UNIVERSITY OF CALIFORNIA, SAN DIEGO

Learning Optimizations for Hardware Accelerated Designs

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

in

Computer Science

by

Pingfan Meng

2016

DEDICATION

To Audrey, Fangang and Jianping.

EPIGRAPH

Any sufficiently advanced technology is indistinguishable from magic.

Arthur C. Clarke

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTINGS

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Ryan Kastner for his guidance and support as my advisor. Due to his encouragement and support, I had the opportunity to be recognized by the international level research community at several venues. This Ph.D. program under his supervision was a very enjoyable process of exploring the unknown in the computer science world.

I would also like to acknowledge my committee members. Professor Scott Baden attended multiple talks of mine and provided some valuable input to the primary part of my research. Professor Steven Swanson and Professor Michael Taylor directed me to several useful materials during my thesis proposal, which eventually provided me major inspirations of this work. Professor Andrew McCulloch's theory provides the foundation of my very first research project and publication. I would like to thank him for this great project idea which initiated my Ph.D. research.

None of this would have been possible without many collaborators. I would like to acknowledge Dr. Matthew Jacobsen. He provided me not only technical help but also insights in my research. I would also like to acknowledge Dr. Motoki Kimura, Alric Althoff, and Quentin Gautier for their extraordinary work and help in my research projects.

The majority of the work herein appears in both academic journals and conferences. Chapter 2 contains material printed in "Hardware Accelerated Alignment Algorithm for Optical Labeled Genomes", Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, the ACM Transactions on Reconfigurable Technology and Systems 2016 and in "Hardware Accelerated Novel Optical De Novo Assembly for Large-Scale Genomes", Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, International Conference on Field-Programmable

Technology (FPT), December 2014. The dissertation author was the primary investigator and author of these papers.

Chapter 3 contains material printed in “FPGA-GPU-CPU Heterogeneous Architecture for Realtime Cardiac Physiological Optical Mapping”, Pingfan Meng, Matthew Jacobsen, and Ryan Kastner, the International Conference on Field-Programmable Technology (FPT) 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 4 contains material printed in “Adaptive Threshold Non-Pareto Elimination: Re-thinking Machine Learning for System Level Design Space Exploration on FPGAs”, Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner, the Conference on Design Automation and Test in Europe (DATE) 2016. The chapter also contains material that was an improvement of the results in the conference paper. The dissertation author was the primary investigator and author of this paper.

Chapter 5 contains material printed in the submission of “Can One Use GPU Performance Data for FPGA Design Space Exploration?”, Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner, to International Conference On Computer Aided Design (ICCAD), 2016. The dissertation author was the primary investigator and author of this paper.

VITA

2009	Bachelor of Science, Shanghai Jiaotong University
2011	Master of Science, University of California, San Diego
2016	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

“Can One Use GPU Performance Data for FPGA Design Space Exploration?”, Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner, International Conference On Computer Aided Design (ICCAD), 2016 (under review)

“Adaptive Threshold Non-Pareto Elimination: Re-thinking Machine Learning for System Level Design Space Exploration on FPGAs”, Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner, Design Automation and Test in Europe (DATE), March 2016

“Hardware Accelerated Alignment Algorithm for Optical Labeled Genomes”, Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, ACM Transactions on Reconfigurable Technology and Systems (TRETS), in press

“Real-time 3D Reconstruction for FPGAs: A Case Study for Evaluating the Performance, Area, and Programmability Trade-offs of the Altera OpenCL”, Quentin Gautier, Alexandria Shearer, Janarbek Matai, Dustin Richmond, Pingfan Meng, and Ryan Kastner, International Conference on Field-Programmable Technology (FPT), December 2014

“Hardware Accelerated Novel Optical De Novo Assembly for Large-Scale Genomes”, Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, International Conference on Field Programmable Logic and Applications (FPL), September 2014

“FPGA Accelerated Online Boosting for Multi-Target Tracking”, Matthew Jacobsen, Pingfan Meng, Siddarth Sampangi, Ryan Kastner, IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2014

“GPU Accelerated Post-Processing for Multifrequency Biplanar Interferometric Imaging”, Pingfan Meng, George R. Cutter Jr., Ryan Kastner, David A. Demer, Oceans, September 2013

“A Hardware Accelerated Approach for Imaging Flow Cytometry”, Dajung Lee, Pingfan Meng, Matthew Jacobsen, Henry Tse, Dino Di Carlo and Ryan Kastner, International Conference on Field Programmable Logic and Applications (FPL), September 2013

“FPGA-GPU-CPU Heterogeneous Architecture for Realtime Cardiac Physiological Optical Mapping”, Pingfan Meng, Matthew Jacobsen, and Ryan Kastner, International Conference on Field-Programmable Technology (FPT), December 2012

“A Hardware in the Loop Wireless Channel Emulator for Software Defined Radio”, Janarbek Matai, Pingfan Meng, Linjuan Wu, Ryan Kastner, International Conference on Field-Programmable Technology (FPT), December 2012

“GPU Acceleration of Optical Mapping Algorithm for Cardiac Electrophysiology”, Pingfan Meng, Ali Irturk, Ryan Kastner, Andrew McCulloch, Jeffrey Omens and Adam Wright, International conference of the IEEE Engineering in Medicine & Biology Society (EMBS), August 2012

“Strategies for Implementing Hardware-Assisted High-Throughput Cellular Image Analysis”, Henry T. Tse, Pingfan Meng, Daniel R. Gossett, Ali Irturk, Ryan Kastner, and Dino Di Carlo, Journal of Laboratory Automation, December 2011 vol. 16 no. 6 422-430

ABSTRACT OF THE DISSERTATION

Learning Optimizations for Hardware Accelerated Designs

by

Pingfan Meng

Doctor of Philosophy in Computer Science

University of California, San Diego, 2016

Professor Ryan Kastner, Chair

Many emerging applications require hardware acceleration due to their growing computational intensities. These accelerated designs use heterogeneous hardware, such as GPUs, FPGAs and multi-core CPUs to process the intensive computations at a higher rate. The first part of this work provides two paradigms of hardware accelerated biomedical applications. These paradigms achieved 115X and 273X speedups respectively.

Developing these paradigms taught us that, in order to efficiently utilize the heterogeneous accelerators, the designer needs to carefully investigate which device is the most suitable accelerator for a particular computing task. In addition, the designer needs to effectively optimize the computations to fully exploit the computing power of the selected accelerator. This process is called design space exploration (DSE). Heterogeneous DSE requires multiple programming skills for these different types of devices.

In recent years, there is a trend to use one unified programming language for multiple heterogeneous devices. The SDKs and hardware synthesis tools have enabled OpenCL as one unified language to program heterogeneous devices including GPUs, FPGAs, and multi-core CPUs. However, one major bottleneck for DSE still exists. In contrast to GPU and CPU OpenCL code compilation, which only consumes several milliseconds, implementing OpenCL designs on a FPGA requires hours of compilation time. Moreover, merely tuning a few programming parameters in the OpenCL code will result in an abundance of possible designs. Implementing all these designs requires months of compilation time. Exploring the FPGA design space with brute force is therefore impractical.

The second part of this work addresses this issue by providing a machine learning approach for automatic DSE. This machine learning approach automatically identifies the optimal designs by learning from a few training samples. In comparison with other state-of-the-art machine learning frameworks, this approach reduces the amount of hardware compilations by 3.28X, which is equivalent to hundreds of compute hours. This work also provides a data mining method that enables the machine to automatically use the estimation data to replace the time consuming end-to-end FPGA training samples for DSE. Mining these estimation data further reduces the amount of hardware compilations by 1.26X.

Chapter 1

Introduction

1.1 Overview

1.1.1 Optimization of Hardware Accelerated Designs

Numerous emerging applications from a variety of domains require computing systems that are capable of processing intensive data at a high rate. However, the general purpose Central Processing Units (CPU) are usually incapable of providing the required computing performance for these applications. Therefore, in order to meet the performance requirement, hardware acceleration is highly desirable. Several applications that used hardware acceleration are enumerated as follows: (1) many real-time detection and tracking tasks [1, 2, 3, 4, 5, 6] require accelerated computer vision systems; (2) bioinformatics applications [7, 8, 9, 10, 11, 12] require accelerated computing systems to process lengthy genetic sequences; (3) some database applications [13, 14, 15, 16, 17] require hardware accelerators to achieve higher throughput; (4) multiple cryptographic applications need hardware accelerators to reduce the compute time of those complex encryption/decryption operations [18, 19, 20, 21]; (5) many biomedical systems [22, 23, 24, 25, 26] require hardware accelerators to achieve the real-time processing rate.

These applications use heterogeneous hardware to accelerate the computationally intensive processes. Graphics Processing Units (GPU) and Field-Programmable Gate

Arrays (FPGA) are two types of typical heterogeneous hardware accelerators. Both the GPU and FPGA accelerate computations by utilizing parallelism. The GPU architecture contains many parallel cores that run in the Single Instruction, Multiple Data (SIMD) fashion. The FPGA provides a customizable architecture where the designers can implement parallelism such as pipelines and replicated computing units.

In order to achieve the high performance goal, the hardware accelerated designs need to be carefully optimized. The essence of this optimization task is to restructure and tune the code in a particular way to map the application onto those parallel hardware features efficiently. On the GPU, the programmer needs to map the original sequential software code to the SIMD architecture by assigning the computation tasks to a massive group of threads [27, 28, 29, 30]. The programmer also needs to carefully manage the memory access pattern of the code to efficiently fit the process on the memory hierarchy of the architecture. On the FPGA, the designer is responsible for customizing the architecture to fulfill the computation task. Due to the hardware design nature of the FPGA implementation, it requires the designer to optimize the architecture on a low level. For example, the designer needs to make decisions on the depth of a pipeline or the bit-width of a register [31, 32, 33, 34, 35, 36, 37].

This dissertation presents several research results on the problem of how to effectively explore the optimizations for hardware accelerated designs. In the first part, I will provide two hardware acceleration paradigms in two emerging application domains: one high throughput genetic sequencing system and one cardiac physiology image processing system. The second part of this dissertation presents a methodology using machine learning to automatically explore the optimization strategies for hardware accelerated designs.

1.1.2 Hardware Accelerated Paradigms

In Chapter 2, I will present a paradigm of the hardware accelerated DNA *de novo* assembly. *De novo* assembly is a widely used methodology in bioinformatics. However, the conventional short-read *de novo* assembly is incapable of reliably reconstructing the large-scale structures of human genomes [38] due to the ambiguity caused by repetition of the nucleobases (i.e. “A”,“T”,“G” and “C”). In recent years, a novel assembly technology has been proposed. This new technology aligns the DNA strings based on the uniquely identifiable patterns of optical labels instead of the nucleobases. In contrast to the four letter nucleobases, these optical patterns are arbitrary floating point numbers. For this reason, these optical patterns are unlikely to have repetitions. Thus, this enables reliable large-scale *de novo* assembly. Despite its advantage in large-scale genome analysis, this new technology requires a more computationally intensive alignment algorithm than its conventional counterpart. For example, the run-time of reconstructing a human genome is on the order of $10,000$ hours on a sequential CPU. Therefore, in order to practically apply this new technology in genome research, accelerated approaches are desirable. The results of this work are three different accelerated approaches, multi-core CPU, GPU and FPGA. Against the sequential software baseline, the multi-core CPU design achieved a $8.4\times$ speedup while the GPU and FPGA designs achieved $13.6\times$ and $115\times$ speedups respectively. This work also provides the details of the manual design space exploration for this application on these three different devices. In addition, this work compares these devices in performance, optimization techniques, prices and design efforts.

In Chapter 3, I will present a FPGA-GPU-CPU heterogeneous hardware architecture for a biomedical image process application – real-time optical mapping. Real-time optical mapping is a technique that can be used in cardiac disease study and treatment technology development to obtain accurate and comprehensive electrical activity over the entire heart [39, 40]. It provides a dense spatial electrophysiology. Each pixel essentially plays the role of a probe on that location of the heart. However, the high throughput nature of the computation causes significant challenges in implementing a real-time optical mapping algorithm. This is exacerbated by high frame rate video (order of 1000 fps) for many medical applications. Accelerating optical mapping technologies using multiple CPU cores yields modest improvements, but still only performs at 3.66 frames per second (fps). A highly tuned GPU implementation achieves 578 fps. A FPGA-only implementation is infeasible due to ultra-large intermediate data arrays generated by the optical mapping algorithm. The main result of this work is the real-time system accelerated by a FPGA-GPU-CPU architecture running at 1024 fps. This represents a 273× speedup over a multi-core CPU implementation.

Developing these paradigms taught us that the heterogeneous design space exploration (DSE) is extremely difficult. In many cases, each type of device requires a unique programming or design skill. Thus, in order to explore multiple acceleration devices for a given application, the designer needs to implement the application using multiple different languages. For example, in order to implement the paradigms described in Chapters 2 and 3, we used multiple programming languages such as C++, CUDA, Verilog and OpenMP. Using some of these design languages such as Verilog is especially tedious due to its low level nature. According to our firsthand development experience, it takes months of development time to implement one application on all these different devices.

In recent years, there is a trend of using one unified language to program all types of heterogeneous devices. The improvement of High Level Synthesis (HLS) tools allows the designer to use software language to program FPGAs. The state-of-the-art HLS tools are capable of automatically converting programs written in high level languages, such as C, CUDA and OpenCL [41, 42, 43, 44, 45, 46, 47, 48], to FPGA accelerated implementations. The OpenCL-to-FPGA synthesis tool is especially appealing since

OpenCL software SDKs have already been available for CPUs and GPUs. This means the designer can use one unified language to program all three heterogeneous devices. In order to achieve a high performance FPGA design, the programmer needs to tune multiple parameters and primitives in the high level code [49, 50, 51, 52, 53, 54]. Tuning these parameters and primitives creates a great number of possible designs. The HLS compilation process is time consuming due to the Place and Route (PnR) stage of the tool chain. PnR assigns the application functionalities onto the actual hardware logic and connects them physically. Multiple subproblems in this PnR task have been proven to be NP-hard [55, 56]. This process usually consumes multiple hours for a typical design. Thus, the brute force DSE method that implements and evaluates all these possible tunings may consume months of compilation time. This issue motivates the second part of this dissertation - machine learning automation.

1.1.3 Automatic Machine Learning Design Space Exploration

As discussed in Section 1.1.2, SDKs and hardware synthesis tools have provided a push forward to enable OpenCL as a unified language to program heterogeneous platforms such as GPUs, FPGAs, and multi-core processors. However, one major bottleneck of the system level OpenCL-to-FPGA design tools is their extremely time consuming synthesis process (including place and route). The design space for a typical OpenCL application contains thousands of possible designs even when considering a small number of design space parameters. It costs months of compute time to synthesize all these possible designs into end-to-end FPGA implementations. Thus, it is impractical to explore a large amount of possible designs by implementing and evaluating them.

One direction to address this issue is to build analytical models to describe the design objectives, e.g. performance and hardware resource utilization. Using these models, one can evaluate the designs without implementing them. There exists many analytical models [57, 58, 59, 60, 61] for GPU architectures. However, in contrast to GPUs, the reconfigurable FPGA devices do not have fixed architectural features. Therefore, it is unlikely to build an analytical model to describe all FPGA designs. There only exist a few FPGA analytical models for particular domains of designs [62, 63]. The effectiveness of these models is moderate since the actual FPGA performance depends on many low level factors and design tool chain operations. These factors are usually nonlinear and sometimes even random (e.g. the random numbers used in the place and route stage). Therefore, it is extremely difficult to analytically describe the final output of a high level design. Moreover, these models only target several specific types of operations. They are not available for most applications on the FPGA.

Another direction is to build a machine to automatically learn the design space of each particular application by a small number of sampled training designs. However, most of the existing machine learning approaches focus on how to predict the model of the design space accurately. This is not the goal of optimization. The actual goal of optimization is to find the “good” designs. It is unnecessary to model the entire design space since it is useless to obtain the performances of those “bad” designs. Thus, the existing machine learning approaches are not suitable for FPGA DSE problems. To address this issue, in Chapter 4, we propose a novel machine learning approach - Adaptive Threshold Non-Pareto Elimination (ATNE). Instead of focusing on regression accuracy improvement, ATNE focuses on understanding and adapting to the inaccuracy. ATNE provides a Pareto identification threshold that adapts to the estimated inaccuracy of the regressor. This approach results in a more efficient DSE. For the same prediction quality, ATNE reduces the amount of required compilations by $3.28\times$ (hundreds of compilation hours) against the other state of the art machine learning frameworks [64, 65] for FPGA DSE. In addition, ATNE is capable of identifying the Pareto designs for certain difficult design spaces which the other existing frameworks are incapable of exploring effectively.

The ATNE approach uses the real FPGA implementations as the training data. Obtaining these training data still consumes a great amount of time due to the PnR stage in compilation. Is there any method to further reduce the time consumption? To answer this question, we investigated how to use pre-Place and Route (pre-PnR) estimation data for FPGA DSE in Chapter 5. We obtained one type of estimation data by running those FPGA OpenCL designs on the GPU. Although the GPU and FPGA are two different compute platforms, they are both dictated by the OpenCL programming model. The OpenCL design on the FPGA contains many GPU-like architectural features such as SIMD, compute unit parallelism, and local/global memory hierarchy. It is therefore reasonable to believe that if one uses OpenCL programming language on both devices, the GPU and FPGA design spaces will share a certain level of similarity. In contrast to implementing a design on the FPGA, it is significantly cheaper to compile a GPU program (milliseconds compilation on the GPU vs. hours of synthesis time on the FPGA). For these reasons, it is possible to use the GPU results to replace the FPGA training data for DSE.

We also obtained other types of estimation data from the pre-PnR stages of the OpenCL-to-FPGA compilation tool chain. These data contain some high level information of the application. Therefore, they can be used to roughly estimate the final FPGA implementation. Generating these estimations requires significantly less time than obtaining the real FPGA measurements does. Therefore, using these estimation data to partially replace the real FPGA training data could reduce the time consumption further. The estimations are not identical to actual FPGA measurements, i.e., there exists some level of inaccuracy (potentially substantial differences) in the estimation data. For this reason, an intelligent data mining approach is proposed in Chapter 5 to effectively extract useful information from the estimation data to improve FPGA DSE. We implemented this approach and evaluated it with 10 end-to-end FPGA benchmarks. The evaluation results indicate that our approach effectively reduces the sampling complexity by 1.26×, which reduces the DSE by hundreds of compute hours.

1.2 Contributions

The primary contributions of this work are enumerated as follows.

A hardware accelerated optical labeled DNA sequence alignment system: This work is the first attempt to accelerate the large-scale genome assembly on hardware. The work provides an end-to-end FPGA accelerated design and a GPU accelerated implementation. In addition, this work provides a comparison and design space exploration of the multi-core CPU, GPU and FPGA. This work has been published and is described in Chapter 2.

A FPGA-GPU-CPU heterogeneous architecture for real-time optical cardio-electrophysiology: This includes a FPGA-GPU-CPU heterogeneous hardware accelerated system that provides real-time optical mapping. This also includes an analysis of how to efficiently partition and assign the application to different hardware accelerators. This work has been published and is described in Chapter 3.

A robust machine learning approach for OpenCL-to-FPGA Design Space Exploration: This includes a mathematical model to investigate how the machine learning technology should be applied in the system level FPGA DSE task. This work also provides a novel approach which reduces the synthesis complexity by 3.28× compared with other state of the art approaches. This work used 10 end-to-end OpenCL applications (real performance data from applications running on the FPGA, not just reports from the synthesis tool) to verify the effectiveness of the proposed approach. This work has been published and is described in Chapter 4.

A data mining approach that uses pre-Place and Route estimation data for OpenCL-to-FPGA Design Space Exploration: This work investigated the question of whether one can use estimation data for FPGA DSE by providing empirical studies on several possible approaches. The output of this work is a data-mining approach that effectively extracts the useful information from the estimated results for FPGA DSE.

1.3 Dissertation Outline

The dissertation is organized as two parts. Chapters 2 and 3 in Part I present the two manually designed hardware acceleration paradigms. Chapter 2 details the first paradigm: hardware accelerated approaches for a DNA sequence alignment algorithm. Chapter 3 focuses on the details of the second paradigm: a FPGA-GPU-CPU heterogeneous system for real-time cardiac physiological image processing.

Chapters 4 and 5 in Part II focus on the investigation of how to use machine learning to automate OpenCL-to-FPGA design space exploration. Chapter 4 describes a novel machine learning approach for the OpenCL-to-FPGA DSE. Chapter 5 discusses the question of whether one can use the pre-PnR estimation data to further improve the efficiency of the DSE.

Chapter 6 concludes this work. Chapter 7 describes several possible future directions of this topic.

Part I

Manual Design Approaches

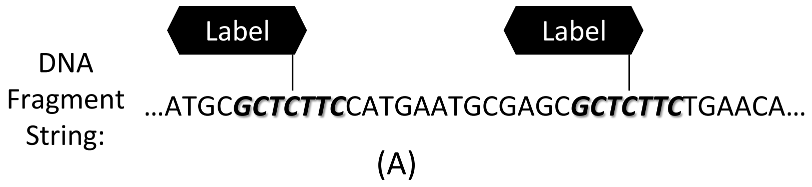
Chapter 2

Hardware Acceleration Approaches for Optical Labeled DNA Alignment

2.1 Background and Motivation

The ability to construct *de novo* assemblies is widely pursued for medical and research purposes. These *de novo* assemblies are especially invaluable in the studies of structural variations of genomes [66]. However, the conventional short-read technology based *de novo* assemblies provide structural information only on a micro-scale ($< 1,000$ bases per fragment). They are not capable of reconstructing the large-scale structures of human genomes [38]. This is due to the fact that using the short-read based assembly leads to ambiguity when these large-scale ($> 100,000$ bases per fragment) genomes have frequent structural repetitions (typical medium to large genomes contain 40 - 85% repetitive sequences [67]).

In recent years, research has shown that a novel optical label based technology is able to overcome this limitation of the short-read technology [68]. This novel technology fluorescently labels the DNA molecule strings at the locations where a specific nucleobase combination appears (e.g. label wherever the combination GCTCTTC appears, as demonstrated in Fig. 2.1(A)). Then the labeled DNA molecules are linearized by being passed through a nanochannel device. These linearized strings with labels are imaged by a CCD camera as demonstrated in Fig. 2.1(B). In the image field, on each string, the physical distances between every two labels are measured and collected. This process results in a uniquely identifiable sequence-specific pattern of labels to be used for *de novo* assembly. As opposed to the four letters, these arbitrary physical distances are unlikely to contain structural repetitions. Therefore, this optical method enables the reconstruction of the large-scale genomes for modern bioinformatic studies. In genomic studies, $N50$ is a widely used metric to measure the ability of a technology to assemble large-scale structures. Research results show that this novel optical assembly enhances the $N50$ by two orders of magnitude compared to the short-read assembly [38].



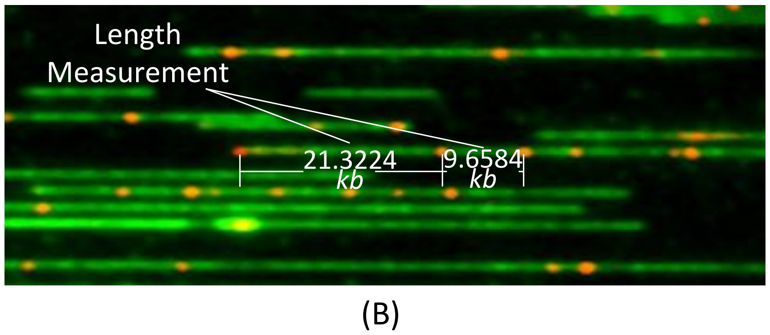


Figure 2.1. Demonstration of the optical labeling process. (A) Fluorescent labels attached to “GCTCTTC”. (B) The real image field of labeled DNA fragments from a microscopy CCD camera. The strings are linearized DNA fragments. The glowing dots are fluorescent labels. The numbers in kilo-bases(*kb*) are examples of physical distance measurement between labels.

The task of the *de novo* assembly is reconstructing the genome from a set of DNA fragments. The most computationally intensive part of this task is the algorithm that aligns every pair from the DNA fragment set. This pair-wise alignment algorithm for the optical assembly is fundamentally different from the short-read alignment. In the conventional short-read based process, as depicted in Fig. 2.2 (A), the alignment algorithm is applied on the strings with “A”,“C”,“G” or “T” DNA nucleobase letters. As opposed to the short-read letters, the new optical method aligns the locations of the fluorescent labels on the strings shown in Fig. 2.2 (B). Aligning these arbitrary numbers obtained from a human genome takes nearly $10,000$ hours on a sequential CPU. Moreover, research [69] has shown that the resolution of the optical label method can be further enhanced by adding multiple types (colors) of labels.

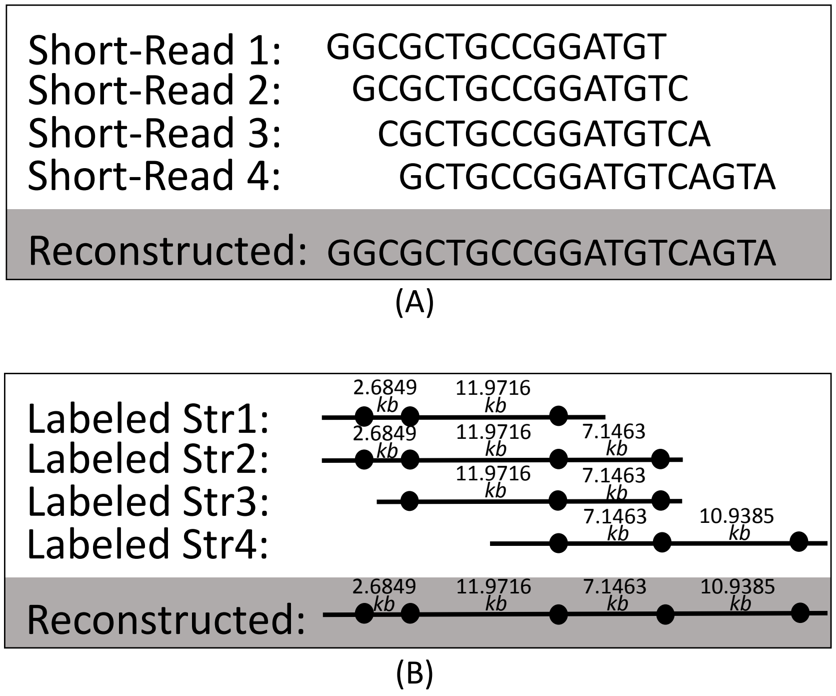

Short-Read 1: GGCGCTGCCGGATGT
Short-Read 2: GCGCTGCCGGATGTC
Short-Read 3: CGCTGCCGGATGTCA
Short-Read 4: GCTGCCGGATGTCAGTA
Reconstructed: GGCGCTGCCGGATGTCAGTA
(A)
Labeled Str1:
Labeled Str2:
Labeled Str3:
Labeled Str4:
Reconstructed:
2.6849 kb
11.9716 kb
7.1463 kb
10.9385 kb
(B)

Figure 2.2. Comparison of the conventional and the novel *de novo* assembly methods. (A) Alignment process in the conventional short-read based method. (B) Alignment process in the novel optical label based method. Each dot represents a fluorescent label.

