer Aided Design (ICCAD)*, 2016.

[ZHX15] Peng Zhang, Muhuan Huang, Bingjun Xiao, Hui Huang, and Jason Cong. "CMOST: A System-level FPGA Compilation Framework." In *Design Automation Conference (DAC)*, 2015.

[ZKM12] Marcela Zuluaga, Andreas Krause, Peter Milder, and Markus Püschel. "Smart design space sampling to predict pareto-optimal solutions." In *ACM SIGPLAN Notices*. ACM, 2012.

[ZLC13] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. "Improving Polyhedral Code Generation for High-level Synthesis." In *IEEE/ACM/IFIP International Conference on Hardware/-Software Codesign and System Synthesis*, 2013.

[ZLS15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks." In *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.

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

[ZPL16] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. "Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators." In *Design Automation Conference (DAC)*, 2016.

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

[ZVL14] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smail Niar. "Design space exploration of multiple loops on FPGAs using high level synthesis." In *International Conference on Computer Design (ICCD)*, 2014.