Therefore, accelerating this alignment algorithm is desired not only for the purpose of shortening the process time but also for enabling this optical based technology in genome studies that require high resolutions.

In this chapter, we present three accelerated approaches for the optical labeled

DNA fragment alignment using multi-thread CPU, GPU and FPGA. These designs are compared against a single thread sequential CPU implementation.

The rest of the chapter is organized as follows. We discuss related work in Section 2.2. We describe the alignment algorithm in Section 2.3. This is followed by descriptions of the accelerated designs in Section 2.4. Experimental performance results are provided in Section 2.5. We compare the hardware accelerators in Section 2.6. Section 2.7 summarizes this chapter.

2.2 Related Work

Multiple accelerated approaches for short-read assembly have been proposed in recent years. Olson *et al.* have proposed a multi-FPGA accelerated genome assembly for short-reads in [7]. They accelerated the alignment algorithm on the FPGAs for the reference guided genome assembly with 250× and 31× speedups reported against the software implementations BFAST and Bowtie respectively. Varma *et al.* have presented a FPGA accelerated *de novo* assembly for short-reads in [8]. They chose to accelerate a pre-processing algorithm on the FPGA to reduce the short-read data for the CPU assembly algorithm. They reported a 13× speedup over the software. They also proposed an improved FPGA implementation exploiting the hard embedded blocks such as BRAMs and DSPs in [70]. Attempts have also been made to accelerate genome assembly on GPUs. Aji *et al.* have proposed a GPU accelerated approach for short-read assembly in [71]. They reported a 9.6× speedup. Liu *et al.* proposed a GPU accelerated DNA assembly tool - SOAP3 [72] which achieves 20× speedup over Bowtie.

Although these approaches have improved the performance of the short-read assembly significantly, they are limited to micro-scale genomes. There is still no high performance solution for large-scale genome structure analysis. Our implementations provide an accelerated solution for this large-scale genome task.

Our implementations are fundamentally different from these previous efforts because they employ the novel optical label based genome assembly. Our accelerated designs differ from the previous short-read approaches in two ways: 1) the data in the optical method requires more precision bits than conventional four letters (A,C,G,T) do; 2) the physical label locations require a different alignment algorithm [73] from the traditional Smith-Waterman. Most short-read methods employed the traditional Smith-Waterman algorithm which computes each score matrix element from its three immediately adjacent elements. The algorithm in our optical label based method computes each element from a 4×4 area as demonstrated in Fig. 2.5. These differences not only increase the computational intensity but also require a different hardware parallel strategy from the ones proposed in these previous short-read based works. To the best of our knowledge, our implementations are the first attempt to accelerate the large-scale genome assembly using GPUs and FPGAs.

2.3 Alignment Algorithm

Our goal is to align every pair of floating point number arrays which represent the physical distances of the optical labels on the DNA fragments. As shown in Fig. 2.3, for arrays X and Y, we decide whether the alignment (X_j aligned to Y_i) is valid based on three evaluations. Firstly, as shown in Fig. 2.3 (A), we need to evaluate the similarity between X_j and Y_i, which is intuitive. Secondly, as depicted in Fig. 2.3 (B), we need to evaluate the boundary offset penalty. When X_j is aligned to Y_i, the leftmost ends of X and Y may create an offset. Large offsets produce unwanted gaps in the DNA assembly. A valid alignment should have minimum offset. The third evaluation is to calculate the similarities between X_j's neighbors and Y_i's neighbors as demonstrated in 2.3 (C). The necessity of this evaluation is also intuitive. Even if X_j is very similar to Y_i, the alignment is not valid if the other elements of the two arrays are dissimilar.

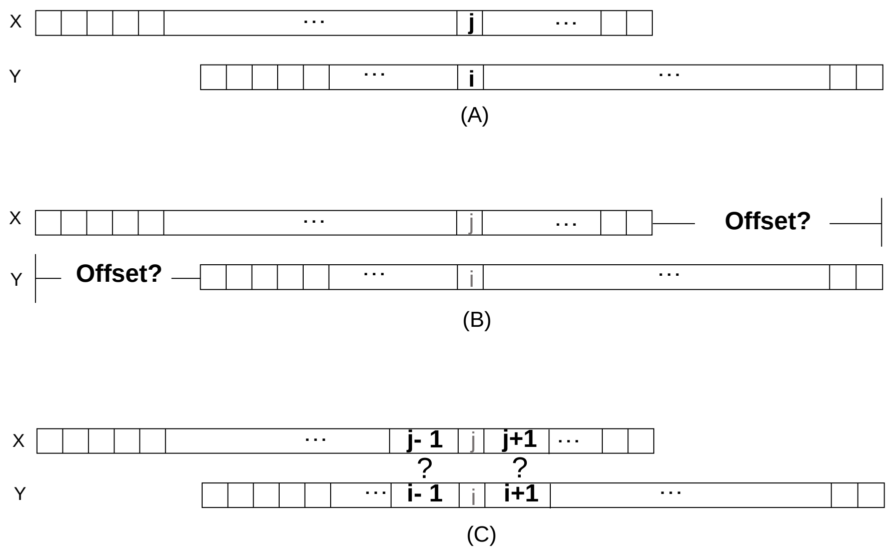


Figure 2.3. Array alignment: assume X_j aligned to Y_i, (A) evaluate the similarity between X_j and Y_i; (B) evaluate the boundary offset penalty when X_j aligned to Y_i; (C) evaluate the similarities between X_j's neighbors and Y_i's neighbors.

These intuitive evaluations of the alignment are realized by a dynamic programming method specifically modified for the optical DNA analysis by Valouev [73]. The overall flow diagram of the algorithm is demonstrated in Fig. 2.4. The algorithm aligns two arrays X and Y of optical label positions by computing a likelihood score matrix and finding the maximum within this matrix. Each score represents the likelihood of a possible alignment between X and Y. Assuming the sizes of the input arrays are M and N, the algorithm computes a $M \times N$ score matrix as depicted in Fig. 2.5. The computation of each element in the matrix requires local scores. The black square in the figure shows an example of a local score. Those elements near the edges, shown as the grey regions in the figure, also require boundary scores. Thus, the alignment algorithm consists of three steps: 1) compute the boundary scores as described in Algorithm 2.1; 2) compute the local scores as described in Algorithm 2.2; 3) find the best score and its correspondent (i, j) in the score matrix as shown in lines 10 - 12 of Algorithm 2.2. If the best score passes the threshold, then we find an alignment between X and Y with X_j

aligned to Y_i using a trace-back operation. In our hardware accelerated approaches, we keep the trace-back operation on the host PC. We therefore only describe the best score computation in detail as follows.

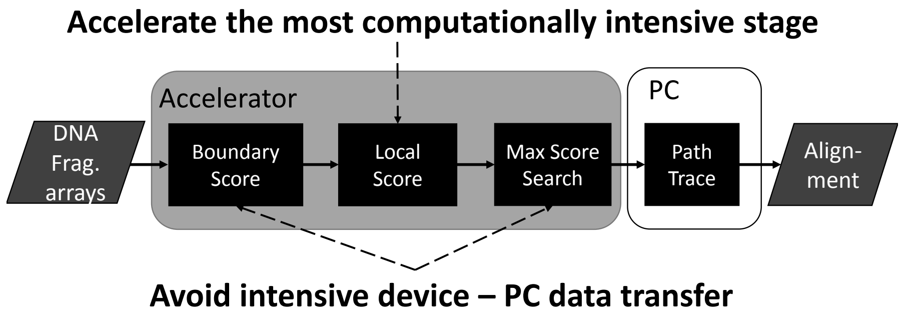

Accelerate the most computationally intensive stage
Accelerator
PC
DNA Frag. arrays
Boundary Score
Local Score
Max Score Search
Path Trace
Align-ment
Avoid intensive device – PC data transfer

Figure 2.4. Overall algorithm and its hardware partitioning. We accelerate the local score stage due to its computational intensity. In our partitioning, we also assign the boundary score and max score search stages to the accelerator to avoid intensive device-PC data communication.

The computation of the boundary scores is described in Algorithm 2.1. In the algorithm, to compute a boundary score element located at (i, j), we firstly compute its leftmost offset $Lx_{i,j}$ or $Ly_{i,j}$ as shown in lines 13 - 22. Then we compute an "end" likelihood and several mixed likelihoods as shown in lines 23 - 33. We choose the maximum among these likelihoods to be the boundary score for this position. This process is iterated, as shown in lines 4 and 12, to produce the boundary scores for the top 4 rows and the leftmost 4 columns of the score matrix. An identical boundary score algorithm is also applied on the rightmost offsets of the input arrays to fill the bottom 4 rows and the rightmost 4 columns of the score matrix. These boundary score locations are visualized in Fig. 2.5.

We compute a local score to represent the similarity between X_j and Y_i as well as the similarities between X_j's neighbors and Y_i's neighbors. In Algorithm 2.2, to compute each local score $score_{i,j}$, we generate 16 score candidates correspondent to its upper-left 4×4 neighbors (refer to Algorithm 2.2 lines 5 - 9). Each of the 16 candidates is computed by adding a local likelihood (this represents the similarity between X_j and Y_i) to its correspondent previous score (this represents the similarity between X_j's neighbors and Y_i's neighbors) from the 4×4 area (the shaded area in Fig.2.5). The score in $score_{i,j}$ is updated with the maximum among all these 4×4 candidates. This process is iterated $M \times N$ times to generate the complete score matrix as shown in lines 3 and 4. Then we find the highest score within the matrix (lines 10 - 12), which represents the best alignment for X and Y. This highest score is used in the post processes to complete the genome reconstruction.

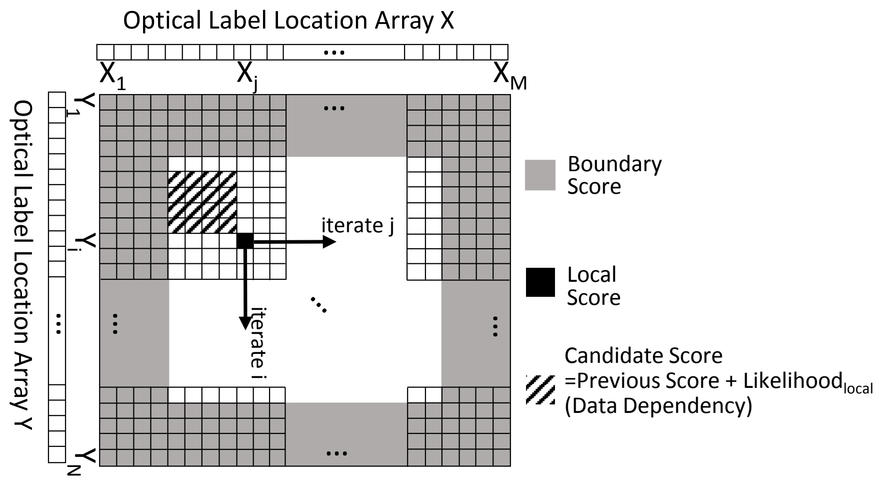

Optical Label Location Array X
X1
Xj
XM
Optical Label Location Array Y
Y1
Yi
YN
iterate j
iterate i
Boundary Score
Local Score
Candidate Score =Previous Score + Likelihoodlocal (Data Dependency)

Figure 2.5. Visualized pair-wise alignment process. The 2D array represents the likelihood score matrix. Each (i, j) element in the matrix is a likelihood score for aligning X_j with Y_i. The top, bottom, left and right grey regions represent the boundary score computations. The black square and the shaded area displays one iteration of the dynamic programming process. The computations for the black square have data dependencies to the shaded area. The arrows show that this computation is iterated to fill the entire matrix.

The likelihood functions in Algorithm 2.1 and 2.2 are derived from an error model proposed in [73]. The functions $Likelihood_{local}(x,y,m,n)$ and $Likelihood_{end}(x,m,n)$ are computed as shown in Equations 2.3 and 2.4 respectively. The $Likelihood_{local}(x,y,m,n)$ function consists of two terms: the bias value B_{XY} (provided in Equation 2.1); the maximum between the penalty value (provided in Equation 2.2) and a constant $P_{OutlierPenalty}$.

The values of the constants used in Equations 2.1 - 2.4 are empirically tuned to suit the optical experiment [73]. Changing these values does not influence the computing speed of the algorithm. Therefore, without the loss of generality, in our implementations, we tuned these constants to suit our experiment input data - a synthetic human genome. These constant values are listed in Table 2.1.

Let F represent the number of DNA fragments of an assembly process. Let M be the number of labels on the fragment. The algorithm requires $O(F^2N^2)$ times of $Likelihood_{local}$ operations to complete an assembly process. The DNA fragment pool typically has $100,000$ - $1,000,000$ arrays. A typical input array length (M or N) is 15 - 100 elements. Therefore, the number of $Likelihood_{local}$ operations, in a human genome assembly process, is on the order of 10^{15}. The total amount of computations requires more than 10,000 hours on a sequential CPU.

Each element of the input arrays represents a distance, which is on the order of thousands of bases, between two neighboring optical labels on the actual DNA fragment. The synthetic data used in our implementations is designed to simulate these properties of the real-world human genomes. Since the data ranges and array lengths are similar, the computation performance tested with this synthetic data reflects the performance with the real-world genomes.

Our focus is to accelerate this pair-wise algorithm which aligns the optical labeled DNA molecule fragments to construct the contigs in the assembly process. The scaffolding process, using optical labeled contigs, is not a computationally intensive operation which can usually be performed on a sequential computer in 10-30 minutes.

$$bias_{XY} = [max(0, x - \delta) + max(0, x - \delta)] * B + B^{'} \tag{2.1}$$

$$pen = C - \frac{(x-y)^2}{V*(x+y)} - P_{miss}*(m+n) - [max(0,x-\delta)+max(0,x-\delta)]*R \quad (2.2)$$

$$Likelihood_{local}(x,y,m,n) = bias_{XY} + max(pen,P) \quad (2.3)$$

$$Likelihood_{end}(x,m,n) = 2*max(0,x-\delta)*B_{end} + B^{'}_{end} - P_{miss}*(m+n-2) \quad (2.4)$$

Algorithm 2.1: The Boundary Score Algorithm

```
Input  : Two arrays of optical label locations $X$, $Y$; Sizes of the input arrays $1:M$, $1:N$
1  $Likelihood_{local}(x,y,m,n)$ local likelihood function
2  $Likelihood_{end}(x,m,n)$ end likelihood function
3  $Likelihood_{mix}(x,y,m,n) = Likelihood_{end}((x+y)/2,m,n) + Likelihood_{local}(x,y,1,1) - Likelihood_{local}((x+y)/2,(x+y)/2,1,1)$ mixed local and end likelihood function
4  for $i$ = 1 to $N$ do
5      $Lx_{i,j} = 0$, $Ly_{i,j} = 0$
6      if $i \leq 4$ then
7          $j_{max} = M$
8      end
9      else
10         $j_{max} = 4$
11     end
12     for $j$ = 1 to $j_{max}$ do
13         if $X_j < Y_i$ then
14             while $Y_i - Y_{Ly_{i,j}} > X_j$ do
15                 $Ly_{i,j}++$
16             end
17         end
18         else
19             while $X_j - X_{Lx_{i,j}} > Y_i$ do
20                 $Lx_{i,j}++$
21             end
22         end
23         $score_{i,j} = Likelihood_{end}(min(X_j,Y_i), j+1-max(1,Lx_{i,j}), i+1-max(1,Ly_{i,j}))$
24         if $X_j < Y_i$ then
25             for $k$ = $Ly_{i,j}$ to $i-1$ do
26                 $score_{i,j} = max(score_{i,j}, Likelihood_{mix}(X_j, Y_i - Y_k, j, i-k))$
27             end
28         end
29         else
30             for $k$ = $Lx_{i,j}$ to $j-1$ do
31                 $score_{i,j} = max(score_{i,j}, Likelihood_{mix}(X_j - X_k, Y_i, j-k, i))$
32             end
33         end
34     end
35 end
Output : Score matrix $score[1:N][1:M]$ filled with boundary scores in the top 4 rows and leftmost 4 columns
```

Algorithm 2.2: The Dynamic Programming Score Algorithm

```
Input  : Two arrays of optical label locations $X, Y$; Sizes of the input arrays $1:M$,
         $1:N$; Score matrix $score[1:M][1:N]$ with boundary scores filled
1  $Likelihood_{local}(x,y,m,n)$ local likelihood score function
2  $score_{best} = -\infty$
3  for $i$ = 1 to $N$ do
4      for $j$ = 1 to $M$ do
5          for $g = max(1, i-4)$ to $i-1$ do
6              for $h = max(1, j-4)$ to $j-1$ do
7                  $score_{i,j} =$
                     $max(score_{i,j}, A_{g,h} + Likelihood_{local}(x_j - x_h, y_i - y_g, j-h, i-g))$
8              end
9          end
10         if $score_{i,j} > score_{best}$ then
11             $score_{best} = score_{i,j}$, $j_{best} = j$, $i_{best} = i$
12         end
13     end
14 end
Output : Best score $score_{best}$; The $X$ and $Y$ indices of the best score $j_{best}$ and $i_{best}$
```

Table 2.1. Constant values for score functions

Constant	Value
V	0.0449
δ	0.0010
B	−0.0705
B'	0.9144
P_{miss}	1.5083
R	−0.0931
P	−8.1114
B_{end}	0.0226
B'_{end}	0.3992

2.4 Accelerated Designs

We partitioned the algorithm by accelerating some parts on the hardware and keeping some parts on the PC. This partitioning strategy is depicted in Fig. 2.4. The most computationally intensive stage of the algorithm is the local score computation. Therefore, in our partitioning, we accelerated the local score computation on the hardware. The two stages, boundary score computation and maximum score search, are not as computationally intensive as the local score stage. However, these two stages have significantly intensive data communication with the local score stage. In order to avoid this communication bottleneck between the PC and the hardware accelerator, we also assigned these two stages on the accelerator. The path trace stage consists of control intensive operations. Therefore, we kept this stage on the PC.

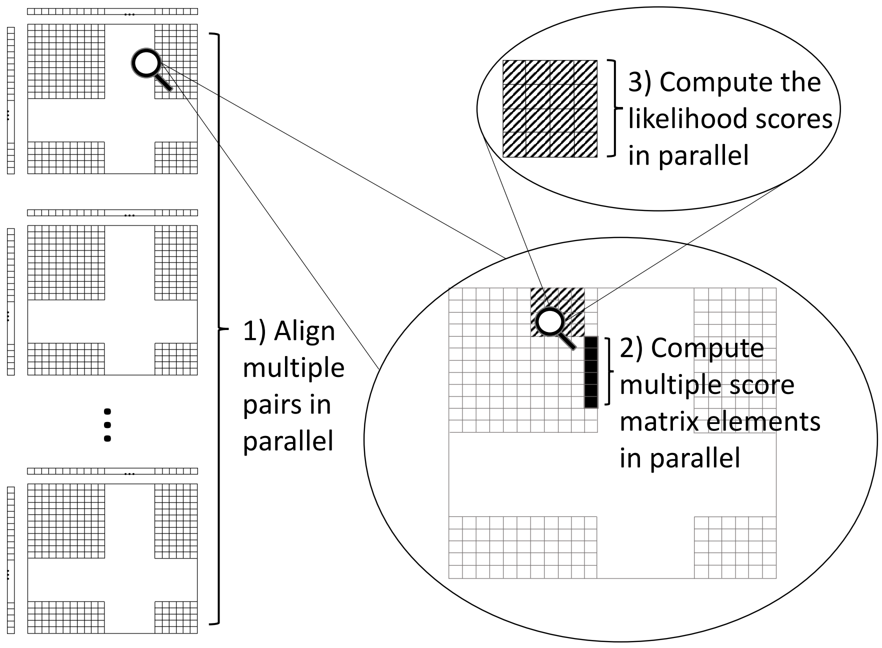

1) Align multiple pairs in parallel
2) Compute multiple score matrix elements in parallel
3) Compute the likelihood scores in parallel

Figure 2.6. Possible parallelism in the algorithm.

We identified three levels of possible parallelism in the algorithm (from coarse-grained to fine-grained): 1) align multiple pairs in parallel; 2) compute multiple elements (rows or columns) in the score matrix in parallel; 3) compute the 16 likelihood scores for each score element in parallel. These three levels and their hierarchy are depicted in Fig. 2.6. Particular computation and data reuse patterns exist in each level of possible parallelism. These patterns create tradeoffs in hardware accelerated designs.

When using level 1 parallelism (processing multiple pairs in parallel), each pair is data independent. Therefore, data communications or synchronization between parallel processes do not exist. However, it requires more computing resource to manage multiple alignments concurrently as well as more storage resource for intermediate data (e.g. multiple score matrices). On the other hand, the other two levels of parallelism (levels 2 and 3) provide more opportunities to share or reuse the data between the parallel processes due to their finer granularity. However, these two fine-grained levels of parallelism may introduce performance challenges such as higher synchronization overhead on processors and placement and route complexity on FPGAs. These complex architectural tradeoffs create design space exploration problems. We explored these design spaces to determine the proper level or combination of levels of parallelism to match the architectural features on the hardware. We also applied multiple optimization techniques on each design. We applied SIMD instructions and multi-thread techniques on the CPU design. For the GPU design, we tuned the CUDA code to tackle the data dependency caused by the local score computation. We also implemented a low level FPGA design due to the inefficient resource utilization provided by the state of the art high level synthesis tools. In the following sections, we describe the design space explorations and the optimal designs in detail.

2.4.1 Multi-core CPU

In the CPU design, we firstly improved the locality of the program by dictating the compiler to store the highly reused variables in the CPU registers. We then parallelized the algorithm by inserting OpenMP directives. The performance is highly correlated with the granularity of the iterations in the algorithm. We evaluated the fine-grained strategy which processes multiple rows and columns in parallel on the multiple CPU cores. The evaluation results indicated that it is expensive to synchronize and exchange fine-grained data among the cores. The multi-core CPU is more suitable for the coarse-grained parallelism. Therefore, we chose to align multiple pairs in parallel on the multi-core CPU.

We divided the total workload into several sets of alignment tasks and assigned each of the sets to a CPU core as demonstrated in Fig. 2.7. When one CPU core finishes its current alignment workload, it can start aligning another pair immediately without synchronizing with the other CPU cores. This setup does not create "dead" parallel processes or threads when the input array sizes change during the run-time. Therefore, all the CPU cores are completely occupied during this process. In addition, within each core, the process is in a sequential fashion which is suitable for control dominated operations such as the boundary score computation. We also forced functions $Likelihood_{local}(x,y,m,n)$ and $Likelihood_{end}(x,m,n)$ to be static and inlined in order to provide more optimization opportunities for the compiler.

The computations of the $Likelihood_{local}$ function provide us an opportunity to utilize the CPU SSE SIMD instructions. Therefore, we program the $Likelihood_{local}$ function to process 4 elements with a SIMD fashion using "__m128" type of its intrinsic operands.

2.4.2 GPU

The GPU design consists of three CUDA kernels, invoked from a C++ host code. The CUDA kernels accelerate the alignment algorithm to keep the intermediate data on the GPU during the process. The C++ host program only sends the input DNA arrays to

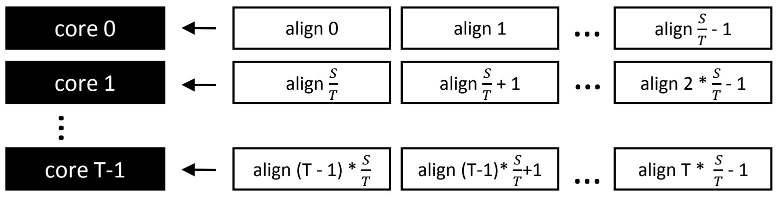

core 0
core 1
core T-1
align 0
align 1
align $\frac{S}{T}$ - 1
align $\frac{S}{T}$
align $\frac{S}{T}$ + 1
align 2 * $\frac{S}{T}$ - 1
align (T - 1) * $\frac{S}{T}$
align (T-1)* $\frac{S}{T}$+1
align T * $\frac{S}{T}$ - 1

Figure 2.7. Multi-core CPU accelerated design. Assume there are S pairs of optical arrays to be aligned and the CPU has T cores.

the first kernel and receives the output maximum score from the third kernel.

There are multiple options for CUDA kernel design based on different levels of granularity. We firstly evaluated the coarse-grained only strategy on the GPU. The evaluation shows that coarse-grained parallelism is significantly bounded by a low GPU occupancy. Therefore, to fully utilize the GPU parallel computing power, we added fine-grained parallelism in our design. The GPU design computes multiple rows and columns in fine-grained parallel within each GPU thread-block. The design also utilizes multiple thread-blocks to align multiple pairs in coarse-grained parallel. Computing the 16 candidates in parallel is not efficient on the GPU since it requires a 16-element reduction process which creates idle threads frequently.

We partitioned the algorithm into three CUDA kernels 1) boundary score kernel; 2) dynamic programming kernel; 3) maximum score search kernel. We chose this kernel partitioning because these parallelized computations require GPU global synchronization after 1) and 2).

In the boundary score kernel design, we fully parallelized the computations due to the data independency. The GPU thread arrangement is: assigning the boundary score computation for each element (lines 12 - 30 in Algorithm 2.1) to one GPU thread; assigning the boundary score computations of each alignment to one GPU thread-block. With this design, we maximized the GPU parallel resource occupancy. Moreover, since this design assigns all the computations of an alignment to the same thread-block, we

Listing 2.1. Pseudo Code for Dynamic Programming GPU Kernel

```
//gridDim.x=number of alignments
//blockDim.x=N, blockDim.y=4
__global__ void par_4_col_kernel(/*input/output arguments*/)
{
    int align_offset=M*N*blockIdx.x;
    //shared mem delecration
    //move input X, Y arrays from global memory to shared memory
    for (int col_id=0; col_id<M; col_id++)
    {
        if (threadIdx.y==0)
        {
            /*use feedback_score to compute the leftmost candidates and find
                the max for col_id+3*/
        }
        else if (threadIdx.y==1)
        {
            /*use feedback_score to compute the 2nd left candidates and find
                the max for col_id+2*/
        }
        else if (threadIdx.y==2)
        {
            /*use feedback_score to compute the 2nd right candidates and find
                the max for col_id+1*/
        }
        else if (threadIdx.y==3)
        {
            /*use feedback_score to compute the rightmost candidates and find
                the max for col_id*/
            //output the score for col_id
            //feedback_score[threadIdx.x]=score for col_id
        }
    __syncthreads();
    }
}
```

were able to store the intermediate data in the shared memory to minimize the memory access delay in the computations.

The pseudo code of the dynamic programming kernel is described in Listing 2.1. We parallelized the score element computations using $N \times 4$ threads in each thread-block. The candidate score computation for each matrix column requires 4 previous columns as described in line 7 of Algorithm 2.2. Parallelizing this part of the algorithm is a challenging task due to this data dependency. We overcame this issue by dynamically assigning the columns of the score matrix to 4 groups of threads. As described in Listing

2.1, we used $threadIdx.y$ to partition $N \times 4$ threads into 4 groups. They form a software pipeline. Each thread group is only responsible for a specific candidate computation (leftmost, 2nd left, 2nd right or rightmost). By increasing col_id, we stream the columns of the score matrix into this pipeline.

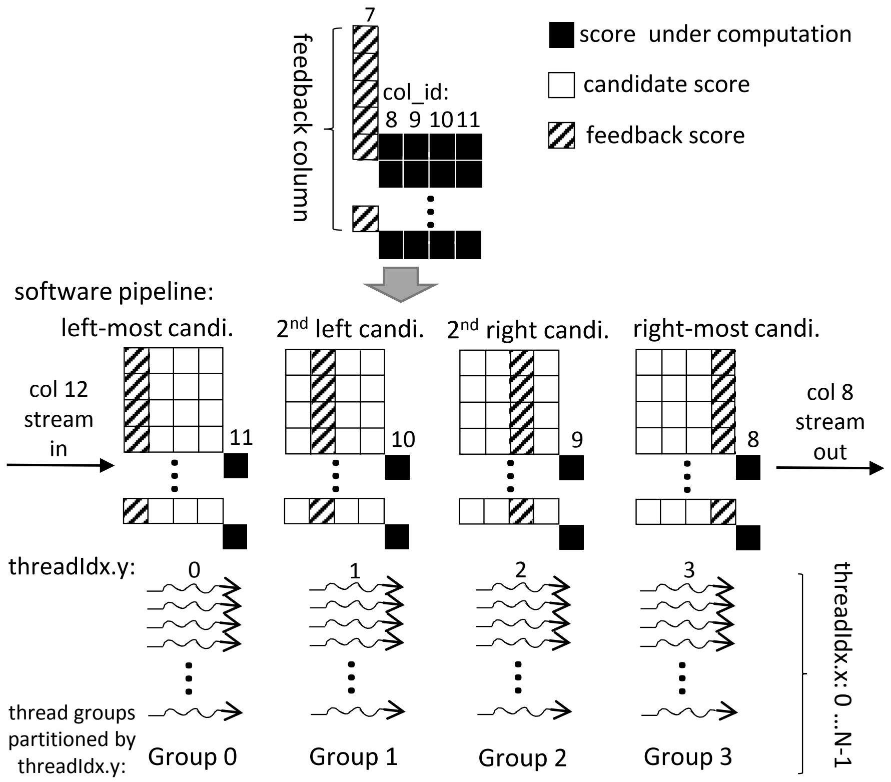


Figure 2.8. Visualized GPU kernel for dynamic programming, assuming the score matrix size is $M \times N$. N rows $\times$ 4 columns of score elements are computed concurrently. For example, columns 8,9,10 and 11 are computed concurrently. At the given state in the example, column 11 is assigned to the threads whose $threadIdx.y = 0$. Then the leftmost candidates of column 11 are computed using the previously computed data in column 7. Similarly, columns 10,9 and 8 are assigned to $threadIdx.y = 1$, $threadIdx.y = 2$ and $threadIdx.y = 3$ respectively. N is set to a multiple of 32 to ensure each warp has the threads with the same $threadIdx.y$.

The example in Fig. 2.8 shows a snapshot of this software pipeline when the GPU is processing columns 8, 9, 10 and 11. These columns are assigned to the different stages (thread groups) of the pipeline: column 11 to group $threadIdx.y = 0$; column 10 to $threadIdx.y = 1$; column 9 to $threadIdx.y = 2$; column 8 to $threadIdx.y = 3$. The computations in the pipeline stages $threadIdx.y = 0 - 3$ are leftmost candidates, 2nd left candidates, 2nd right candidates and rightmost candidates respectively, as shown in the shaded blocks in Fig. 2.8. In the snapshot, these computations all require the data from column 7 which has already been computed in the previous col_id iteration (refer to the "for" loop in Listing 2.1). Once the computations in the snapshot are finished, the data in column 8 is then ready. With the data from column 8, the pipeline streams a new column (column 12 in the snapshot) by increasing the iteration index col_id. These 4 thread groups execute different instructions to implement the 4 stages of the pipeline. In order to fit this design on the GPU SIMD architecture, we ensured the threads of each GPU warp to execute the same instruction by extending N to a multiple of 32.

Once the dynamic programming kernel finishes computing the score matrix, the third kernel searches the matrix to find $score_{best}$, i_{best} and j_{best}. We implemented this maximum score search kernel using the reduction approach. We kept the reduction process of each alignment within one thread-block. Therefore, this process does not require the expensive global synchronization on the GPU. Then, we created multiple thread-blocks to concurrently process the reductions for multiple alignments. We also applied shared memory and efficient warp arrangement in the reduction.

2.4.3 FPGA

Similar to the GPU, the FPGA also accelerates the algorithm by processing the computations in a parallel fashion. The FPGA is a customizable architecture. There are usually two ways to implement the parallelism on the FPGA: 1) replicate a logic module multiple times to physically create multiple parallel data paths; 2) pipeline the architecture to process the multiple data concurrently in a streaming fashion. A high performance design requires a proper decision on which technique is used to implement each of the three levels of the algorithmic parallelism. Moreover, due to the FPGA resource constraints, a feasible design also requires the proper number of replications in each level of parallelism. There exists many possible settings of choices of parallel techniques and numbers of replications. In order to reduce the size of the design space, we firstly constructed a reasonable structure of the FPGA design based on heuristics. Fig. 2.9 depicts this FPGA structure. Due to the algorithmic data dependency, it is impossible to replicate parallel data paths for both row and column dimensions. Therefore, we chose to only replicate the row parallel data paths (level 2 parallelism)and pipeline the column dimension (level 2 parallelism). We replicated the likelihood score units (level 3 parallelism) to sustain the throughput of this full pipeline in the column dimension. We also replicated the entire alignment module multiple times (level 1 parallelism) to maximize the overall throughput. We then permutated the numbers of parallel paths in this structure to find the optimal setting.

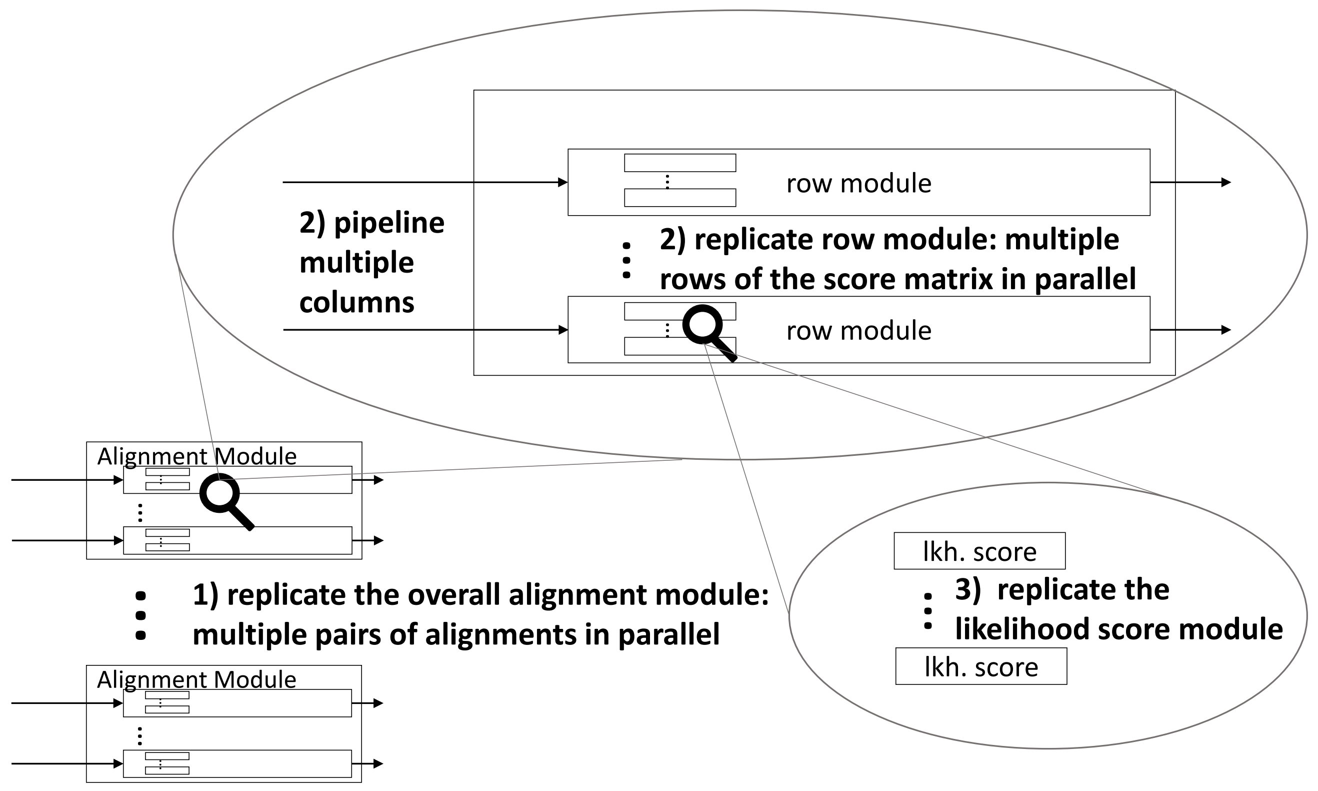
Implementing multiple RTL designs to measure the performances of these permutations requires a significant amount of effort. Therefore, exploring the FPGA design space using RTL designs is inefficient in terms of the development complexity. Instead of manually implementing multiple RTL designs, we propose a method using Vivado HLS which enables rapid FPGA implementations to explore different parallel structures.

We restructured the original software C code, as described in Listing 2.2, into the format that represents the parallel hardware structure. We firstly constructed a function *lkh_score()* to implement the likelihood score computation in equation 2.3. To implement the full pipeline, we restructured the local score computation code into a function *pipeline_unit()* with 4 pipeline stages. Lines 35 - 39 describe a line buffer used to feed the input into this pipeline. We then call *pipeline_unit()* in the *alignment_module()* function.

Listing 2.2. Pseudo Code for local score element computation function in HLS C code

```
void lkh_score(/*argument declaration*/)
{
 //compute likelihood score as shown in equation ref{equ:s_function}
}

void pipeline_unit(/*argument declaration*/)
{
    #pragma HLS ALLOCATION instances=lkh_score limit=16
    //use HLS ALLOCATION directive to control the number of lkh_score
        replications

    /*declare buffering variables for the 4 stages (4 columns)*/

    /*call lkh_score() to compute likelihood scores for 0 − 3 columns*/

    //pipeline
    /*stage0: max for column 0*/
    /*stage1: max for column 1*/
    /*stage2: max for column 2*/
    /*stage3: max for column 3*/
}

void alignment_module(/*argument declaration, e.g. input: DATA_TYPE x*/)
{
    #pragma HLS ALLOCATION instances=pipeline_unit limit=5
    /*use HLS ALLOCATION directive to control the number of pipeline_unit
        replications*/

    /*declare x, y line buffers:
    e.g. DATA_TYPE x0,x1,x2,x3,x4;*/

    for(/*iterate row index*/)
    {
        for(/*iterate column index*/)
        {
            //update the x line buffer
            x4=x3;
            x3=x2;
            x2=x1;
            x1=x0;
            x0=x;
            /*call pipeline_unit(x0,x1,x2,x3,x4,...)*/
        }
    }
}

void top_module(/*argument declaration*/)
{
    #pragma HLS ALLOCATION instances=alignment_module limit=2
    /*use HLS ALLOCATION directive to control the number of alignment_module
        replications*/

    /*call alignment_module()*/
}
```

We replicated multiple instances of *lkh_score()*, *pipeline_unit()* and *alignment_module()* to generate parallel data paths in the three levels. Finally, we used function *top_module()* to wrap up these sub-modules. This new C code structure eases the scheduling task for the HLS tool to generate efficient architectures.



2) pipeline multiple columns
row module
2) replicate row module: multiple rows of the score matrix in parallel
row module
Alignment Module
1) replicate the overall alignment module: multiple pairs of alignments in parallel
Alignment Module
lkh. score
3) replicate the likelihood score module
lkh. score

Figure 2.9. FPGA implementation of the three levels of parallelism. 1) replicate the overall architecture to process multiple alignments in parallel; 2) within each alignment score matrix, replicate the row modules to process multiple rows in parallel and pipeline multiple columns; 3) for each score element, replicate the likelihood score module to compute multiple likelihood scores in parallel.

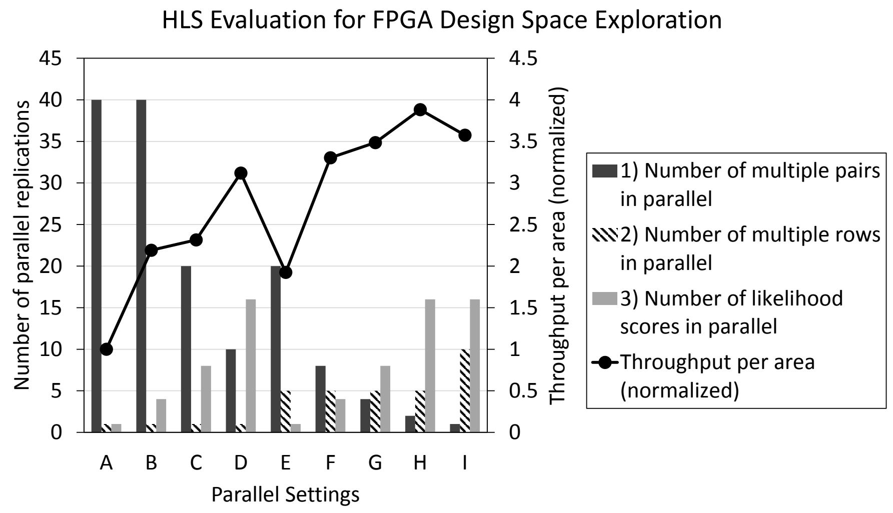


Figure 2.10. FPGA design space exploration using HLS. Evaluations of 10 different designs on the Xilinx VC707 FPGA board (Design A - J). Throughput per area for each design is normalized to the lowest value.

We permutated the numbers of the three levels of data path replications in the restructured C code by modifying the *limit* parameter in the *HLS ALLOCATION* directives. We evaluated 10 different settings by running the entire HLS design tool chain including the placement and route phase. Fig. 2.10 depicts the evaluations of these HLS designs. The experimental results indicate that design *H* achieves the highest throughput efficiency among all the evaluated designs. We then implemented design *H* in RTL to further improve the resource efficiency.

Our RTL FPGA design consists of two modules: 1) boundary score module and 2) dynamic programming and maximum score module. To achieve a high throughput, we fully pipelined the FPGA architecture to output a new likelihood score every clock cycle. The two modules are able to run concurrently in a streaming fashion.

The architecture of the boundary score module is described in Fig. 2.11. In this figure, we demonstrate the boundary score module by only showing an example computing the scores at the 4th top row. The other rows and columns are identical to this example. We replicated this architecture 16 times to process the top 4 rows, bottom 4 rows, left 4 columns and right 4 columns of boundary scores in parallel. This boundary score module is fully pipelined and consists of control logic (the black blocks in the figure), arithmetic units (the grey blocks), muxer and a shifting register for X. The control logic and arithmetic units correspond to Algorithm 2.1. The shifting register is for accessing $X_{Lx_{i,j}}$ and X_k as shown in lines 17 and 28.

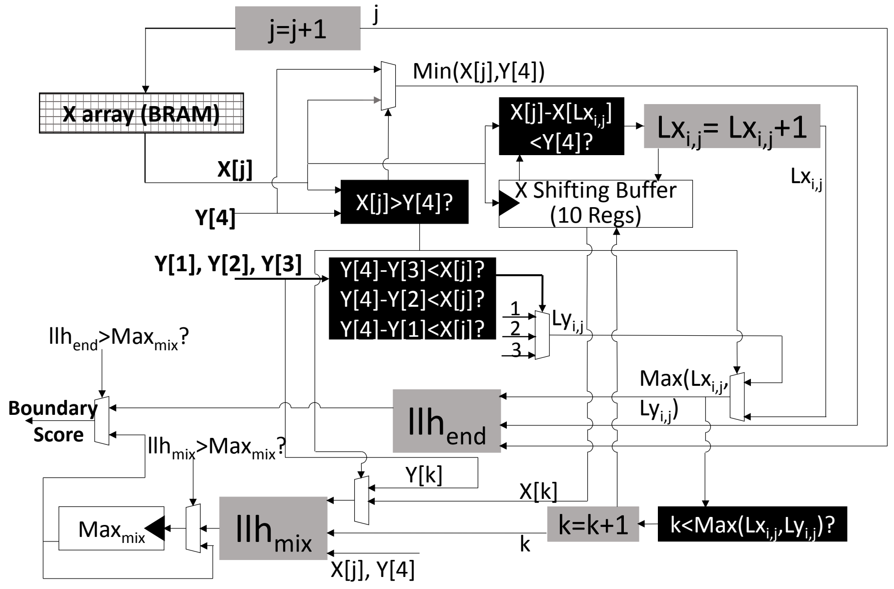


Figure 2.11. FPGA boundary score module. The control logic corresponds to Algorithm 2.1. A shifting register storing 10 elements of X is used for accessing $X_{Lx_{i,j}}$ and X_k efficiently. llh_{end} and llh_{mix} represent the likelihood score modules for $Likelihood_{end}$ and $Likelihood_{mix}$.

The design of the dynamic programming module is described in Fig. 2.12. This architecture consists of 5 major pipeline stages as shown in Fig. 2.12. Stage 0 computes 16 (4×4) $Likelihood_{local}$ functions in parallel. These $Likelihood_{local}$ modules are fully pipelined. Stages 1 - 4 compute the maximums of the leftmost, second left, second right and rightmost columns of candidates, respectively.

We replicated the described architecture 5 times to process 5 rows of scores in parallel. After the last column of the current 5 rows, the next 5 rows will enter this architecture to continuously fill the pipeline. The output of all the rows are passed to a pipeline maximum module to find $score_{best}$, i_{best} and j_{best}. We chose to process 5 rows in parallel to match the throughput of the boundary score module. The two modules are thus able to run in a streaming fashion without idling.

As depicted in Fig. 2.12, the results of Stage 0 are delayed by the registers to feed Stages 2 - 4 at the correct cycles. Shown in the figure, the computation for $score[i][j+3]$ is at Stage 1; $score[i][j+2]$ is at Stage 2; $score[i][j+1]$ is at Stage 3; $score[i][j]$ is at Stage 4. These stages are all using $score[i-1:i+3][j-1]$. $score[i-1:i-4][j-1]$ are the scores created and stored in the BRAMs during the computation of the previous 5 rows. $score[i:i+3][j-1]$ are created from the previous cycle as a feedback loop. Therefore, in order to keep the pipeline outputting new data every cycle with the constraint of this feedback loop, we designed a combinational logic to compute the 4 parallel additions and the "Max 5 to 1" operation within one clock cycle.

We used fixed point numbers and arithmetic in the FPGA design. Due to the data range, we used 26 bits for the scores and 18 bits for the input arrays, both with 10 decimal bits. These fixed point numbers can represent the optical labeled fragments up to $1,000,000$ bases. This range covers most human genome assembly applications. In the score functions, we implemented the divisions using lookup tables.

Since the RTL implementation is more resource efficient compared with the HLS implementation, we were able to replicate the overall alignment architecture 5 times to align 5 pairs of DNA molecules concurrently. In the HLS design, we were only able to replicate this architecture 2 times. We enhanced the resource efficiency by more than 200% using the RTL design.

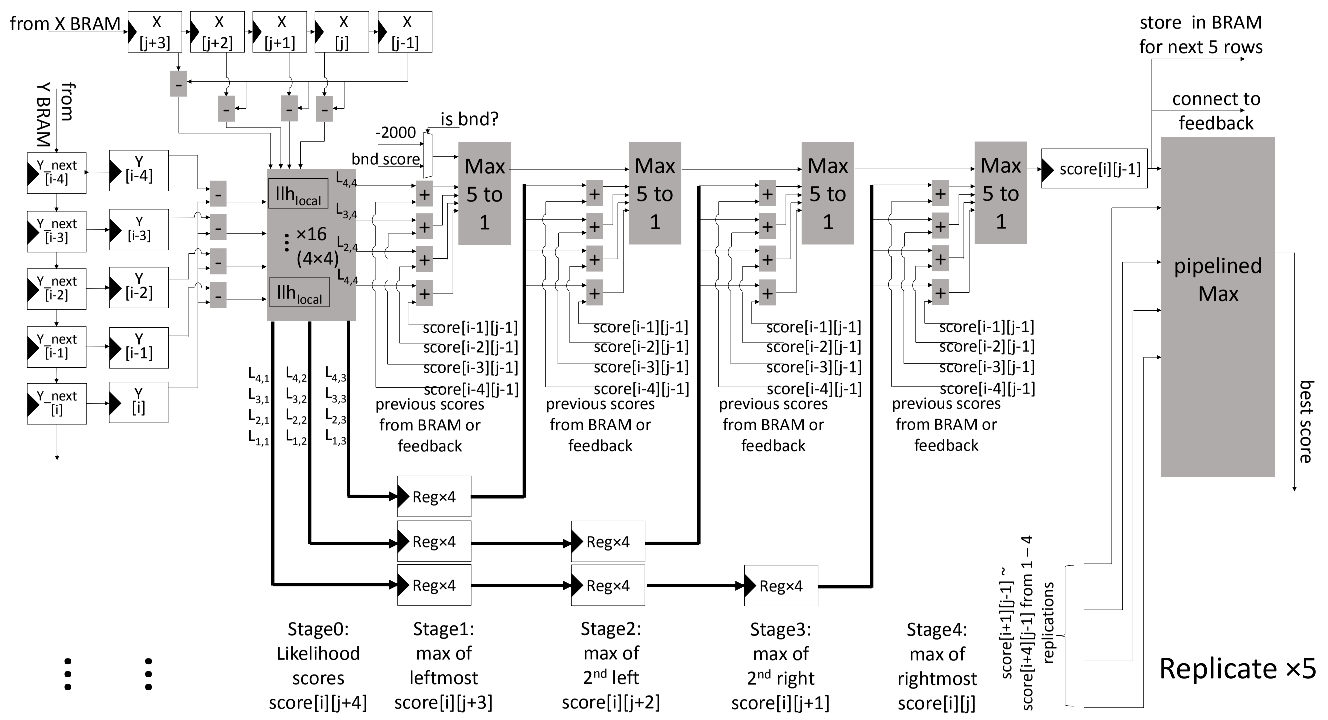


Figure 2.12. FPGA dynamic programming module with three levels of concurrency. First, the 16 (4×4) $Likelihood_{local}$ functions are computed concurrently with 16 parallel llh_{local} modules. Second, this architecture is fully pipelined. $score[i][j+3]$, $score[i][j+2]$, $score[i][j+1]$ and $score[i][j]$ are processed concurrently in the different stages of the pipeline. Third, the architecture is replicated to process 5 rows in parallel.

2.5 Results

The input data for the alignment algorithm in our experiment consists of 16642 DNA molecule fragments. Each fragment contains 5 – 182 labels. The range of the distances between labels is 500 – 3.79×10^5 bases.

We tested the multi-core CPU design on a 3.1GHz Intel Xeon E5 CPU with 8 cores. The CPU design was compiled with O3 GCC optimizations. The GPU design was tested on a Nvidia Tesla K20 Kepler card. The FPGA design was implemented and tested on a Xilinx VC707 FPGA development board. The input data used in our experiments is a set of synthetic human genome sequences.

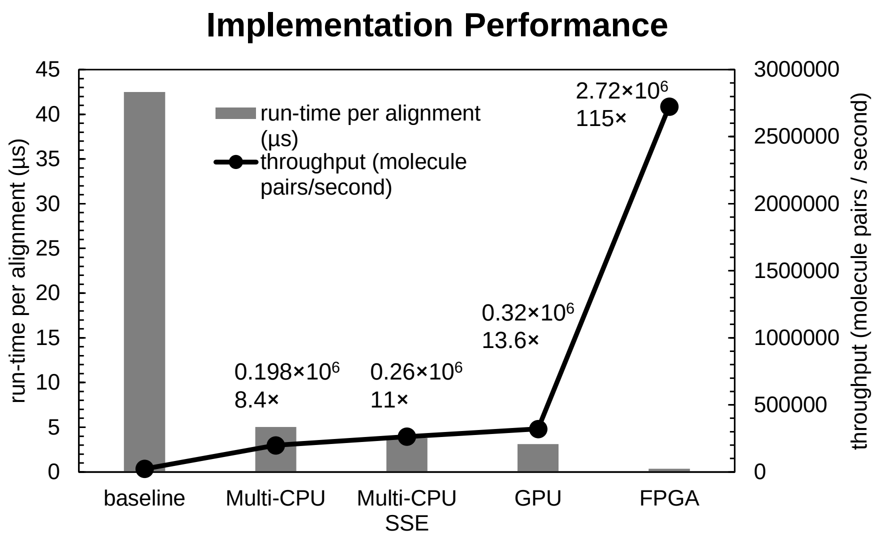


Figure 2.13. Performance of the accelerated designs. Speedup factors against the single CPU baseline implementation. Throughput is defined as the number of DNA molecule pairs that a design is able to process in 1 sec.

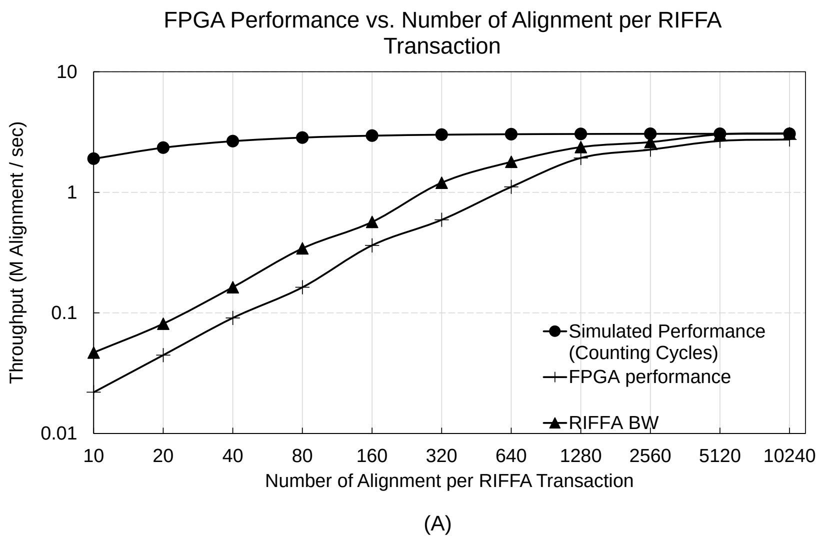
Fig. 2.13 presents the performance of our implementations. The baseline is a highly optimized C++ program without any parallelism. The average time for aligning two optical labeled molecules is $42.486\mu s$ in the baseline implementation. The run-times for the boundary score, dynamic programming and maximum score operations are $9.351\mu s$, $30.594\mu s$ and $2.541\mu s$ respectively.

The OpenMP parallelized C++ program consumes $5.04\mu s$ aligning a pair of molecules on an 8 core CPU with hyper-thread technology on each core. The performance of this multi-core implementation achieves a $8.4\times$ speedup which is proportional to the number of cores. The extra $0.4\times$ speedup is contributed by hyper-thread. The CPU SSE SIMD technique boosts the performance with a $11\times$ speedup against the baseline.

Our GPU implementation was written using the Nvidia CUDA 6.5 SDK. The GPU runs at a base frequency of 706 MHz and has 2496 CUDA cores. We varied the number of input alignments per host-device data transfer transaction from 10 to 10240 to investigate how the GPU design performs. As shown in Fig. 2.14 (B), the dynamic programming kernel converges to the minimal run-time after increasing the number of alignments per transaction to 2560. The boundary kernel and the max reduction kernel converge to their minimums when the number of alignments per transaction hits 640. The data copying operation from the device to host keeps speeding up with the increase of the number of alignments. This is due to the fact that the output array only contains very few data (each alignment only generates one max score, and two indices of the X and Y arrays) which never saturates the memory transaction bandwidth. However, the dynamic programming kernel dominates the overall run-time. Therefore, as a consequence, the overall performance saturates the max throughput after increasing the number of alignments to 2560.

The best performance of the GPU design is at $3.116\mu s$ per alignment with a $13.6\times$ speedup against the baseline. The run-time for the boundary score, dynamic programming and maximum score kernels are $0.940\mu s$, $1.484\mu s$ and $0.494\mu s$ respectively. The data transferring time between the host memory and GPU memory is $0.198\mu s$.

The FPGA design was built using Xilinx ISE 14.7 in Verilog. The FPGA design was implemented on a Xilinx Virtex 7 VC707 board receiving input and sending output using RIFFA [74] (configured as a x8 Gen 2 PCIe connection to the PC). We also



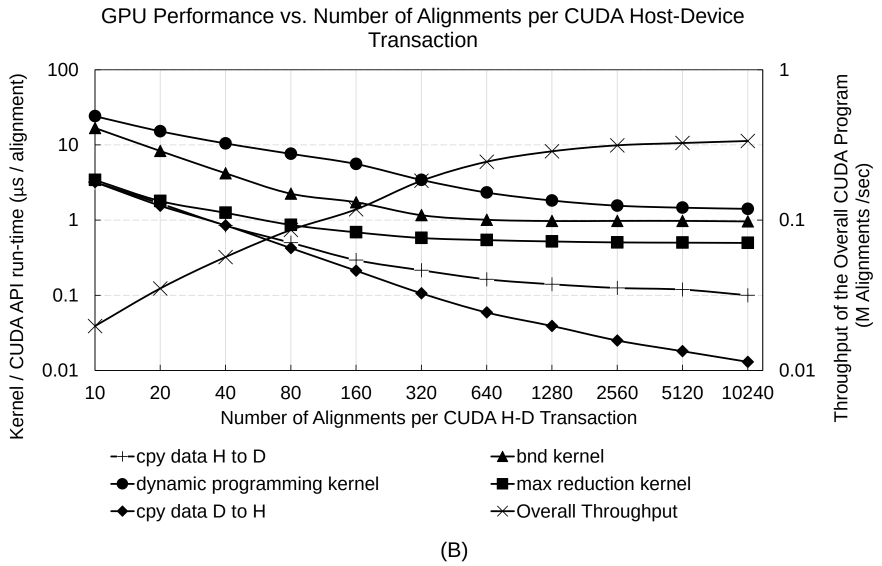


Figure 2.14. Performance vs. number of alignments per host - accelerator device transaction. (A) FPGA performance vs. number of alignments per RIFFA send/receive transaction; the simulated performance is calculated by counting the cycles of the RTL design; the RIFFA bandwidth is measured in M alignments per second. (B) GPU performance (throughput) vs. number of alignments per CUDA H-D or D-H transaction; the run-time curve of each kernel and CUDA API is also plotted.

investigated how the FPGA design performs when changing the number of alignments per RIFFA transaction between the host CPU and the FPGA. We also varied the number of alignments per transaction from 10 to 10240. The simulated design generates the ideal throughput of the FPGA acceleration module. We measured the bandwidth of RIFFA by sending the data to the FPGA and receiving the same data back to the host without doing any computation on the FPGA. We only measured the host to device bandwidth since the transfer from the device to host contains very few data. As shown in Fig. 2.14 (A), the actual FPGA performance is significantly lower than the simulated ideal performance due to the RIFFA bandwidth limit when the number of alignments is between 10 and 640. After the number of alignments reaches 5120, the actual FPGA performance converges to its maximum which is slightly lower than the ideal performance due to the host software overhead.

Our FPGA experimental result shows the best throughput at 2.7 million pairs of molecules per second or equivalently $0.367\mu s$ per alignment. Thus, the FPGA implementation achieves a $115\times$ speedup against the baseline. Our FPGA implementation runs at a frequency of 125 MHz. Table 2.2 lists the resource utilization of the entire design including the PCIe communication logic. The design occupied 89% of the slices on the FPGA. In order to meet the timing constraint with such a high logic utilization, we used "SmartXplorer" to permutate multiple placement and route strategies in ISE. Since we used fixed-point number representation in the FPGA design, compared to the baseline floating-point design, we observed a 0.019% error which is negligible in real applications.

Table 2.2. FPGA design resource utilization on VC707

Slice Reg.	Slice LUT.	BRAM	DSP48E
150412	251979	159	2280
24%	82%	15%	81%

2.6 Hardware Comparison

Table 2.3 presents the summary of the comparison between the three hardware. We compare the performances and prices of the hardware accelerators.

Table 2.3. Hardware Comparison

Accelerator	Multi. Core CPU	GPU	FPGA
Parallel Architecture	Coarse-grain cores	Massively parallel threads	Replicated paths
Solution for data dependency	NA	Thread/Warp strategy	Manually design full pipeline & single-cycle logic
Customized Operator	NA	NA	Look up table division
Customized Bit-width	No	No	Yes
Frequency	3.1 GHz	706 MHz	125 MHz
Performance	8.4×	13.6×	115×
DSE & Develop Effort	2 weeks	3 months	9 months
Price	\$2,000	\$3,200 + \$2,000	\$3,495 + \$2,000
Performance per \$ (aligns/sec/\$)	99.2	61.7	495.5

2.6.1 Performance

Although the multi-core CPU has the highest operating frequency among the three hardware, it achieves the lowest speedup. This is due to the fact that the multi-core CPU has very limited parallel computing resources: 8 cores with hyper-thread. These cores are not closely coupled in the architecture. Frequently synchronizing these cores for fine-grained parallel computations becomes significantly expensive. Therefore, we were only able to utilize these cores to align multiple molecule pairs in a coarse-grained parallel fashion. The SSE SIMD extension provides a limited level of fine-grained parallelism. The CPU 128-bit SIMD extension does not provide dedicated SIMD units to achieve massive fine-grained parallelism.

The GPU, as opposed to the multi-core CPU, has a SIMD architecture that supports fine-grained parallelism. We therefore observed a higher speedup on the GPU. However, the control dominated boundary score computations introduce a significant amount of diverse instructions which harm the parallelism in the SIMD architecture. The GPU accelerates the dynamic programming algorithm by $20\times$ while it only accelerates the boundary score algorithm by $10\times$. Compared with the GPU, each CPU coarse-grained parallel core is processing each alignment in a sequential fashion which has more advantage in dealing with the control dominated instructions. Moreover, the array size differences create multiple *inactive* threads. With the CUDA profiler "nvvp", we observed that these inactive threads occupy more than 45% of the GPU computing resource due to the control branch diversities. Compared with the GPU design, the multi-core CPU and the FPGA suit this feature of the algorithm better. The multi-core CPU coarse-grained parallelism avoids inactive threads. Unlike the GPU threads issued before the program starts and unchangeable during the run-time, the FPGA pipeline terminates and moves on to the next array when the current array finishes during the run-time. These unsuitable GPU features limit the performance for the alignment algorithm.

On the FPGA architecture, the customized logic avoids the diverse instruction issue in the boundary score algorithm. In the dynamic programming module, the pipeline on the FPGA is spatial. The data is transferred from one logic to the next logic using on-chip registers. In opposition, in the GPU design, we implemented a similar optimization using warps (different $threadIdx.y$) as shown in Listing 2.1 and Fig. 2.8. Each GPU warp represents a logic module on the FPGA. Unlike the spatial pipeline, the GPU warps are scheduled temporally. The data is not transferred spatially between warps. In contrast, the warps read or write the data on the shared memory. Although these warps are designed to be processed efficiently on the GPU, the FPGA spatial pipeline still outperforms the GPU warps without the overhead from scheduling and memory access. Moreover, the boundary score module stores its output in the low latency BRAM on the FPGA. The dynamic programming module can then access these boundary scores within one clock cycle. As opposed to the FPGA, the GPU dynamic programming kernel reads the boundary scores from the global memory with a higher latency. For these reasons, the FPGA implementation achieves the highest performance.

2.6.2 Hardware Prices

The prices of the acceleration hardware vary significantly depending on the complexities of the devices. The devices used in our implementations belong to the high-end category. The Xilinx VC707 FPGA evaluation board costs about $\$3,495$ [75]. The Nvidia K20 GPU can be purchased for $\$3,200$. These high-end GPUs and FPGAs have comparable prices. The high-end CPU, Intel Xeon E5 has a relatively lower price which is roughly $\$2,000$. In our comparison, we added the CPU price to the cost of the GPU and FPGA accelerated systems since both of them used the CPU as a host to send and receive data. In our application, the performances per dollar are 99.2 $aligns/sec/\$$, 61.7 $aligns/sec/\$$ and 495.5 $aligns/sec/\$$ for the multi-core CPU, GPU and FPGA respectively.

2.7 Summary

In this chapter, we have addressed the necessity to accelerate the optical label based DNA assembly. We have presented three different accelerated approaches: a multi-core CPU implementation, a GPU implementation and a FPGA implementation. We have also presented the detailed design space explorations for these three approaches. The speedups over the sequential CPU baseline are $8.4\times$, $13.6\times$ and $115\times$ for the multi-core

CPU, GPU and FPGA respectively. Using spatial pipelines, the FPGA design has been customized to suit the algorithm more efficiently than the other two hardware. The tradeoff to this performance efficiency on the FPGA is its significant design complexity in comparison with the approaches on the other two hardware.

Acknowledgements

This chapter contains material printed in “Hardware Accelerated Alignment Algorithm for Optical Labeled Genomes”, Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, the ACM Transactions on Reconfigurable Technology and Systems 2016 and in “Hardware Accelerated Novel Optical De Novo Assembly for Large-Scale Genomes”, Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner, International Conference on Field-Programmable Technology (FPT), December 2014. The dissertation author was the primary investigator and author of these papers.

Chapter 3

FPGA-GPU-CPU Heterogeneous Architecture for Real-time Cardiac Physiological Optical Mapping

3.1 Background and Motivation

Optical mapping technology has proven to be a useful tool to record and investigate the electrical activities in the heart [39, 40]. Unlike other cardio-electrophysiology technologies, it does not physically interfere with the heart. It provides a dense spatial electrical activity map of the entire heart surface. Each pixel acts as a probe on that location of the heart. Variation in pixel intensity over time is proportional to the voltage at that location. Thus a 100×100 resolution video is equivalent to 10,000 conventional probes. This produces more accurate and comprehensive information than conventional electrode technologies.

The process of optical mapping involves processing video data to extract biological features such as depolarization, repolarization and activation time. The challenge in this process is primarily in the image conditioning. Raw video data contains appreciable sensor noise. Direct extraction of biological features from the raw data yields results too inaccurate for most medical use. Therefore, the process includes an image conditioning algorithm, which has been presented and validated by Sung *et. al* [76]. The effect of this image conditioning is shown in Fig. 3.1.

Real-time optical mapping is useful and potentially necessary in a wide range of applications. One domain is real-time closed loop control systems. This includes dynamic clamp [77, 78], and the usage of tissue-level electrophysiological activity to prevent the onset of arrhythmia [79, 80]. These systems offer the unique ability to understand the heart dynamics by observing real-time stimulus/response mechanisms over a large area. Another domain of applications is immediate experimental feedback. The ability to see the optical mapping results during the experimental procedure can significantly reduce both the duration of the experiment and the required number of experiments.

Achieving real-time optical mapping is computationally challenging. The input data rate and the required accuracy for biological features results in a throughput on the order of $10,000$ fps. At such high throughput, a software implementation takes 39 mins to process just a second of data. Even a highly optimized GPU accelerated implementation can only reach 578 fps. A FPGA-only implementation is also infeasible due to the resources required for processing intermediate data arrays generated by the optical mapping algorithm.

In this chapter, we propose a real-time FPGA-GPU-CPU heterogenous architecture for cardiac optical mapping that runs in real-time, capturing 100×100 pixels/frame at 1024 fps with only 1.86 seconds of end to end latency. Experimental parameters and data are based on the experiments by Sung *et. al* [76]. Our design has been implemented on an Intel workstation using an NVIDIA GPU and a Xilinx FPGA. The implementation is a fully functioning end to end system that can work in an operating room with a suitable camera.

The rest of the chapter is organized as follows. We discuss related work in Section 3.2. In Section 3.3, we describe the optical mapping algorithm in detail. We discuss algorithm partitioning decisions in Section 3.4. We describe the design and

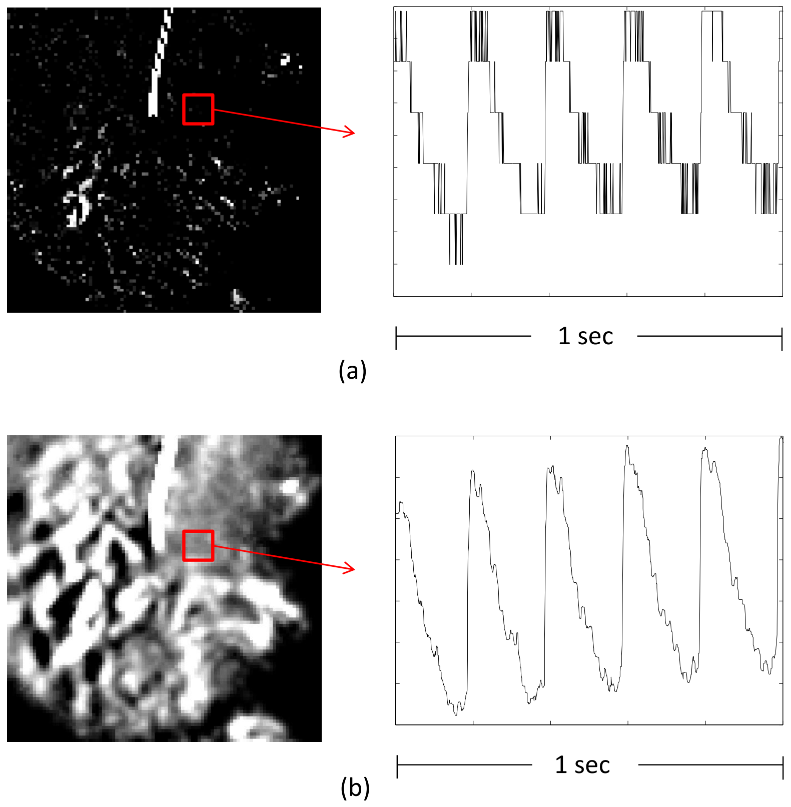


Figure 3.1. Image conditioning effect (left: the grayscale image of a random frame, right: the waveform of a random pixel over time). (a) before image conditioning. (b) after image conditioning.

implementation of the heterogenous architecture in Section 3.5. In Section 3.6, we present the experimental results and accuracy of our implementation. Section 3.7 summarizes this chapter.

3.2 Related Work

The optical mapping process involves three types of computations: spatiotemporal image processing, spectral methods, and sliding-window filtering that can result in performance challenges. A variety of approaches have been proposed to accelerate image processing algorithms that have one or more of these computations. There are FPGA and GPU accelerated approaches for real-time spatiotemporal image processing [81, 82]. Govindaraju *et. al* have analyzed the GPU performance on spectral methods [83]. Pereira *et. al* have presented a study of accelerating spectral methods using FPGA and GPU [84]. Many sliding-window filtering applications have been presented in the past [35, 85]. None of the approaches described above combine all three of the computations as in the optical mapping algorithm.

Several FPGA-GPU-CPU heterogeneous acceleration systems have been proposed in recent years. Inta *et. al* have presented a general purpose FPGA-GPU-CPU heterogeneous desktop PC in [86]. They reported that an implementation of a normalized cross-correlation video matching algorithm using this heterogeneous system achieved 158 fps with 1024×768 pixels/frame. However, they ignored the throughput bottleneck of the PCIe which is critical in real-time implementations. Bauer *et. al* have proposed a real-time FPGA-GPU-CPU heterogeneous architecture for kernel SVM pedestrian detection [87]. However, instead of having spatiotemporal image processing and spectral methods (across frames), this application only has computations within individual frames.

We present a stage level algorithm partitioning according to the computational characteristics and data throughput. To the best of our knowledge, the system presented in this chapter is the first implementation of a real-time optical mapping system on a heterogenous architecture.

3.3 Optical Mapping Algorithm

Fig. 3.2 (a) depicts an overview of the algorithm. Video data is provided by a high frame rate camera. The input video data is zero score normalized to eliminate the effects of varying background intensities. After normalization, there are two major noise removing facilities: a phase correction spatial filter and a temporal median filter.

3.3.1 Normalization

Normalization is performed for each pixel in a temporal fashion, across frames. In our experiments the input video arrives at 1024 fps. Normalization is performed on each second of video, disjointly. To compute the normalization base value for pixel location, we find the weighted mean of the largest three values in the temporal array. We can then normalize each pixel in the frames using Equation 3.1 with the correspondent normalization base value.

$$normed.\ pixel = 100\frac{-(raw\ pixel - base\ val.)}{base\ val.} \tag{3.1}$$

3.3.2 Phase Correction Spatial Filter

The action potential is distributed as a waveform on the heart surface. Thus, if we merely apply a Gaussian spatial filter on the video data, we will lose the critical depolarization properties (the sharp edges of the waveform in cardiac physiology). Therefore, a phase correction algorithm needs to be applied to cause the pixels in the window to be in phase before the Gaussian spatial filter.

The phase correction spatial filter operates as a sliding window function across the entire frame, where each operation uses all frames across time (see Fig. 3.2 (b)). Fig. 3.2 (c) illustrates an example 5×5 Gaussian filter.

3.3.3 Phase Correction Algorithm

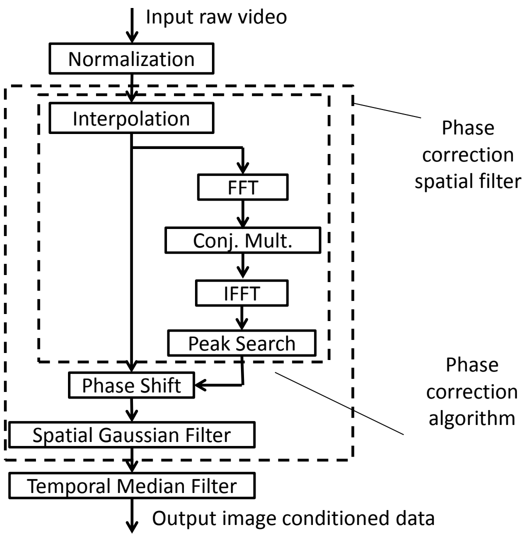
In order to correct the phases of the pixels, the phase difference must be computed between the center pixel and all of its surrounding neighbors in the filter window. We can calculate this difference using a bit of signal processing theory as presented by Sung *et. al* [76].

This phase correction operation is illustrated graphically in Fig. 3.2 (d). First, the frame arrays are interpolated by a factor of 10 using an 81 tap FIR filter. This provides a higher resolution for phase differences. Then pairs of temporal arrays are compared, the center pixel array and a neighbor pixel array. The arrays are converted into the Fourier domain by a FFT. After that, the neighbor FFT array is conjugated and multiplied with the center FFT array. The result of the multiplication is converted back into time domain by an IFFT. The index of the pulse in the IFFT array represents the phase difference.

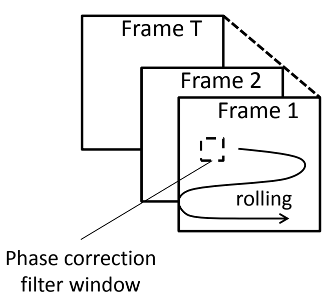
After finding the phase difference, the interpolated neighbor array is shifted by the relative position/time difference and down sampled by 10 to obtain the phase corrected neighbor array. Usually, the phase correction algorithm requires two long input arrays to obtain accurate phase difference result. In our implementation, the length of the input arrays is chosen to be 1024 because this is an empirically good tradeoff between the precision and runtime performance [76].

3.3.4 Temporal Median Filter

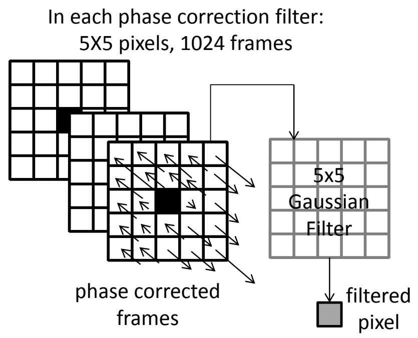
The temporal median filter is applied at the end to further remove noise after the phase correction spatial filter. The temporal median filter replaces each pixel with the median value of its temporal neighbors within a 7-element tap. After filtering, the image is conditioned and ready for analysis.



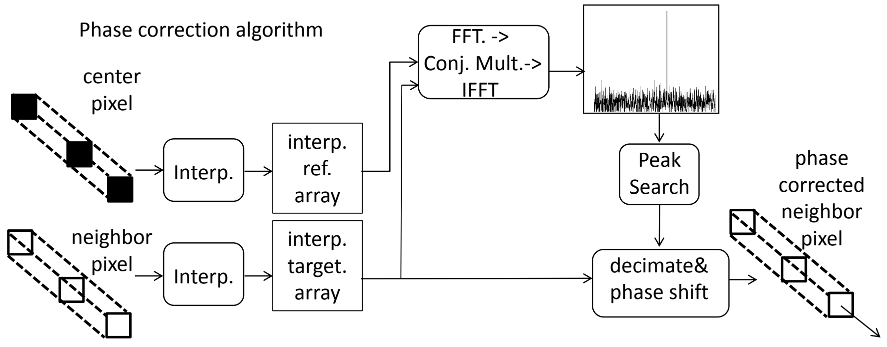
(a)



(b)



(c)



(d)

Figure 3.2. Optical mapping algorithm. (a) Overview of the image conditioning algorithm. (b) Visualization of the rolling spatial phase correction filter on the entire video data. (c) Visualization of a phase correction spatial filter window. Arrows on the pixels represent phase shifting (correction). (d) Visualization of the phase correction algorithm.

3.4 Application Partitioning

Partitioning a high throughput video application requires careful analysis at design time. Our initial design was to accelerate the software version of the algorithm developed by Sung *et. al* [76] using a FPGA. However, the algorithm operates on a second's worth of captured data at a time. This became problematic for our FPGA as the phase correction FFT would need to support a length of 32 K (1024 frames, interpolated to 10,240 frames, then padded out to 32 K frames). A single FFT core of this size would consume nearly all the resources of our FPGA. Piecewise execution of the FFT was considered, but was quickly discarded in favor of using a GPU.

Using a GPU matched well with the large array and massively parallel operations. But the frame interpolation and peak search computations are data flow barriers in the algorithm. This causes poor GPU performance. This phenomenon is discussed in [88], where a GPU implementation of the optical mapping algorithm achieves a rate about half as fast as real-time.

We chose instead to design a heterogenous system with both a GPU and FPGA. This allowed us to map the portions of the design that can benefit from deep pipelining and small buffers to the FPGA. Steps requiring large buffers with massively parallel operations leveraged the GPU. Finally, coordination, low throughput, and branching dominated tasks were assigned to the CPU. Table 3.1 shows our partitioning decisions.

The granularity of our partition is based largely on the algorithm blocks, illustrated in Fig. 3.2(a). In addition to the inherent strengths of different hardware in our system, the I/O bandwidth between portions of the algorithm drove many of our design decisions. Limited bandwidth interconnects can make it challenging to quickly and efficiently transfer data between the GPU, FPGA, and CPU. Thus, we attempted to move data as little as possible while matching algorithmic blocks to the most appropriate device.

Video is captured using the FPGA. The FPGA also performs frame interpolation and normalization of base values. This decision was based on the fact that we can pipeline the interpolation on the FPGA so that interpolated frames would be produced concurrently with camera input.

Our FPGA-PCIe connection is limited to a single PCIe lane (bandwidth limit of 250 MB/s). Thus we represented pixels using 8 bits of precision. However, the normalization step uses 32 bit floating point numbers. To adapt, we decomposed the normalization step into a calculation of base values and normalization of pixels. We compute the the base values on the FPGA and reordered the algorithm to perform normalization on the GPU. The reordered algorithm is equivalent to the original algorithm. However, representing pixels with 8 bits introduces errors in the result. We demonstrate that the error is tolerable in Section 3.6.3.

The FFT, conjugate multiplication, and IFFT computations run on the GPU. Massive data parallelism in each *butterfly* stage of the FFT and IFFT improves core occupancy on the GPU's SIMD architecture.

Instead of calculating the relative positions between all pixels and their neighbors, we calculate partial relative positions and use the fact that they are transitive between pixel array pairs to optimize the process. It results in reducing redundant computation by $5\times$. For I/O bandwidth reasons, we perform the peak search on the GPU, but chose to allocate the relative phase difference conversion to the CPU. The phase difference conversion is a low throughput and intensively branched process aiding the peak search. We describe this optimization in Section 3.5.3.

The final processing steps are run on the GPU: phase shifting, 2D spatial Gaussian filter, and temporal median filter. The GPU already has the interpolated frame data stored in memory at this point, so it is the obvious location to shift the pixel arrays and perform filtering.

Table 3.1. Optical mapping algorithm partition decisions.

	Interp.	Norm. Base Val.	Norm. Pixels	FFT	Conj. Mult.	IFFT	Lean Peak Search	Relative Phase Diff.	Phase Shift	Spatial Filter	Temp. Filter
Input Bandwidth (MB/s)	9.5	9.5	95.4	381	2304	2304	2304	0.009	381	38.1	38.1
Output Bandwidth (MB/s)	95.4	0.0098	381	2304	2304	2304	0.009	38.1	38.1	38.1	38.1
Accelerator Allocation	FPGA	FPGA	GPU	GPU	GPU	GPU	GPU	CPU	GPU	GPU	GPU

3.5 Design and Implementation

3.5.1 Overall System

The architecture of the system is shown in Fig. 3.3. It illustrates which portions of the optical mapping algorithm run on which hardware. The shaded boxes encapsulate computation groups. The architecture is designed to run continuously on a system with constant camera input. Thus, it runs in a pipelined fashion. Group ① runs in a pipelined stage concurrently with groups ②, ③ and ④ in a separate pipeline stage.

Camera data is captured by the FPGA at a rate of 1024 fps and up sampled (interpolated) to 10,240 fps. Frames of interpolated data and normalization base values are DMA transferred to the host workstation's GPU over a PCIe connection. This represents computation group ①. The GPU normalizes the pixels then performs a FFT, conjugate multiplication operation, and IFFT on arrays of pixels across frames (temporally). The result of this spectral processing produces large 32 K length arrays for each pixel location. The max value in each array is found using a max peak search over all the data. The output of this group ② is the relative position of the max values in each array. These relative positions are used to calculate the absolute positioning for each pixel array. This is performed on the CPU in group ③. The CPU is used because it is faster to transfer the data out of the GPU, iterate over it on the CPU and transfer it back, than to utilize only a few cores on the GPU. Once calculated, the absolute positions are sent back to the GPU where they are used to shift each array temporally. The arrays are shifted and then down sampled back to 1024 fps. The rest of computation group ④ consists of a 2D Gaussian filter and a temporal median filter to remove noise.

3.5.2 FPGA Design

FPGA processing is performed in a streaming fashion. For temporal interpolation, only 8 frames of video are buffered. This buffering is necessary for the FIR filter. The most challenging aspect of the FPGA design is keeping the FIR filter pipeline full. The pixel data arrives from the camera in a row major sequence, one frame at a time. The FIR interpolation filter operates on a sequence of pixels across frames. Each interpolated frame must be produced one pixel at a time, using the pixels from the previous frames. This means filling the FIR filter with previous values for one pixel location, capturing interpolated pixels for 10 cycles, then re-filling the pipeline with a temporal sequence for another pixel location. Most of the time is spent filling and flushing the FIR filter (80 out of every 90 cycles).

To avoid this inefficiency, we parallelized the FIR filter with 9 data paths and staggered the inputs by 10 cycles. This allows the FIR filter to produce valid output every cycle from one of the 9 data paths. The output is then used to calculate the normalization base values and both are DMA transferred to the host workstation over a PCIe connection. We used the RIFFA [89] framework to connect the FPGA to the host workstation (and thus the GPU).

3.5.3 GPU Design

We designed each component on the GPU as an individual CUDA kernel. Kernels use global memory for inter-kernel coordination and for I/O data transfer. Using multiple computation dedicated kernels can improve performance over a single monolithic kernel. The data access strategies and thread dimensions can vary from kernel to kernel to more closely reflect the computation. This results in overall faster execution of all the components.

In the design of each kernel, we fully parallelized each stage to obtain the highest

GPU core occupancy. We implemented the normalized pixel calculation, conjugate multiplication, and phase shift using straight forward element-wise parallelism. The spatial Gaussian filter and temporal median filter use window/tap-wise parallelism. We used the *cuFFT* library provided by NVIDIA to implement the 32 K element FFT and IFFT operations. The peak search is implemented as a CUDA reduction, which uses memory access optimizations such as shared memory, registers, and contiguous memory assignment.

The FFT, conjugate multiplication, IFFT, and peak search are the major components of the algorithm on the GPU. Each requires ultra-high throughput and their performance is directly related to the amount of data they must process. We were able to reduce the throughput requirements for these computations, and thus improve performance, with the aid of the CPU. To do so, we created two stages, a lean peak search and a relative phase difference conversion (RPDC) to replace the original peak search stage. The lean peak search only calculates the necessary peaks by the same reduction method used in the original peak search stage. The RPDC converts the result of the lean peak search stage to the full phase difference by using the fact that relative differences are transitive. For example, we calculate the phase difference between pixel arrays a and b, and between arrays b and c using lean peak search. Let these differences be t_{ab} and t_{bc} respectively. Then $t_{ac} = t_{ab} - t_{bc}$. This optimization reduces the throughput in the FFT, conjugated multiplication, IFFT and peak search by $5\times$. The RPDC is a low-throughput computation, dominated by branching logic. This would execute with low efficiency on the GPU's SIMD architecture. We therefore implemented the relative phase conversion stage on the CPU shown as ③ in Fig. 3.3.

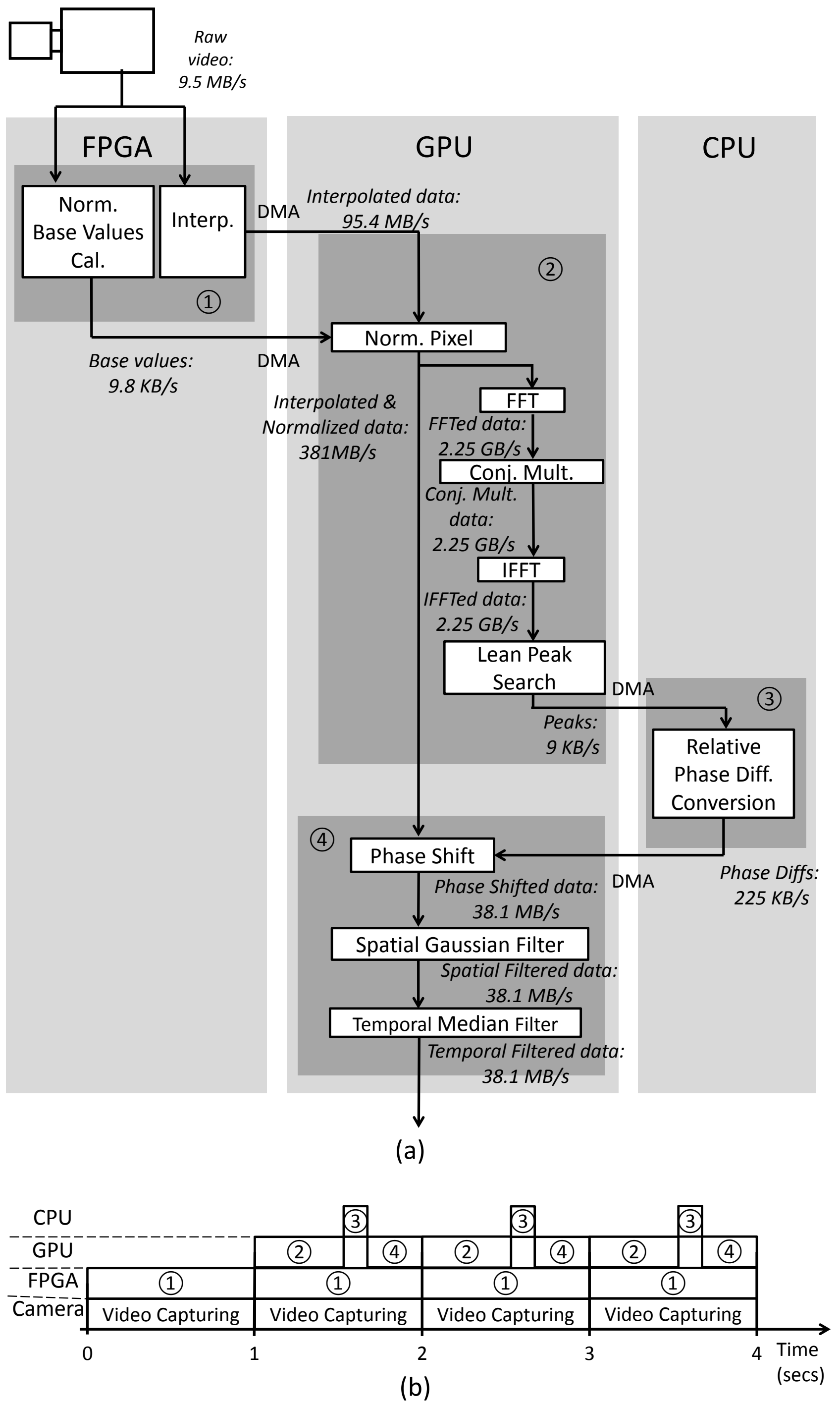


Figure 3.3. FPGA-GPU heterogenous architecture. (a) Algorithm execution diagram with throughput analysis. (b) Computation groups running concurrently in the system. Groups ①, ②, ③ and ④ are the shaded regions shown in (a).

3.6 Results and Analysis

3.6.1 Experimental Setup

We use the same experimental parameters described by Sung *et. al* [76] to guide our experiments. Input video is 100×100 resolution 8 bit grayscale video.

All our experiments are run on an Intel i7 quad-core 3.4 GHz workstation running Ubuntu 10.04. The FPGA is connected to the workstation via x1 PCIe Gen1 connector. We use a Xilinx ML506 development board with a Virtex 5 FPGA. All FPGA cores were developed using Xilinx tools, ISE and XPS, version 13.3. The GPU is an NVIDIA GTX590 with 1024 cores.

Our heterogenous design is controlled by a C++ program and compiled using GCC 4.4 and CUDA Toolkit 4.2. The C++ program interfaces with the CUDA API and the RIFFA API [89] to access the GPU and FPGA respectively. It provides simulated camera to the FPGA and coordinates transferring data to and from the FPGA and CPU/GPU.

3.6.2 Performance

Our design can execute both stages (group ① and groups ②, ③, and ④) concurrently as stage one executes on the FPGA and stage two executes on GPU/CPU. Stage one can process a second's worth of video in 0.82 seconds, at a rate of 1248 fps. However since the camera only delivers data at a rate of 1024 fps, the FPGA takes a full second to complete stage one. Transfer time is masked by pipelined DMA transfers. Thus at the end of one second, effectively all the data from stage one is in CPU memory. The GPU executes all computations in stage two in 0.86 seconds. Because an entire second's worth of data must be processed in at a time in stage two, the total latency is 1.86 seconds from the time the camera starts sending data until the time a full second's worth of processed data is available in CPU memory. This only affects latency. Both stages execute at, or faster than real-time. A video of our FPGA-GPU-CPU implementation working on captured data can be found at: http://www.youtube.com/watch?v=EfvXenkiGAA.

We compare our performance against the original serial software implementation, an optimized C++ multi-threaded software implementation, and an optimized GPU implementation in Fig. 3.4.

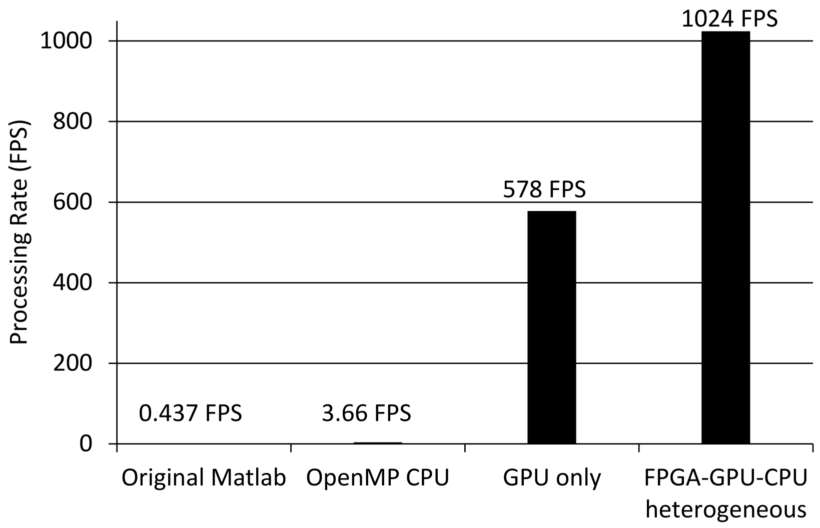


Figure 3.4. The performance of the FPGA-GPU-CPU heterogenous implementation in comparison to the original Matlab, the OpenMP C++, and the GPU only implementation.
.

The original serial software implementation was designed and published by Sung *et. al* [76]. The authors did not provide execution times for a full second's worth of data. However, running the same software on our i7 workstation takes 39 mins for one second's worth of data. To attempt a more fair comparison that uses all the cores of a modern workstation, we implemented an optimized C++ version (with the same algorithm implemented on the heterogeneous system). This version uses the OpenMP API to parallelize portions of the application across multiple cores. The optimized C++ program also used direct access tables to avoid computation such as trigonometric functions and the FFT output indices. This implementation took 4.6 mins to perform the same task. This is equivalent to 3.66 fps. We feel that this is an appropriate baseline for a software comparison. Our FPGA-GPU-CPU design runs 273× faster that an optimized C++ software version.

An optimized GPU implementation is described fully in [88]. It represents months of optimization tuning. It performed at a respectable rate of 578 fps. But it would have to be nearly twice as fast to achieve real-time performance. Additionally, like all the other implementations except the FPGA-GPU-CPU implementation, it would require the use of a frame capture device to be used in any real world scenario. This is a detail often overlooked when comparing performance.

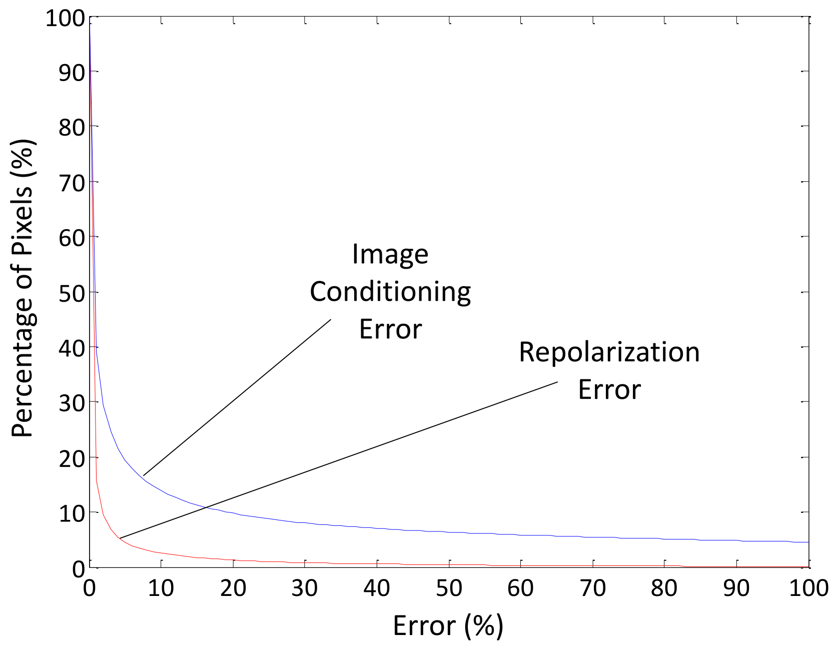


Figure 3.5. Error of the output of the optical mapping image conditioning (blue line) and error in repolarization analysis (red line). For any point (x, y) on the curve, the x represents the error in percentage scale while the y represents the percentage of pixels whose errors are greater than x.

3.6.3 Accuracy

As described in Section 3.4, the 8 bit representation of pixels (instead of 32 bit) introduces errors to the result. Algorithmic parameters limit processing of pixel arrays to those with values above 60 and with variance above 2. This limits the amount of error any one pixel can incur to 0.83 % when using 8 bits instead of 32 and rounding to the nearest integer. However the normalization base value may be arbitrarily close to to any pixel value. Therefore, the normalized error for any pixel is unbounded. Indeed, this is evident in Fig. 3.5. Some of the pixel locations show relatively significant errors. For example, about 13.8% of pixels have error greater than 10%. In practice however, we show that this is not as significant to the medical analysis.

We applied the repolarization extraction algorithm described in [76] on both the FPGA-GPU-CPU and baseline CPU implementation outputs. Fig. 3.5 shows the repolarization error. This error is significantly lower than the image conditioning error. Only 2.6% of the repolarization analysis have error greater than 10%. This result indicates that using an 8 bit representation of interpolated pixels only slightly impacts biomedical features that would be extracted from the output.

3.7 Summary

We have addressed the challenge of real-time optical mapping for cardiac electrophysiology and presented a heterogeneous FPGA-GPU-CPU architecture for use in medical applications. Our design leverages the stream processing of a FPGA and the high bandwidth computation of a GPU to process video in real-time at 1024 fps with an end to end latency of 1.86 seconds. This represents a 273$\times$ speedup over a multi-core CPU OpenMP implementation. We also discussed our partitioning strategy to achieve this performance.

Acknowledgements

This chapter contains material printed in “FPGA-GPU-CPU Heterogeneous Architecture for Realtime Cardiac Physiological Optical Mapping”, Pingfan Meng, Matthew Jacobsen, and Ryan Kastner, the International Conference on Field-Programmable Technology (FPT) 2012. The dissertation author was the primary investigator and author of this paper.

Part II

Automatic Approaches with Machine Learning

Chapter 4

Re-thinking Machine Learning for OpenCL Design Space Exploration on FPGAs

4.1 Background and Motivation

FPGAs have demonstrated multiple advantages as a heterogeneous accelerator in many different high performance computing tasks [28, 7, 90, 91, 92, 93]. One of the greatest challenges to implement FPGA accelerated systems is the design complexity due to the nature of hardware design. Recently, system-level (or high-level) synthesis tools have effectively addressed this challenge by replacing the register-transfer level (RTL) design technique with software languages (C++, OpenCL) [42, 43, 94, 95, 96, 97, 98]. The adoption of OpenCL is especially appealing since the language has been widely utilized to program other heterogeneous accelerators such as multi-core CPUs and GPUs. With the state of the art OpenCL synthesis tools, heterogeneous system designers are now able to achieve a longtime desired goal – to program different devices with a single unified language.

Although the OpenCL-to-FPGA tools have significantly reduced the design complexity, one major bottleneck still exists in design space exploration (DSE) due to the extremely long synthesis runtime of these tools. The compilation process of the software

OpenCL code running on the processors (CPUs or GPUs) usually only consumes several milliseconds. In contrast to the software compilation, implementing an OpenCL design on the FPGA usually requires multiple hours for a high performance workstation to complete the hardware synthesis (including place and route) process. Moreover, the size of the design space grows exponentially with the number of the tunable parameters in the design. For example, the designer can generate thousands of different designs for a simple matrix multiplication by tuning the parameters in the OpenCL code. Synthesizing all of these designs (demonstrated in Fig. 4.1) for the brute force DSE is impractical since the process consumes thousands of computing hours.

One solution to this challenge is the machine learning technology which predicts the Pareto-frontier ("good" designs) based on a small training sample. The small training sample reduces the required synthesis time significantly. However, using machine learning is risky. Machine learning may incorrectly exclude some actual Pareto designs in its predicted output. Most of the existing machine learning frameworks attempt to accurately model how the design objective functions respond to the tunable parameters in the designs. This track of effort does not enhance the eventual goal of seeking the "good" designs.

To address this issue, we re-thought how to use machine learning for system level DSE on FPGAs. We propose a framework - *Adaptive Threshold Non-Pareto Elimination* (ATNE) which takes a fundamentally different track from the other existing attempts. **Instead of focusing on improving the conventional accuracy of the learner, our work focuses on understanding and estimating the risk of losing "good" designs due to learning inaccuracy. The goal of ATNE is to minimize this risk.**

To the best of our knowledge, our work is the first attempt to investigate machine learning aided system level (or high-level) DSE with **real performance data from end-to-end (the synthesis process includes the place and route stage) applications**

running on the FPGA.

The rest of this chapter is organized as follows. We review the state of the art machine learning frameworks for hardware DSE problems in section 4.2. We formulate the problem in section 4.3. In section 4.4, we provide the theoretical foundation for ATNE. In section 4.5, we describe the ATNE algorithm. We report the evaluation results in section 4.6. We summarize the chapter in section 4.7.

4.2 Related Work

In recent years, researchers have been applying machine learning algorithms on hardware DSE problems such as IP core generation[99] and timing results[100]. The high-level design is processed by more layers of the design tool-chain. Moreover, high-level designs are usually written in “software-like” styles. It is more difficult to predict how a high-level parameter would affect the low-level architecture. Therefore, applying machine learning on high-level DSE is a different task than those addressed in the existing low-level tuning frameworks.

Researchers have also attempted to use machine learning to aid high-level DSE. Liu and Carloni proposed a framework using experimental design in [65]. This type of framework focuses on seeking the sampling that describes the design space accurately. The second type of the existing frameworks are uncertainty sampling based. Zuluaga *et al.* proposed an active learning algorithm that iteratively samples the design which the learner cannot clearly classify in [64].

Most of these existing frameworks focus on how to learn an accurate model to describe the design space. The learning accuracy improves the quality of the models that describe how the design space responds to specific tuning parameters. However, this effort may still not be enough to improve the probability of correctly identifying the Pareto-set, which is the ultimate goal of DSE.

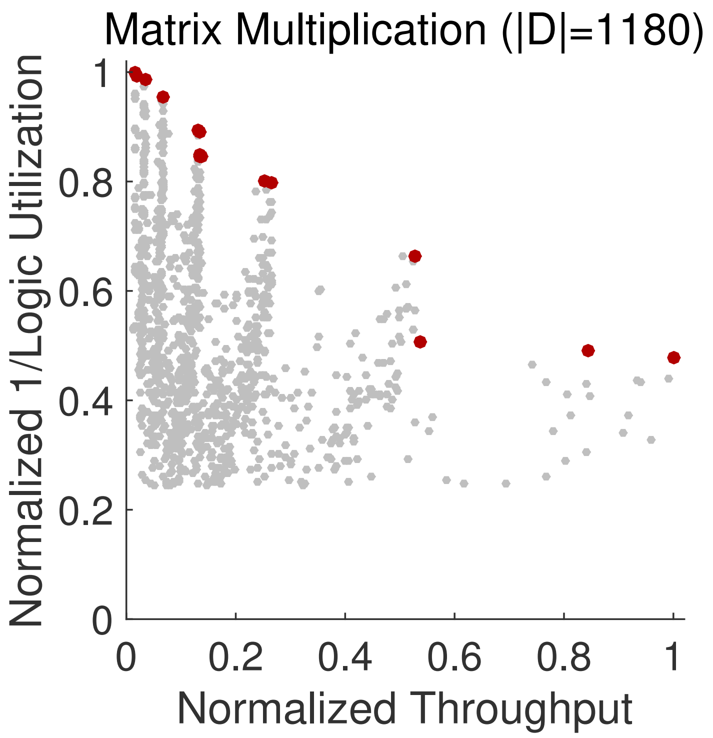


Figure 4.1. The visualization of the design space of the matrix multiplication application in the objective space using the OpenCL-to-FPGA tool. The red dots are the Pareto-frontier.

Our approach explores a fundamentally different track from these existing works. Instead of further improving the learning accuracy, we chose to understand and estimate the inaccuracy of the learner. Based on the estimated inaccuracy, we adaptively choose a threshold to improve the quality of Pareto design identification.

4.3 Problem Formulation

In this section, we firstly introduce the concepts of the design space and the Pareto-frontier. Then we formulate the problem that our framework tackles.

Design space and objective space: We refer to the programming choices in the OpenCL code as tunable *knobs*. The examples of knobs and their impacts on the hardware architecture are listed in Table 4.1.

Table 4.1. Tunable Knobs in the OpenCL Code and Their Impacts on the Hardware Architecture

Number	OpenCL Knob	Impact on HW Arch.
1	Num. SIMD	SIMD parallel width & mem. BW
2	Num. comp. unit	Num. parallel ctrl. & mem. BW
3	Unroll factor	module replic. mem. access pattern
4	Local buffer height and width	BRAM partition & access pattern
5	Num. private variables	Num. registers
6	Vectorization Width	SIMD parallel width
7	Num. work-items per group	ctrl. logic
8	Data size per work-item	Data locality

For a given application which owns K knobs in the OpenCL code, we use denotation $\mathbf{x} = (x_1,x_2,x_3,...,x_K)$ to refer to a knob setting. All possible knob settings for a given OpenCL code form $D = \{\mathbf{k}\}$ (called design space). The designers usually evaluate the FPGA designs with multiple objectives (e.g. throughput, logic utilization) which can be represented by functions $\phi(\mathbf{x}) = (\phi_1(\mathbf{x}),\phi_2(\mathbf{x}),\cdots,\phi_M(\mathbf{x}))$: $D \mapsto \mathbb{R}$. Then the objective space is the image of the design space: $\phi(D) \subset \mathbb{R}^M$. Fig. 4.1 demonstrates the design space image in the objective space of the matrix multiplication application.

Pareto-frontier: In DSE, the goal is to make trade-offs on the Pareto-frontier ("good" designs). Here, we provide the formal definition of the Pareto-frontier. We use canonical partial order "$\prec$" in $\mathbb{R}^M$ to represent one design is inferior in all objectives to another: $\phi(\mathbf{x}) \prec \phi(\mathbf{x}')$ iff $\phi_i(\mathbf{x}) < \phi_i(\mathbf{x}')$, $1 \leq i \leq M$. The Pareto-frontier $P \subseteq D$ is defined as: $P = \{\mathbf{x} \in D \mid \forall \mathbf{x}' \in D, \phi(\mathbf{x}) \succeq \phi(\mathbf{x}')\}$. The red dots in Fig. 4.1 demonstrate the Pareto-frontier of the matrix multiplication design space.

Problem Formulation: *Given an OpenCL application, predict P synthesizing the minimal number of designs in D.*

Prediction quality: We use *average distance from reference set*(ADRS)[101] as the metric to evaluate the prediction quality of the framework. ADRS is defined as :

$$ADRS(P_{gt}, P_{pred}) = \frac{1}{|P_{gt}|} \sum_{\mathbf{p}_{gt} \in P_{gt}} \min_{\mathbf{p}_{pred} \in P_{pred}} d(\mathbf{p}_{gt}, \mathbf{p}_{pred}),$$

where

$$d(\mathbf{p}_{gt}, \mathbf{p}_{pred}) = \max_{m=0,1,\ldots M} \left(0, \frac{\phi_m(p_{gt}) - \phi_m(p_{pred})}{\phi_m(p_{gt})}\right).$$

4.4 Re-thinking Machine Learning for System Level FPGA DSE

In this section, we firstly reveal a pitfall in the existing machine learning approaches for FPGA DSE. Then we provide some theoretical results as the foundation of ATNE.

4.4.1 Pitfall: attempting to learn more accurately

The main effort of most existing machine learning frameworks in hardware DSE focuses on how to regress the design objective functions more accurately. Improving the regression accuracy certainly generates more precise models to describe the objective function. However, this strategy has a pitfall in the further Pareto design prediction.

Here, we provide empirical results using the matrix multiplication example to demonstrate this pitfall. We used *transductive experimental design* (TED) technology [102] to sample the design space of an OpenCL matrix multiplication code on the FPGA. We used the random forest (RF) algorithm to learn two objective functions – performance and logic resource utilization. We directly use the regressed functions to identify the Pareto designs (i.e. those predicted as inferior to at least one design are considered non-Pareto). We measure two metrics: (1) mean square error (MSE) to illustrate the regression accuracy of the objective functions; (2) percentage of $\mathbf{x}$ such that $\mathbf{x} \in P_{gt}$ and $\mathbf{x} \notin P_{pred}$ to illustrate how many actual Pareto designs are not selected by the learner. We

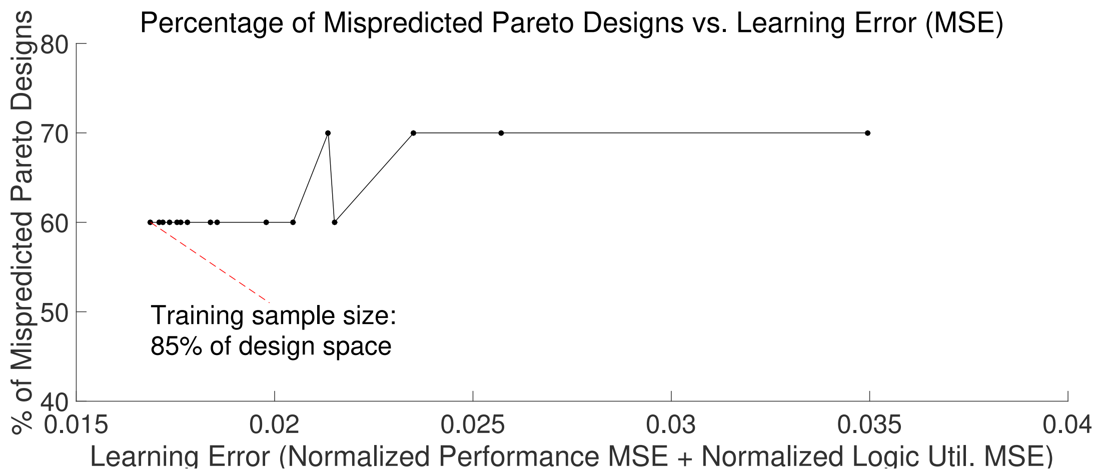


Figure 4.2. Demonstration of the pitfall of improving learning accuracy. Improving the MSE does not improve the misprediction of Pareto designs.

tested multiple training sample sizes (5%, 10%, 15% ... 85% of the entire design space) to generate regressed functions with different accuracies.

As shown in Fig. 4.2, improving the regression error does not necessarily improve the correctness of the Pareto prediction. Moreover, even when we sampled 85% of the design space to train the learner, the learner still mispredicts approximately 60% of the Pareto points. **This means if we follow the conventional track to focus solely on the learning accuracy, the designer may still miss more than half of the "good" designs!**

4.4.2 Re-thinking: understanding the error from the learner

We investigated how the learner inaccuracy affects the final Pareto design prediction quality. We still use RF as an example for the regresser. The RF takes several designs as a training sample to learn the design objective functions. Let $\hat{\phi}$ denote the design objective function vector regressed by the RF. Let T denote the number of decision trees in the RF. The RF algorithm produces the final output by computing an arithmetic mean of the outputs of all the trees:

$$\hat{\phi} = \frac{1}{T}\sum_{t=1}^{T}\hat{\phi}_t.$$

Each tree of the forest is trained on an independently and randomly selected bootstrap of the training data. The outputs of all the trees should independently have the same distribution. Therefore, according to the central limit theorem, the output of the RF is normally distributed (even though we specifically selected RF as the learner here, our theoretical result still stands as long as the output of the learner is an arithmetic mean of i.i.d. random variable). Since the difference between two normally distributed random variables is still normally distributed, for any two designs $\mathbf{x}$ and $\mathbf{x}'$, the difference $\hat{\phi}(\mathbf{x}) - \hat{\phi}(\mathbf{x}')$ is also normally distributed.

With this reasoning, we can describe $\hat{\phi}(\mathbf{x}) - \hat{\phi}(\mathbf{x}')$ as a normally distributed random variable when we apply RF to regress the objective functions of a design space. The expectation and the variance of this random variable are determined by the training data and how we train the forest (the parameter setups of the RF). With the aid of this mathematical model, we can analyze how the regressor (RF) affects the Pareto prediction.

Assuming we use the regressed objective functions to identify the Pareto designs, the conventional threshold is:

$$\mathbf{x} \notin P_{gt}, \text{ if } \exists \mathbf{x}' \in D \text{ s.t. } \hat{\phi}(\mathbf{x}) \prec \hat{\phi}(\mathbf{x}').$$

The cumulative distribution function (CDF) of $\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x})$ is demonstrated in Fig. 4.3. Assuming $\mathbf{x}$ is incorrectly predicted as inferior to $\mathbf{x}'$ by the RF, the expectation of $\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x})$ should be greater than 0. According to the monotonicity of the normal distribution CDF, we can show that $Pr[\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x}) \leq 0]$ is less than 50%. If variance of $\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x})$ decreases, this probability of correcting the misprediction becomes even lower (shown by the "dot" curve). Therefore, training a more stable learner may

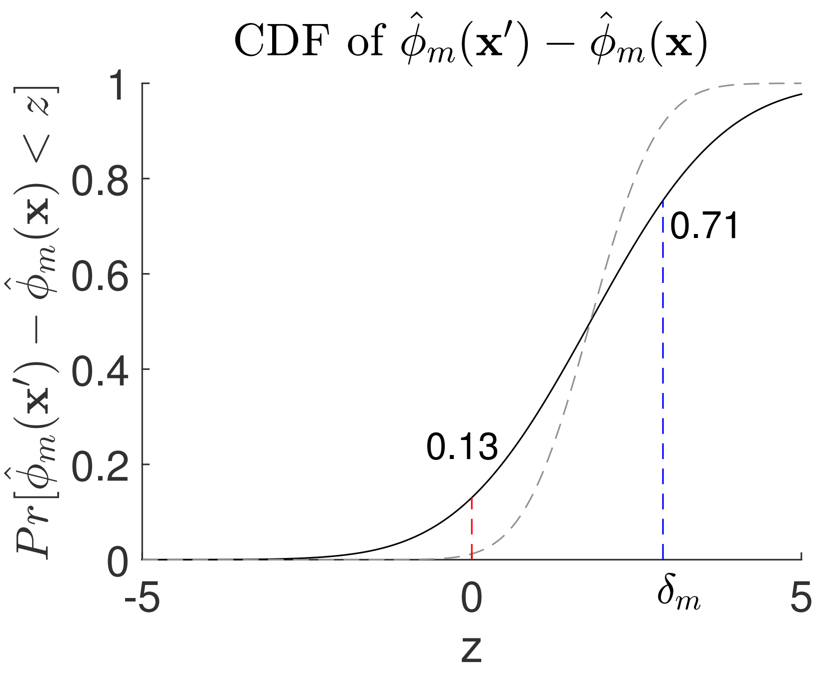


Figure 4.3. CDF of $\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x})$

decrease the probability for the actual Pareto design to be identified.

An apparent solution to this issue is to lift the threshold from 0 to $\boldsymbol{\delta}$ (a vector of thresholds for multiple design objectives):

$$\mathbf{x} \notin P_{relaxed}, \text{ if } \exists \mathbf{x}' \in D \text{ s.t. } \hat{\phi}(\mathbf{x}) + \boldsymbol{\delta} \prec \hat{\phi}(\mathbf{x}')$$

With this $\boldsymbol{\delta}$, the probability for the learner to think an actual Pareto design $\mathbf{x}$ is not inferior to $\mathbf{x}'$ becomes $Pr[\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x}) \leq \delta_m]$. Due to the monotonicity of the CDF, this new probability is definitely greater than $Pr[\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x}) \leq 0]$. However, lifting this threshold $\boldsymbol{\delta}$ increases the probability for the framework to output those non-Pareto ("bad") designs. For this reason, we name this output set $P_{relaxed}$ meaning "relaxed" Pareto-set. An extreme case of this side-effect is all non-Pareto designs are selected into this $P_{relaxed}$ by lifting the threshold beyond the necessity. Then the framework outputs the entire design space which is equivalent to the brute force approach. Therefore, in order to predict the Pareto designs efficiently, the framework requires an appropriate δ.

We describe this appropriate δ in Theorem 1.

Let μ_m and σ_m denote the expectation and variance of $\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x})$ respectively. Let erf^{-1} denote the inverse Gauss error function (Gauss error function is used in the expression of normal distribution CDF function). We describe our theoretical result as:

Theorem 1 *The m component of the minimal threshold* $\boldsymbol{\delta}$ *to guarantee* $Pr[\boldsymbol{x} \in P_{relaxed} | \boldsymbol{x} \in P_{gt}] \geq \alpha$ *is*

$$\delta_m = \max_{\forall \boldsymbol{x} \in P_{gt},\ \forall \boldsymbol{x}' \in D} (\mu_m + \sigma_m \sqrt{2} erf^{-1}(2\alpha - 1)).$$

The proof of Theorem 1 is straightforward – use the quantile formula of the normal distribution and its monotonicity. We describe how to practically estimate this minimal threshold in lines 8 to 17 of Algorithm 4.1.

4.5 ATNE Algorithm

We describe ATNE in Algorithm 4.1. ATNE actively samples data and iteratively eliminates the non-Pareto designs. Although ATNE uses two existing technologies, active learning and experimental sampling, we applied these technologies with a completely novel strategy. The major novelty of the ATNE algorithm is the three stages (lines 8 to 33 in Algorithm 4.1): (1) estimate $\boldsymbol{\delta}$, (2) elimination and (3) active sampling. The stages initial sampling and model learning only serve as the starting point for the other three stages. We describe each stage in detail in the rest of this section.

Initial Sampling: At the initial state, we use TED to sample the designs purely based on the knob setting information. TED has been proven effective for this task in [65]. This initial stage only serves as a starting point for the $\boldsymbol{\delta}$ estimation and elimination stages. We only sample 6 designs (minimum for the first round of random forests regression).

Algorithm 4.1: The ATNE Algorithm

```
Input  : Design space $D$; Number of initial samples $S_{init}$; Target final size of
         $P_{relaxed}$: $S_{final}$; Target correctness probability $\alpha$; Number of forests $F$;
         Minimal number of candidates for $\delta$ estimation: $C$
1  $P_{relaxed} = D, L = \emptyset$
   ———————— Initial Sampling ————————
2  initially sample the designs $L$=TED$(D, S_{init})$
3  while $|P_{relaxed}| < S_{final}$ do
4      set of designs to be eliminated $E = \emptyset$
       ———————— Model Learning ————————
5      for $f = 1$ to $F$ do
6          regress $\hat{\phi}_f$ on $L$ using random forest $f$
7      end
       ———————— Estimate $\delta$ ————————
8      $\Delta_{candi.} = \emptyset$
9      for $\forall \boldsymbol{x}$ and $\forall \boldsymbol{x}' \in P_{relaxed} \cap L$ do
10         $\hat{\mu}_{\mathbf{x},\mathbf{x}'} = \frac{1}{F} \sum_{f=1}^{F} \hat{\phi}_f(\mathbf{x}') - \hat{\phi}_f(\mathbf{x})$
11         $\hat{\sigma}_{\mathbf{x},\mathbf{x}'} = \sqrt{\frac{1}{F} \sum_{f=0}^{F} (\hat{\phi}_f(\mathbf{x}') - \hat{\phi}_f(\mathbf{x}) - \hat{\mu}_{\mathbf{x},\mathbf{x}'})^2}$
12         $\delta_{\mathbf{x},\mathbf{x}'} = \hat{\mu}_{\mathbf{x},\mathbf{x}'} + \hat{\sigma}_{\mathbf{x},\mathbf{x}'} \sqrt{2} erf^{-1}(2\alpha - 1)$
13         if $\hat{\mu}_{x,x'} > 0$ and ground truth $\boldsymbol{x}$ is superior to $\boldsymbol{x}$' then
14             $\Delta_{candi.} = \Delta_{candi.} \cup \{\delta_{\mathbf{x},\mathbf{x}'}\}$
15         end
16     end
17     $\delta = \max(\Delta_{candi.})$
       ———————— Elimination ————————
18     $E = \emptyset$
19     for $\forall \boldsymbol{x}$ and $\forall \boldsymbol{x}' \in P_{relaxed}$ do
20         if $\hat{\phi}_f(\boldsymbol{x}') - \hat{\phi}_f(\boldsymbol{x}) \prec \delta$ and $|\Delta_{candi.}| \geq C$ then
21             $E = E \cup \{\mathbf{x}\}$
22         end
23     end
24     $P_{relaxed} = P_{relaxed} - E$
       ———————— Active Sampling ————————
25     sample the design $\hat{\mathbf{x}}$ that is most difficult to eliminate in $P_{relaxed}$
26     synthesize $\hat{\mathbf{x}}$, $L = L \cup \{\hat{\mathbf{x}}\}$
27 end
28 synthesize designs in $P_{relaxed} - (P_{relaxed} \cap L)$
29 $P_{pred}$ = brute force Pareto-frontier search on $P_{relaxed} \cup L$
Output : predicted Pareto set $P_{pred}$
```

Model Learning (Regression): The random forest is suitable for the discrete nature of the design knob settings. Therefore, we chose RF for regression. As described in lines 5 to 7 of Algorithm 4.1, we train multiple RFs with the sampled designs instead of a single forest. Using these trained forests, we can estimate the μ and σ of the distribution of $\hat{\phi}_m(\mathbf{x}') - \hat{\phi}_m(\mathbf{x})$ for the next stage. Also, the regressed objective functions are used for eliminating the non-Pareto designs.

Estimate $\boldsymbol{\delta}$: Estimating an appropriate $\boldsymbol{\delta}$ aids the framework to find the Pareto designs accurately and efficiently. The theoretical minimal $\boldsymbol{\delta}$ is provided by Theorem 1. However, in reality, the framework cannot obtain the real distribution as in the theorem. Therefore, we use the sampled data to estimate this $\boldsymbol{\delta}$. We compute the approximate μ and σ from the regressed results of the F forests as shown in lines 10 to 12 of the algorithm. Among all the sampled data, we only collect those "under-estimated" (i.e. the forests predict this design is inferior to another design when in fact the ground truth design is non-inferior, as shown in line 13) ones to compute the $\boldsymbol{\delta}$ candidates. In order to maximize the probability $Pr[\mathbf{x} \in P_{relaxed} | \mathbf{x} \in P_{gt}]$), we then select the greatest candidate to be the estimated $\boldsymbol{\delta}$.

Elimination: The elimination happens only when the number of $\boldsymbol{\delta}$ candidates is greater than the parameter C (shown in line 20). This prevents the case when the framework does not have enough ground truth data to estimate $\boldsymbol{\delta}$. When the framework has enough ground truth to estimate $\boldsymbol{\delta}$, it eliminates the designs that meet the criteria $\hat{\boldsymbol{\phi}}_f(\mathbf{x}') - \hat{\boldsymbol{\phi}}_f(\mathbf{x}) \prec \boldsymbol{\delta}$. As discussed earlier, using this criteria, the algorithm is unlikely to eliminate the ground truth Pareto designs. The size of $P_{relaxed}$ iteratively decreases. The transition of this elimination process is visualized in Fig.4.4. As shown in this example, the eliminated designs are mostly non-Pareto in each iteration.

Active Sampling: The traditional active sampling aims to provide the data to refine the learner [103, 64]. However, as discussed in Section 4.4, this traditional strategy

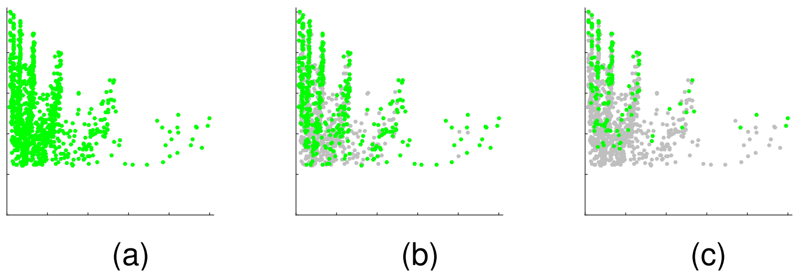


Figure 4.4. Transition of eliminating the predicted non-Pareto designs. The grey dots represent the eliminated designs (set E). The green dots represent set $P_{relaxed}$). (a) initial state; (b) iteration = 15; (c) iteration = 80.

may not serve the DSE goal effectively. In contrast to traditional active sampling, our strategy in ATNE focuses on mitigating the side-effect that $\boldsymbol{\delta}$ increases $Pr(\mathbf{x} \in P_{pred} | \mathbf{x} \notin P_{gt})$. We select the design that has the highest $Pr(\mathbf{x} \in P_{pred} | \mathbf{x} \notin P_{gt})$. Therefore, in order to eliminate this design quickly, we synthesize it to enable its elimination by using the ground truth data.

4.6 Results

In this section, we firstly describe the experimental setup. Secondly, we compare the prediction quality of ATNE against the other state of the art frameworks.

4.6.1 Experimental Setup

We selected 8 OpenCL applications: matrix multiplication (MM), Sobel filter, FIR filter, histogram (HIST), discrete cosine transform (DCT), breadth-first search (BFS), sparse matrix-vector multiplication (SpMV), and the Needleman-Wunsch (NW) algorithm. For BFS and SpMV, we used two different input data sets. Therefore, we created 10 benchmarks in total. The synthesis processes of these applications include the place and route stage. Collecting the ground truth data of all 10 design spaces costs us more than 10000 computing hours (5 months using 3 high-end 8-core workstations). We

Table 4.2. Details of OpenCL Benchmarks

Application	Domain	Specification	Tuning knobs
MM	matrix operation	1024×1024	1 - 8
Sobel	sliding window	1080p	1,2,4-8
FIR	sliding tap	128 taps	1,2,4,5,7,8
HIST	global sum	256 bins	2-5,7,8
DCT	transformation	8×8 block	1 - 8
BFS	graph operation	sparse and dense graph	1 - 8
SpMV	sparse matrix	density: 0.5% and 50%	2 - 8
NW	dynamic programming	string length: 512	1 - 8

Tuning Knobs: 1) #SIMD, 2) #Units, 3) Unroll Factor, 4) Local Memory size, 5) #Registers, 6) Vectorization Width, 7) #Work-items, 8) Data Size per Work-item

use these ground truth data to evaluate the prediction quality of ATNE.

These benchmarks represent several common computation patterns in high performance computing. We also applied multiple widely used OpenCL programming techniques. We tuned multiple OpenCL parameters, such as blocking size, SIMD width, and local array size to generate a considerable design space for each benchmark. The details of these benchmarks are listed in Table 4.2.

The OpenCL-to-FPGA tool we used is Altera OpenCL SDK 14.1. The experimental board is Terasic DE5-net with an Altera Stratix V FPGA. We choose the two most important design metrics – throughput and logic utilization (ALMs) as the learning objectives.

We configure the parameters of ATNE as follows: we set $S_{init} = 6$ (only as a starting point, the main sampling process relies on the active learning stage of ATNE); we set the target size $S_{final} = 5$ as a reasonable synthesizing time. We empirically set the parameter C to 12. We set the number of forests $F = 10$. We ran ATNE with multiple α's to evaluate the prediction qualities of ATNE for different synthesis costs. The RF algorithm we used is the Matlab MEX version converted from the Fortran version designed by Breiman [104]. We configured the number of trees to 50 and the bootstrap coefficient to 0.37 for the RF.

4.6.2 Prediction Quality

Fig. 4.6 visualizes the Pareto prediction results produced by ATNE. The synthesis complexities required to produce these results are also listed in Fig. 4.6. The framework identified most of the Pareto designs as shown in the figure.

We quantify the prediction quality of ATNE using ADRS calculated by the normalized ground truth data. We compare the ADRS qualities of ATNE against the other two state of the art frameworks PAL[64] and TED[65]. As shown in Fig. 4.7, we report the ADRS vs. sampling complexity of all three frameworks. From the figure, it is obvious that ATNE outperforms the other two frameworks. More specifically, we set the accuracy threshold ADRS≤ 0.01 meaning the predicted Pareto points are inferior to the ground truth Pareto points no more than 1% for any design objective. For a real-world design task, this means we find a design that is almost the optimal. The results in Fig. 4.7 indicate that our ATNE framework achieves less prediction errors with lower sampling complexities for most of the benchmarks. Especially for FIR, ATNE achieves ADRS≤ 0.01 with a significantly low sampling complexity while the other two frameworks are incapable of achieving such a prediction quality. In addition, for multiple benchmarks other than FIR, TED and PAL are incapable of achieving ADRS≤ 0.01. In contrast to TED and PAL, ATNE reaches this ADRS threshold for all benchmarks. Since all 10 benchmarks have very different design spaces (as shown in Fig. 4.6), this result indicates the robustness of ATNE for the DSE problem of multiple types of applications.

Moreover, ATNE significantly reduces the sampling complexity. The minimal

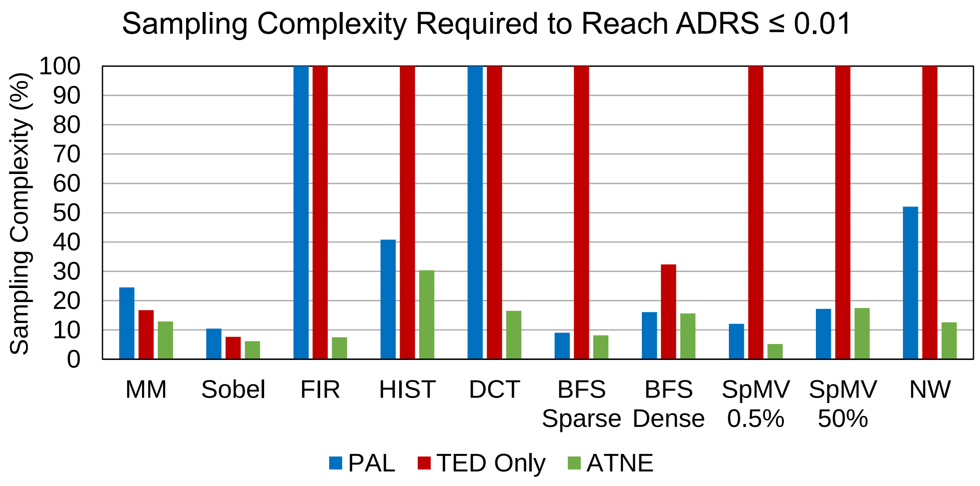


Figure 4.5. Sampling complexity to achieve ADRS$\leq$ 0.01. We let the complexity equal 100% if the method is incapable of achieving ADRS$\leq$ 0.01, e.g. PAL for benchmark FIR.

sampling complexities required to reach ADRS$\leq$ 0.01 are reported in Fig. 4.5. In comparison with PAL and TED, ATNE reduces the sampling complexity by 1.29$\times$, 1.24$\times$, 1.34$\times$, 1.11$\times$, 1.02$\times$, 2.30$\times$ and 4.12$\times$ for MM, Sobel, HIST, BFS Sparse, BFS Dense, SpMV 0.5% and NW respectively. For FIR and HIST, TED and PAL are incapable of achieving ADRS$\leq$ 0.01. Therefore, for these two benchmarks, we compare ATNE against the brute force method, i.e. 100% complexity. Only for SpMV 50%, ATNE consumes 0.02$\times$ more sampling complexity than PAL does. On average, ATNE reduces the sampling complexity by 3.28$\times$ against PAL and TED. Since each synthesis run takes hours, these speedups could save hundreds of computing hours for the designers.

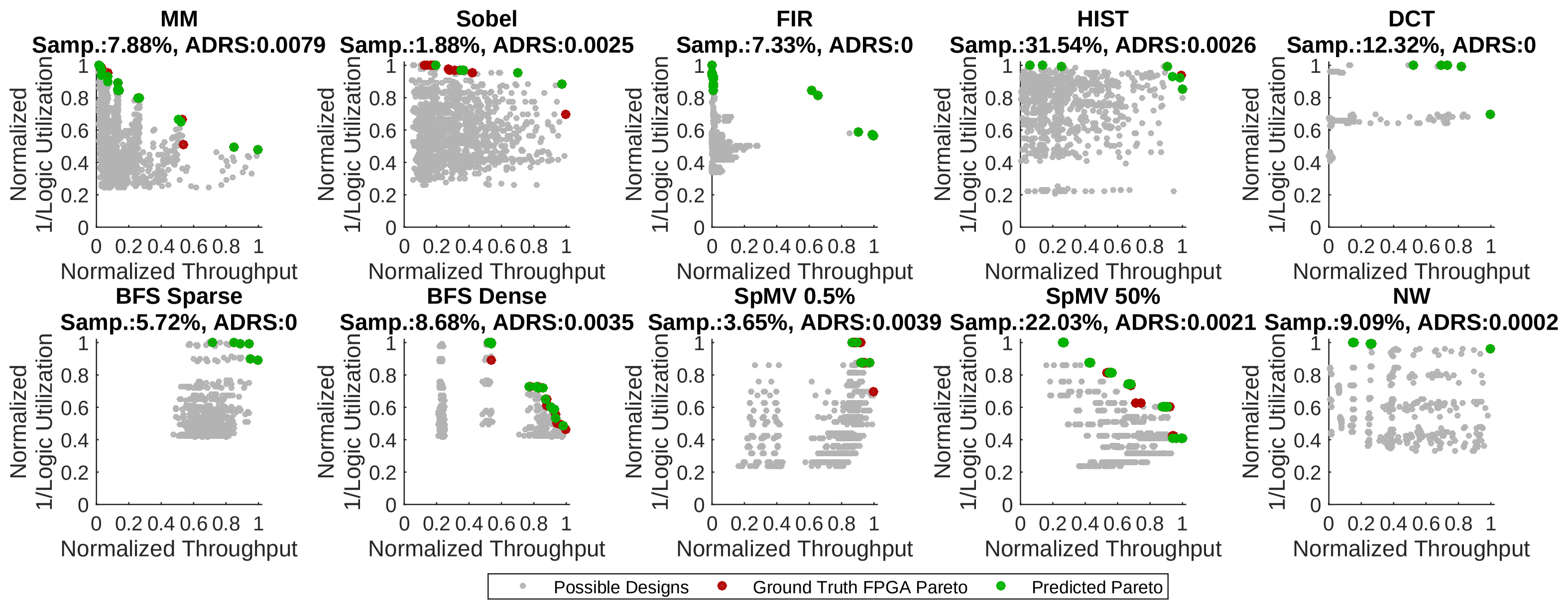


Figure 4.6. Demonstration of predicted Pareto Points. The Pareto prediction results produced by ATNE with ADRS$\leq$ 0.01 are demonstrated. The green dots represent the predicted Pareto designs. The red dots represent the ground truth designs that are not identified by the framework.

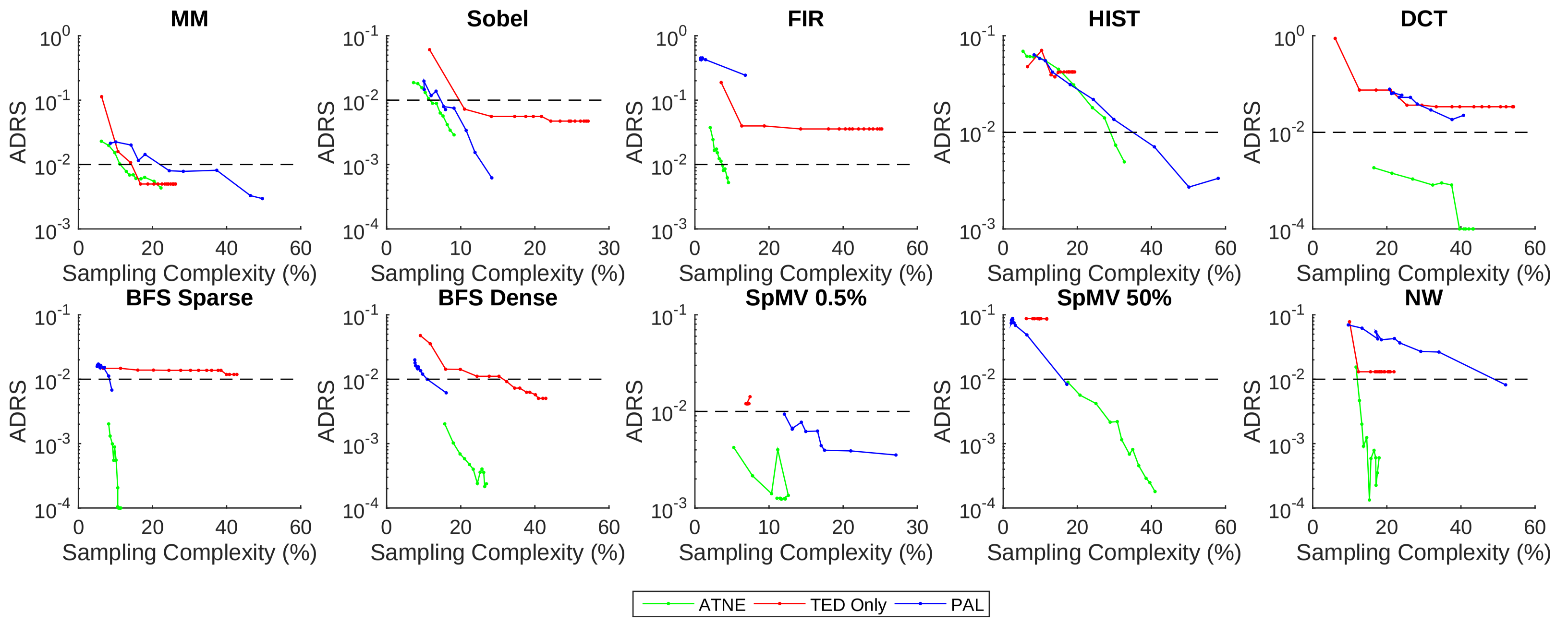


Figure 4.7. The comparison between prediction qualities (ADRS) of ATNE and the other state of the art approaches PAL[64], TED only[65]. We ran ATNE with α from $1-0.4$ to $1-0.4\times0.5^{11}$. The vertical axes are in logarithmic scale.

4.7 Summary

In this chapter, we presented a machine learning framework – ATNE for the design space exploration on the OpenCL-to-FPGA tool. ATNE applies a novel strategy of machine learning on the system level FPGA DSE task. We evaluated the effectiveness of ATNE using 5 end-to-end OpenCL applications running on the actual FPGA board. The experimental results indicate that ATNE outperforms the other state of the art frameworks by $3.28\times$ in sampling complexity for the same prediction accuracy. ATNE is also capable of identifying the Pareto designs for the difficult design spaces which the other existing frameworks are incapable of exploring effectively.

Acknowledgements

This chapter contains material printed in “Adaptive Threshold Non-Pareto Elimination: Re-thinking Machine Learning for System Level Design Space Exploration on FPGAs”, Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner, the Conference on Design Automation and Test in Europe (DATE) 2016. The chapter also contains material that was an improvement of the results in the conference paper. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Can One Use Pre-PnR Estimation Data to Aid FPGA Design Space Exploration?

5.1 Background and Motivation

In contrast to GPU and CPU OpenCL code compilation, which only consumes several milliseconds, implementing OpenCL designs on an FPGA requires hours of compute time. Moreover, merely tuning a few programming parameters in the OpenCL code will result in an abundance of possible designs (as demonstrated in Fig. 5.8). Implementing all these designs requires months of compilation time. Due to this fact, exploring the FPGA design space with brute force is almost impossible. Even performing hundreds of FPGA synthesis runs is prohibitive. In order to address this problem, machine learning framework in Chapter 4 has been developed to automatically predict the "good" (Pareto) designs by implementing and sampling only a small number of designs.

Although the GPU and FPGA are two different compute platforms, they are both dictated by the OpenCL programming model. The OpenCL design on the FPGA contains many GPU-like architectural features such as SIMD, compute unit parallelism, and local/global memory hierarchy. It is therefore reasonable to believe that if one uses OpenCL programming language on both devices, the GPU and FPGA design spaces will share a certain level of similarity. In contrast to implementing a design on the FPGA, it is significantly cheaper to compile a GPU program (milliseconds compilation on the GPU vs. hours of synthesis time on the FPGA). Therefore, one can easily obtain the GPU performance for a substantial number of the FPGA designs. Based on these observations, we aim to answer the question: *Can one use the GPU performance as an estimation to reduce the amount of implementations required for the FPGA design space exploration (DSE)?*

In addition to GPU performance, one can quickly obtain the pre-place and route (pre-PnR) hardware utilization report and pre-PnR estimated throughput, which are generated quickly during the beginning stages of the OpenCL-to-FPGA tool chain. Therefore, we developed another question: *Can one also use these estimation data to reduce the implementation time in the FPGA DSE?*

In this chapter, we answer these questions. To the best of our knowledge, this work is the first attempt to mine estimation such as GPU performance and pre-PnR results to aid FPGA DSE. Our approach reduces the sampling complexity for FPGA DSE by a factor of $1.26\times$ against the other state-of-the-art approaches.

The remainder of the chapter is organized as follows. In Section 5.2, we review related work. In Section 5.3, we provide a high-level description of the problems that we investigated. We provide more detailed empirical studies for these problems in Section 5.4. We describe the algorithm for our approach in Section 5.5. In Section 5.6, we provide a theoretical analysis to determine an important parameter in our algorithm. We evaluate the effectiveness of our approach in Section 5.7. Section 5.8 summarize the chapter.

5.2 Related Work

The topic of this chapter covers two areas of study. One area is using machine learning for hardware DSE. In recent years, researchers have been investigating how to use machine learning techniques to aid high-level synthesis DSE [65, 64]. The main idea of these studies is to predict the entire design space by sampling a small number of implementations. These previous works effectively reduced the compute time of the synthesis tools. However, these works only tried to use data from the target architecture (e.g., FPGA). These existing methods overlook the opportunity to use the training data from another architecture that is similar to the target architecture, which may provide a further reduction of the DSE time. Our work is not merely an extension of these machine learning frameworks. In this chapter, we use machine learning to research a more fundamental architecture/EDA question, while the contributions in [65, 64] focus on building a machine learning algorithm for FPGA DSE.

The second area of related work is investigating the performance probabilities between different architectures that can be programmed with the OpenCL language. In recent years, researchers have proposed multiple automated porting tools such as [105, 106, 107]. These tools study the differences among various processors (e.g., CPUs and GPUs). These studies have not addressed the OpenCL DSE problem on the FPGAs. Moreover, these studies are solving the problem of how to mitigate the differences between the architectures instead of how to mine the valuable similarities between different architectures.

In this chapter, we address these questions that are overlooked by the previous studies. Our approach focuses on extracting the useful information from the similarities between different architectures. We also studied how to use the extracted information to aid the existing machine learning DSE frameworks.

5.3 Overview

The OpenCL programming model provides the designer multiple tunable parameters such as size of local memory arrays, vector widths, and unroll factors. Modifying these parameters results in different performance. In the remainder of this chapter, we will refer to these tunable parameters as “knobs”. For an OpenCL-to-FPGA design, these knobs also determine the hardware utilization. Therefore, for a given FPGA application, tuning these knobs can generate a space which consists of many designs with different performances and hardware resource utilizations as shown in Fig. 5.8. The goal of DSE is to find the Pareto designs for an application.

In the remainder of this chapter, we will refer to GPU performance, pre-PnR throughput estimation, and pre-PnR resource utilization estimation as *estimation data* since they can be used to roughly estimate the real designs on the FPGA. Generating these estimations requires significantly less compute time than obtaining the real FPGA measurements does. However, the estimations are not identical to actual FPGA measurements, i.e., there exists some level of inaccuracy (potentially substantial differences) in the estimation data. For these reasons, the possibility of whether one can use the estimation data for FPGA DSE is unclear. The focus of this chapter is to investigate this problem.

We divided this broad problem into several more specific questions. The first question is: “Can one directly replace the real FPGA data with the estimation data for DSE?” The answer is “no”. The benefit of this approach is that it requires no PnR process except for the final implementations of its predicted Pareto designs. However, a serious issue exists in this approach – the predicted Pareto points may not be the true Pareto designs of the actual FPGA implementations. Therefore, using this approach may result in implementing the sub-optimal designs and ignoring the optimal ones. We provide a more detailed study on this issue in Section 5.4.2.

The second question is an extension of the first one: “Starting from the output of the first approach, can one permutate the knobs to conduct a local search around these predicted Paretos?” The answer is still “no”. It is a complex procedure to permutate multiple knobs for the local search. In fact, in order to achieve the real Pareto designs, one needs to specifically tune multiple knobs in the right direction to particular values. Due to this fact, the local search may require several hundreds of real FPGA implementations. This defeats the original purpose of using estimation data to save compute time.

The estimations provide us a mixture of correct and incorrect information for the real FPGA design space. This is the main cause of the issues in the previous two approaches. Based on this observation, the third question is: “Can one mine the estimation data to find the useful information for FPGA DSE?” Our final evaluation results in Section 5.7 provide a positive answer to this question. More details of how to build the data mining method are provided in Section 5.4.3.

Fig. 5.1 provides a brief overview of our approach. We built a method to mine the estimations and inject the extracted useful information into an existing machine learning framework. The data mining method extracts the order information, i.e., whether design A outperforms design B. We use the order information to identify the non-Pareto designs and remove them. In this way, the framework can avoid wasting compute time on implementing these non-Pareto designs.

As shown in Fig. 5.1, our estimation mining approach works independently from the machine learning framework. It shares the same sampled FPGA data with the machine learning framework. Therefore, our approach can be connected to various types of machine learning frameworks with almost no modification. In this chapter, we chose to connect our estimation mining approach with the machine learning framework in chapter 4 to build a full demonstrative working flow.

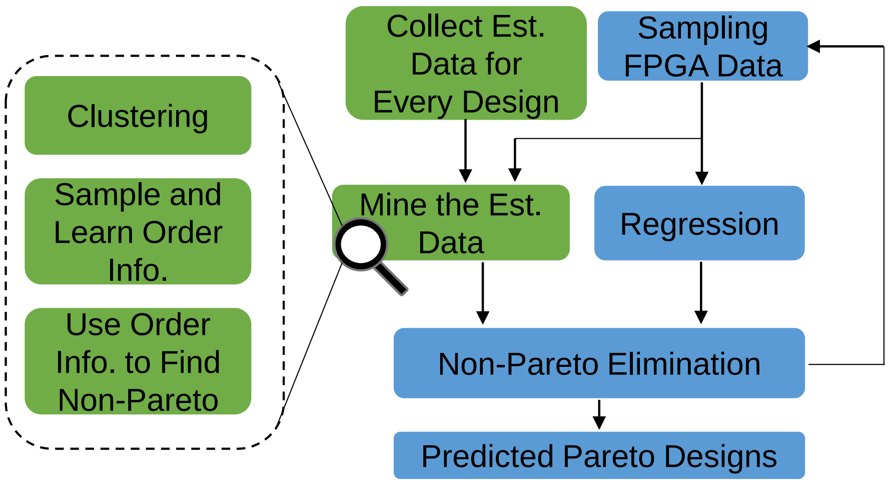


Figure 5.1. Flow diagram of the FPGA DSE process using estimation data. The blue blocks show the existing machine learning framework without using estimation data. The green blocks represent the estimation data mining method.

5.4 Can One Use Estimation Data for FPGA Design Space Exploration?

The discussion in this section consists of several empirical studies for the answers to the questions we introduced in Section 5.3.

5.4.1 Experimental Setup

For the experiments in the rest of this chapter, we use the 10 benchmarks as described in Section 4.6. We implemented the entire design space for each benchmark on the GPU and FPGA to collect the ground truth data. For the FPGA data, we used Altera OpenCL SDK 14.1 to generate end-to-end implementations on the Terasic DE5-net FPGA; this process required more than 6 months on 3 high-end workstations. We also used the “compile only” command flag of the Altera OpenCL SDK to generate the pre-PnR hardware utilization and throughput estimation data; this process required less than one week. We used the OpenCL SDK attached in CUDA 7.5 toolkit to measure the performance of these benchmarks on an Nvidia Kepler K20 GPU; this process required less than 3 days. The compute time for collecting the estimation data is negligible in comparison to the compute time for end-to-end FPGA implementations.

5.4.2 Can one directly use the Pareto points from the estimated data?

The first approach we investigated is to directly use the Pareto designs from the estimated design space, as if these estimated Pareto designs are the optimal on the FPGA. In this approach, we only use the estimation data, without any FPGA ground truth samples, to construct an objective space. Fig. 5.2 (a) provides an example of such an estimation objective space generated by using GPU performance and pre-PnR hardware utilization data. Then, we compute the Pareto designs (blue cross in Fig. 5.2 (a)) of the estimation space. These estimated Pareto designs are considered as the final FPGA DSE output. However, the results in Fig. 5.2 (b) indicate that this approach may not be able to reach the optimal FPGA designs. Some these estimated Pareto designs (blue crosses) are located quite far away from the real FPGA Pareto points (red dots).

In order to quantify the ineffectiveness of these falsely estimated Pareto designs, we use *average distance from reference set (ADRS)* [101] metric. ADRS measures the distance between the falsely estimated Pareto points (blue crosses) and the real Pareto points (red dots) in Fig. 5.2 (b). Therefore, the smaller the ADRS that an approach achieves, the better prediction quality it provides. We conducted this study using the 10 benchmarks. For each benchmark, we tested the GPU performance and pre-PnR throughput estimation as two different types of performance estimations. For hardware utilization estimation, we used pre-PnR results from the tool. As a comparison reference without using any estimation data, we tested the machine learning approach described in Chapter 4 on these benchmarks. Fig. 5.3 reports the ADRS results for these tests. From our experience, an application usually has more than 10 FPGA Pareto designs.

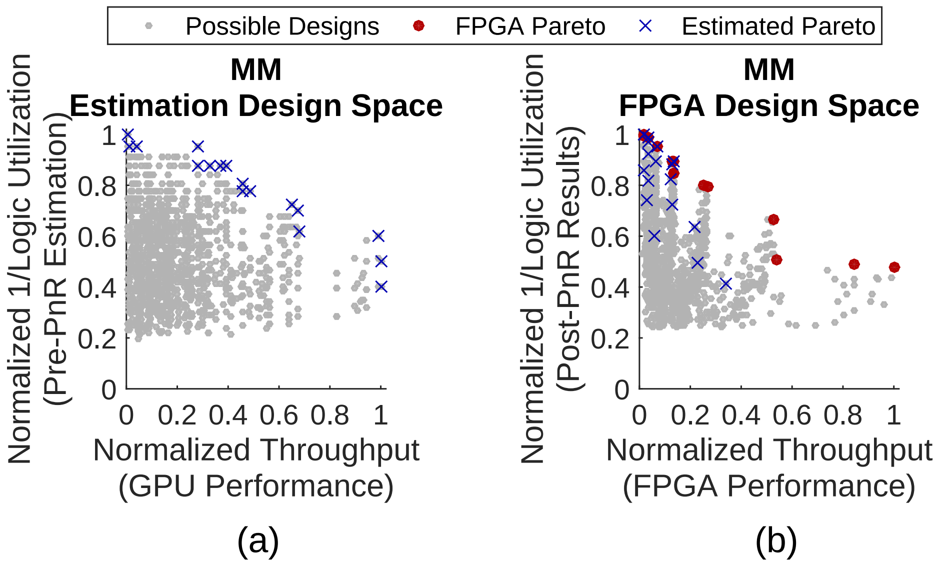


Figure 5.2. Example of directly using estimation data to find the Pareto frontier. (a) Estimated objective space by directly using GPU performance and Pre-PnR resource utilization report (blue crosses: estimated Pareto designs.). (b) Real FPGA objective space. Blue crosses in (b): the locations of the estimated Pareto designs in the real objective space. There exist significant distances between estimated Pareto designs (blue crosses) and the real FPGA Pareto frontier (red dots).

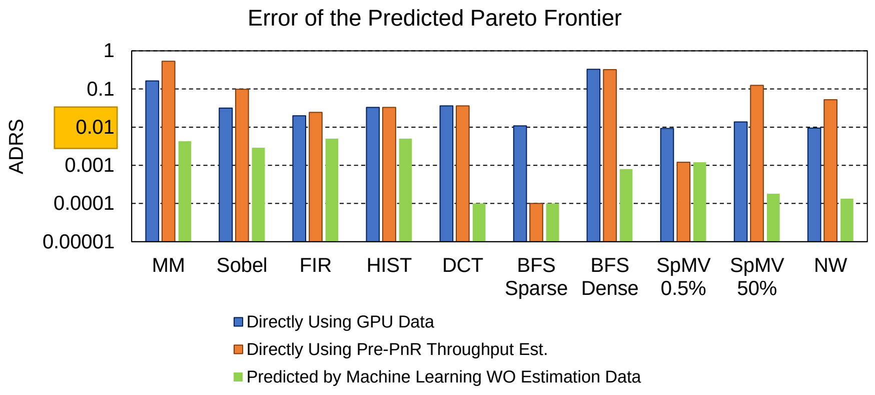


Figure 5.3. Prediction errors (ADRS) of 3 different approaches: using the GPU performance, the pre-PnR throughput estimation and a state-of-art learning algorithm (the learning algorithm only samples the FPGA data and requires no estimation data).

This means if the ADRS is more than 0.01, the entire predicted Pareto set will have a total error of more than 10%. Therefore, we set ADRS$\leq$ 0.01 as the error threshold for acceptable predictions (highlighted on the y-axis in Fig. 5.3).

From results in Fig. 5.3, we observed the approach that directly uses estimated Pareto points fails to achieve acceptable predictions for most applications. In most cases, using machine learning without estimation data outperforms the approaches using estimations in terms of prediction error. For certain cases such as BFS Sparse, SpMV 0.5% and NW, the estimated Pareto points achieved acceptable errors (≤ 0.01). However, these results are false appearances of effectiveness. This is caused by the extremely inaccurate estimation spaces. For example, in Fig. 5.4 (a), the designs in the estimation space are located in two vertical lines. All the designs on one of the lines are predicted as Pareto points. The designer receives a significantly large amount of predicted Pareto points (the massive blue cross set in Fig. 5.4 (b)). This massive Pareto prediction set has a great chance to contain all of the real Pareto designs. However, it is not an intelligent approach. Consider an extreme version of this situation: if the estimation predicts all designs as Pareto, the ADRS error becomes 0, but it is equivalent to using brute force since it requires the designer to implement the entire design space on the FPGA. Implementing all these estimated Pareto designs defeats the original purpose of using estimation to reduce the FPGA PnR compute time.

In the BFS Sparse example, the estimation data outputs 90 Pareto points. The number of the actual FPGA Pareto designs is only 6. This means the designer needs to implement all 90 of those designs on the FPGA and manually identify the 6 real Pareto points among them. Using the machine learning approach in Chapter 4 can achieve a comparable error, while it only requires 36 FPGA implementations. Therefore, in this case, although the estimation data provides an accurate Pareto prediction, it requires more compute time for FPGA PnR instead of reducing it.

In summary, it is unreliable to directly use the estimated data for FPGA DSE. Moreover, although the approach achieves an acceptable error result in a few cases, the cost of implementing all of its output defeats the purpose of the prediction. Due to these

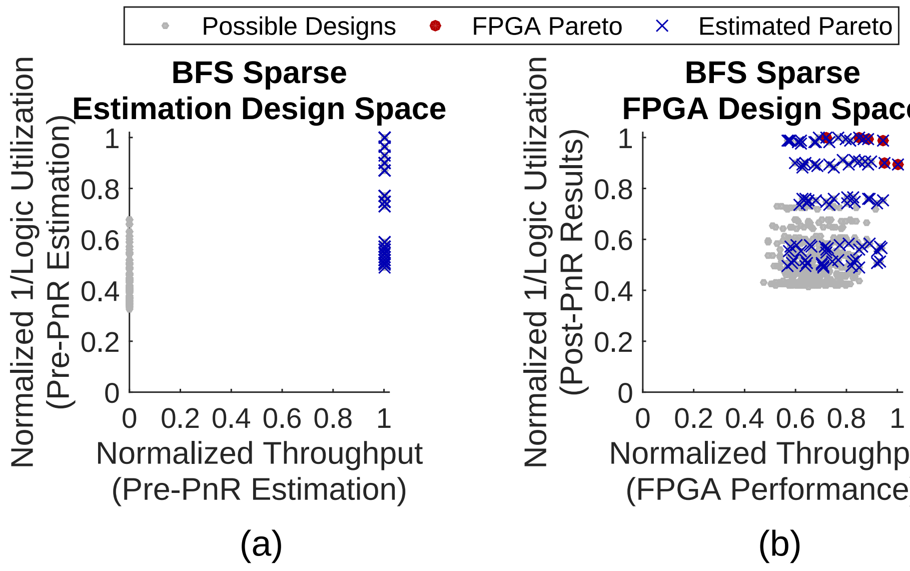


Figure 5.4. An example of extremely inaccurate estimation space. (a) Estimation objective space. The designs are located in two lines in the estimation space. All of the designs (90 designs depicted by the massive blue cross set) on one of the lines are predicted as Pareto points. (b) Real FPGA objective space. Only 6 designs (red dots) are real FPGA Pareto.

reasons, our answer is “no, one cannot directly use the estimated data to find FPGA Pareto points”.

5.4.3 Can one mine the estimation data for FPGA DSE?

As mentioned in Section 5.3, the quick answer is “yes”. Here, we provide a more detailed answer to this question by showing the process of how we built the approach for estimation data mining. We use the GPU and FPGA performances as an example to describe this approach. In this example, the GPU performance is the estimation while the FPGA performance is the ground truth. We assume the designer has the GPU performance data of the entire design space since one can quickly compile and run the program on the GPU. We also assume the designer only has a few sampled FPGA performance data due to the time consuming implementation process on the FPGA.

We chose to predict the pair-wise order (e.g., design *i* outperforms design *j*) for the FPGA design space. Each pair has a correspondent element in the order correlation matrix shown in Fig. 5.5 (a). The order correlation represents whether the pair has the same order or opposite orders on the GPU and FPGA. If we know the order correlation of (i,j), we can accurately use the GPU performance data to calculate the order between designs i and j on the FPGA. Furthermore, if we know every value in the order correlation matrix, we can calculate the FPGA Pareto points. Therefore, we built a data mining approach to predict this order correlation matrix.

However, since the order matrix is a mixture of -1's and 1's, it is impossible to predict each element accurately from a few samples. As a solution to this challenge, we used clustering to divide the matrix into smaller groups. We compute the GPU performance difference between every two designs as shown in Fig. 5.5 (b). In this way, each design pair has a correspondent element in this GPU performance (estimation) difference matrix. We cluster the estimation difference matrix. After clustering, each pair is assigned into a group (shown as the blocks in Fig. 5.5 (b)). Now, we move back to the order correlation prediction problem. As shown in 5.5 (c), some of these groups contain the pairs whose order correlation values are all -1's or all 1's. In each of these groups, one can sample a few values and predict the rest (the x's in Fig. 5.5 (c)). The predicted values reveal the orders between designs on the FPGA without actually implementing the correspondent designs. Similarly, one can use this procedure on the pre-PnR hardware utilization estimation. Eventually, with both performance and pre-PnR hardware utilization orders, one can determine which designs are non-Pareto and eliminate them to speedup the FPGA DSE process.

In the following experiment, we investigate whether the clustering method is effective with all 10 of our benchmarks. We quantify the effectiveness of the clustering by measuring how many non-Pareto points the method can determine. In the experiment, we also used two types of data: the GPU data against the Pre-PnR throughput in this study to compare their effectiveness. This experiment is not the evaluation of the entire working flow. In this experiment, we did not connect the data mining method with the machine learning framework as shown in Fig. 5.1. We only investigate whether the clustering is capable of dividing the order matrix into pure groups and whether one can use the information in the pure groups to identify non-Pareto designs. The evaluation of the full flow in Fig. 5.1 can be found in Section 5.7.

The experimental results in Fig. 5.6 indicate that our clustering method is effective. We also observed the more groups we divide the data into, the more effective this method is. According to this observation, we developed Algorithm 5.2. More details of the algorithm can be found in Section 5.5. From further experiments, we observed that Algorithm 5.2 is usually capable of creating more than 1000 groups which provide a high possibility to discover non-Pareto points. To answer the question we introduced earlier, yes, one can mine the estimation data for FPGA DSE by using the clustering approach.

The results in Fig. 5.6 also indicate that the GPU performance is a better type of estimation data than the pre-PnR throughput estimation. The pre-PnR estimation does not take into account the run-time behavior such as memory accesses, inter module communication, and branches on the architecture. Therefore, for certain applications, the GPU performance estimates the FPGA behavior more accurately than pre-PnR estimation does. However, for certain applications, the FPGA performance highly depends on how the designer customizes the hardware (e.g., the number of control unit replications). In contrast, the fixed architecture on the GPU is incapable of estimating such a hardware customization. In these situations, the pre-PnR throughput report estimates the FPGA behavior more effectively since it has better knowledge about the hardware customization. The results for BFS Sparse and SpMV 50% in Fig. 5.6 indicate this phenomenon.

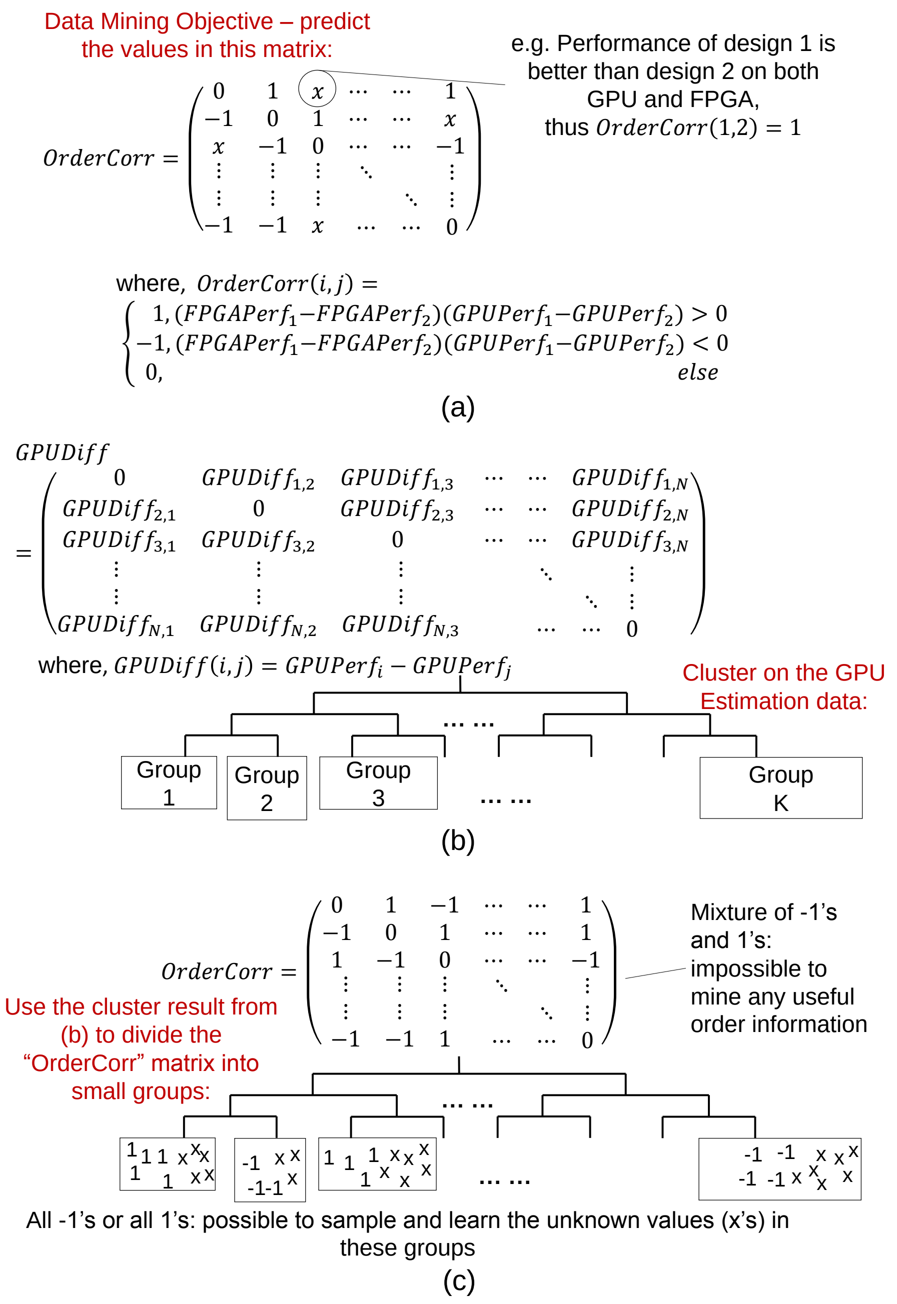

Data Mining Objective – predict the values in this matrix:
e.g. Performance of design 1 is better than design 2 on both GPU and FPGA, thus $OrderCorr(1,2) = 1$
$OrderCorr = \begin{pmatrix} 0 & 1 & x & \cdots & \cdots & 1 \\ -1 & 0 & 1 & \cdots & \cdots & x \\ x & -1 & 0 & \cdots & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ -1 & -1 & x & \cdots & \cdots & 0 \end{pmatrix}$
where, $OrderCorr(i,j) = \begin{cases} 1, (FPGAPerf_1 - FPGAPerf_2)(GPUPerf_1 - GPUPerf_2) > 0 \\ -1, (FPGAPerf_1 - FPGAPerf_2)(GPUPerf_1 - GPUPerf_2) < 0 \\ 0, \quad else \end{cases}$
(a)
$GPUDiff = \begin{pmatrix} 0 & GPUDiff_{1,2} & GPUDiff_{1,3} & \cdots & \cdots & GPUDiff_{1,N} \\ GPUDiff_{2,1} & 0 & GPUDiff_{2,3} & \cdots & \cdots & GPUDiff_{2,N} \\ GPUDiff_{3,1} & GPUDiff_{3,2} & 0 & \cdots & \cdots & GPUDiff_{3,N} \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ GPUDiff_{N,1} & GPUDiff_{N,2} & GPUDiff_{N,3} & \cdots & \cdots & 0 \end{pmatrix}$
where, $GPUDiff(i,j) = GPUPerf_i - GPUPerf_j$
Cluster on the GPU Estimation data:
Group 1
Group 2
Group 3
… …
Group K
(b)
$OrderCorr = \begin{pmatrix} 0 & 1 & -1 & \cdots & \cdots & 1 \\ -1 & 0 & 1 & \cdots & \cdots & 1 \\ 1 & -1 & 0 & \cdots & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ -1 & -1 & 1 & \cdots & \cdots & 0 \end{pmatrix}$
Mixture of -1's and 1's: impossible to mine any useful order information
Use the cluster result from (b) to divide the "OrderCorr" matrix into small groups:
1 1 1 x x x 1 1 x x
-1 x x -1 -1 x
1 1 1 x x x 1 x x x
… …
-1 -1 x x x -1 -1 x x x x
All -1's or all 1's: possible to sample and learn the unknown values (x's) in these groups
(c)

Figure 5.5. Our data mining approach: cluster, sample and learn the order information. (a) The order correlation matrix between the GPU data and FPGA data. The value represents whether the pair has the same order or opposite orders on the GPU and FPGA. Note this matrix is the objective of the data mining. (b) Clustering on the GPU data difference matrix. The element (i, j) in the matrix is the difference of the GPU data between designs i and j. (c) Use the cluster result from (b) to divide the order matrix into smaller groups. Some of these smaller groups have pure value of all -1's or 1's. This provides a possibility to predict the values of a group based on a few samples.

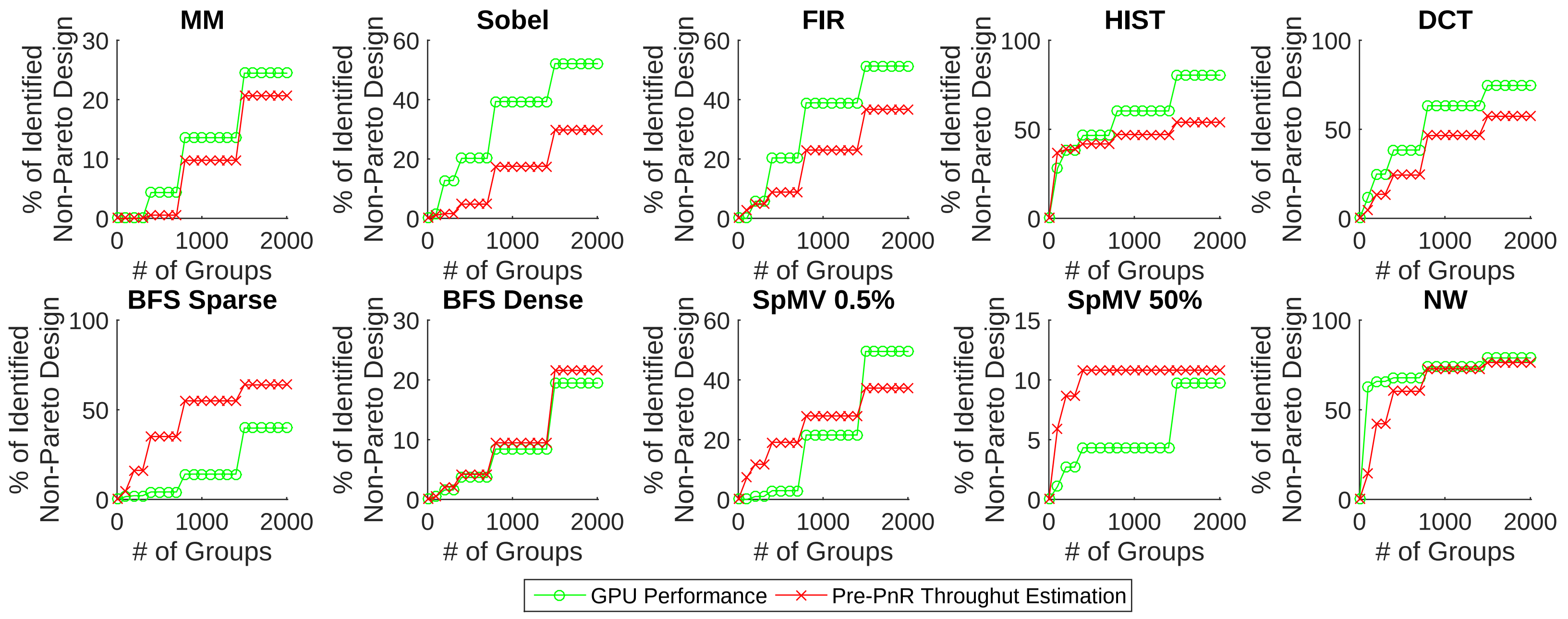


Figure 5.6. The effectiveness of the data mining method. The results indicate that our data mining method with clustering is effective for all benchmarks. The GPU data is more effective in most of the benchmarks.

5.5 Algorithms

In this section we describe the estimation data mining algorithm. The overall data mining algorithm is described by Algorithm 5.1. The algorithm consists of three stages (correspondent to the three stages within "Mine the Est. Data" block in Fig. 5.1): (1) clustering; (2) predicting the order correlation between the estimation and FPGA data; (3) using the order information to find non-Pareto points. The clustering step is a recursive function described in Algorithm 5.2. The predicting step is described by lines 7 - 18 in Algorithm 5.1. The step of using the order information to identify the non-Pareto points is described by lines 19 - 29 in Algorithm 5.1.

The data mining algorithm takes the estimation data of all designs and the FPGA data of a few sampled designs as input. The algorithm firstly computes the estimation difference between every two designs to obtain matrices MP_{diff} and MH_{diff} for performance and hardware utilization respectively. The next step is to fill the order correlation matrices MP_{order} and MH_{order} with 0's since the values of these elements are all unknown without the FPGA samples. In the next step, scan all the sampled FPGA data, if design i and j are both sampled, then compute the values for (i,j) in MP_{order} and MH_{order}. After calling the clustering procedure (described in Algorithm 5.2), each of the element in the order correlation matrices belongs to a specific group. Lines 7 - 18 show the procedure of predicting the values (predict those x's demonstrated in Fig. 5.5 (c)) in each group. The algorithm scans all the groups (depicted by the blocks in Fig. 5.5). In each group, two conditions are checked: if the number of the sampled data is above a threshold N_{th} in this group; if the sampled values are all -1's or all 1's. If both conditions are met, then we predict all the unknown values in that group are the same as the sample value. The threshold N_{th} is to guarantee that we have enough samples to accurately predict those unknown values. If a group does not have at least N_{th} sample, we choose to leave those values unknown. We provide a theoretical analysis of why N_{th} works and how to compute a proper N_{th} in Section 5.6.

We use these predicted orders to identify non-Pareto designs. As described by lines 19 - 29 in Algorithm 5.1, we loop through all pair-wise orders. In each iteration, we check one design is superior to another design in both performance and hardware utilization. If the check is positive, then put the inferior design in the non-Pareto set. The estimation mining approach will provide this non-Pareto set to the overall process shown in Fig. 5.1.

Recursive Clustering: The input of Algorithm 5.2 are order correlation matrix and estimation difference matrix for all the pairs in the design space. The functionality of this algorithm is to cluster these pairs. The algorithm recursively calls the clustering function to split each parent group into 2 groups only if the number of samples in that parent group is greater than N_{th}, as shown in lines 5 - 6 of Algorithm 5.2. When the number of samples $< N_{th}$, the recursion terminates. As output, this algorithm generates a group ID for each pair to indicate which group the pair and its correspondent element in the order correlation matrix belong to. This clustering algorithm is called twice in lines 5 and 6 in Algorithm 5.1 for clustering the performance and hardware utilization data respectively.

Algorithm 5.1: Estimation Data Mining Algorithm

```
Input  : Performance Est. $Est_p$; Hardware Utilization Est. $Est_h$; Sampled FPGA
         Performance $S_p$; Sampled FPGA Hardware Utilization $S_h$
1  Compute differences of estimations $MP_{diff}$ (Perf.) and $MH_{diff}$ (HW Util.)
2  Initialize the order correlation matrices $MP_{order}$ (Perf.) and $MH_{OrderCorr}$ (HW Util.)
     with all 0's
3  Compute $MP_{OrderCorr}(i,j)$ if $i$ and $j$ found in $S_p$
4  Compute $MH_{OrderCorr}(i,j)$ if $i$ and $j$ found in $S_h$
5  $GIDP = ClusterRecur(MP_{OrderCorr}, MP_{Diff})$
6  $GIDA = ClusterRecur(MA_{OrderCorr}, MA_{Diff})$
7  for $g = 1$ to $max(GIDP)$ do
8      Compute sampling threshold $N_{th}$
9      if The number of $(MP_{OrderCorr}(GIDP == g) != 0) \geq N_{th}$ then
10         if all of $(MP_{OrderCorr}(GIDP == g) != 0) == -1$ then
11             $MP_{OrderCorr}(GIDP == g) = -1$
12         end
13         if all of $(MP_{OrderCorr}(GIDP == g) != 0) == 1$ then
14             $MP_{OrderCorr}(GIDP == g) = 1$
15         end
16     end
17 end
18 Compute $MA_{OrderCorr}(GIDA == g)$ in the same way
19 $NonP = \emptyset$
20 for $i = 1$ to number of designs do
21     for $j = 1$ to number of designs do
22         if $MP_{OrderCorr}(i,j) * MP_{Diff}(i,j) < 0$ and
             $MH_{OrderCorr}(i,j) * MH_{Diff}(i,j) < 0$ then
23             Add $i$ in $NonP$
24         end
25         if $MP_{OrderCorr}(i,j) * MP_{Diff}(i,j) > 0$ and
             $MH_{OrderCorr}(i,j) * MH_{Diff}(i,j) > 0$ then
26             Add $j$ in $NonP$
27         end
28     end
29 end
Output : The predicted non-Pareto set NonP
```

Algorithm 5.2: Recursive Clustering

```
1  Procedure ClusterRecur($M_{OrderCorr}$, $M_{Diff}$)
2      Compute sampling threshold $N_{th}$ using equation 5.2
3      if Number of samples in M is greater than the threshold n then
4          Cluster on $M_{Diff}$ into 2 groups, obtain matrix of group ID – $GID_{tmp}$
5          $GID_{left} = ClusterRecur(M_{OrderCorr}(GID_{tmp} == 1), M_{Diff}(GID_{tmp} == 1))$
6          $GID_{right} = ClusterRecur(M_{OrderCorr}(GID_{tmp} == 2), M_{Diff}(GID_{tmp} == 2))$
7          $GID_{output}$ = combining ($GID_{left}$) and $GID_{right}$
8      end
9      else
10         $GID_{output}$ = matrix of all 1’s
11     end
12     Return $GID_{output}$
```

5.6 Analysis: Determine N_{th}

In Algorithm 5.2 we recurse until the number of samples is less than the threshold N_{th}. If we set N_{th} too low, it may result in using very few sample data to guess a huge group of unknown data (e.g. using 1 sample to guess a group of 1000 elements). However, if we set N_{th} too high, the recursive process may terminate too soon to create enough groups. This reduces the chance for the algorithm to find and eliminate the non-Pareton points efficiently (see Fig. 5.6). Therefore, it is important to determine an appropriate N_{th}.

The sampling process within each group can be mathematically modeled as drawing boolean samples without replacement. Therefore, its underlying probability distribution is hypergeometric:

$$P(X=k) = \frac{C_k^K C_{n-k}^{N-K}}{C_n^N} \tag{5.1}$$

where C_j^i is "i choose j", $N =$ the population size, $K =$ the number of 1's versus -1's in the population, $k =$ the number of 1s drawn, and $n =$ the total number of samples.

Under this model the ideal threshold n is the answer to the question, "With a fixed likelihood $1-\beta$, what is the minimum number of samples n for which I am likely with probability $1-\beta$ to draw all 1's even though the population contains (εN) -1's and $(1-\varepsilon N)$ 1's?" To see this more clearly, note that we are attempting to determine how likely it is that we have been tricked by our sampling limitations into thinking that a cluster is either entirely composed of 1's, or vice versa. We use a recursive numerical method described in equation 5.2 to solve this problem.

$$f(x) := \begin{cases} x & \text{if } \frac{N!(\hat{K}-x)!}{\hat{K}!(N-x)!} \leq \beta \text{ or } x = N \\ f(x+1) & \text{otherwise} \end{cases} \tag{5.2}$$

where $\hat{K} = \lfloor N(1-\varepsilon) \rceil$. We let the user of our tool provide β and ε as two parameters describing how much error is acceptable for the DSE. Then, we can take the β and ε into the numerical method to produce the appropriate N_{th}.

5.7 Evaluation Results

We evaluated the effectiveness of the overall process (Fig. 5.1) on the 10 benchmarks. In the experiment, we set $\beta = 0.24$, $\varepsilon = 0.1$. We used the Matlab hierarchical clustering built-in function. In the function, we configured the clustering metric to inner squared Euclidean distance.

Fig. 5.8 provides a visualization of the predictions by using the GPU performance as the estimation. Here, we choose to demonstrate the runs which achieved ADRS≤ 0.01. As shown in the figure, although these designs are significantly different from each other (even different input data create different design spaces for the same application: e.g. BFS and SpMV), our approach successfully used the GPU performance to find most of the real FPGA Pareto designs.

Next, we report the amount of sampling complexity that our approach reduces. Our comparison baseline is the machine learning approach from Chapter 4, which does not use any estimation data (only the blue part in Fig. 5.1). There exists a parameter α (introduced in Chapter 4) that controls the trade-off between the prediction error and sampling complexity. For each benchmark, we tuned this α from $1-0.4$ to $1-0.4\times 0.5^{11}$ to generate a variety of results with different sampling complexities and prediction errors as shown in Fig. 5.9. We also tested two types of performance estimations: GPU performance and pre-PnR estimation. The results in Fig. 5.9 show, for most applications, the mining approach consumes less sampling complexities and achieves comparable prediction qualities in comparison with the baseline approach.

Fig. 5.7 shows the minimal complexities to achieve an ADRS≤ 0.01 of these

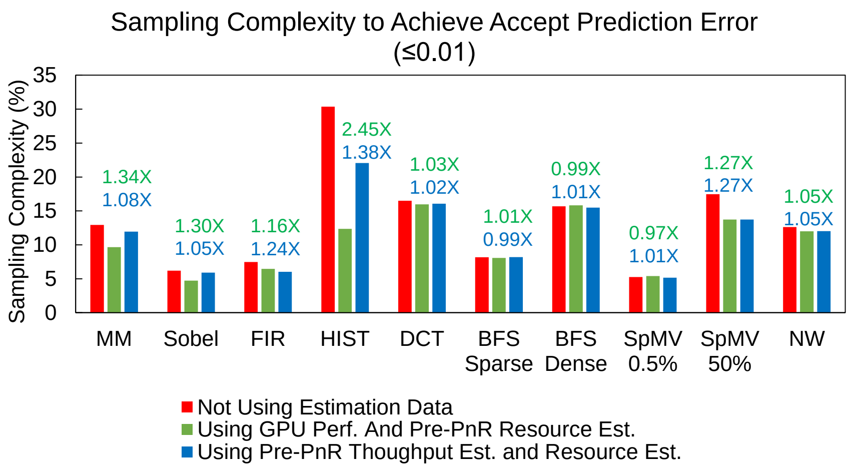


Figure 5.7. Sampling Complexities to Achieve Accept Prediction Error (ARDS $leq0.01$). The colored numbers (green for GPU performance, blue for pre-PnR throughput estimation) represent the complexity reductions in comparison with the baseline.

approaches. For all 10 benchmarks, in comparison with the baseline, the GPU performance approach reduces the sampling complexity by $1.26\times$ on average, while the pre-PnR estimated throughput achieves a complexity reduction of $1.11\times$ on average. This result indicates that our approach saves more PnR time than an extremely efficient state-of-the-art machine learning tool does for FPGA DSE. The reduction factor $1.26\times$ means one could save more than hundreds of compute hours by using this approach.

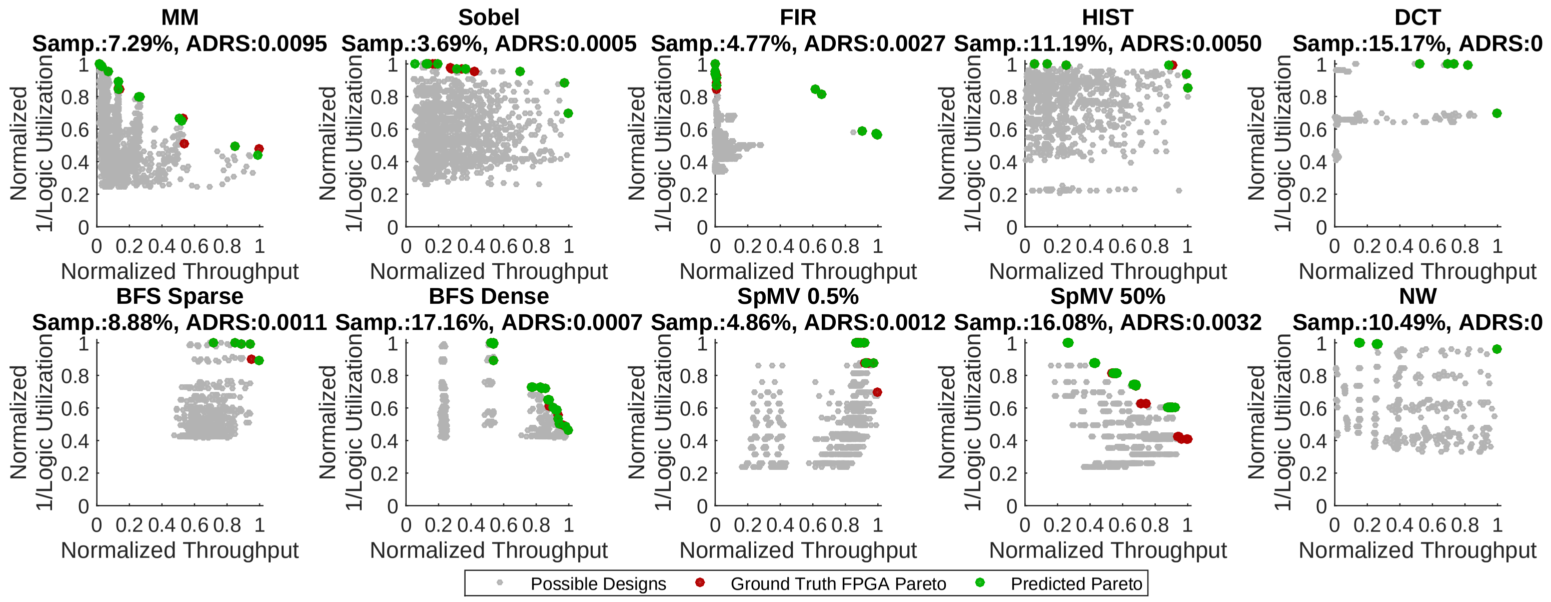


Figure 5.8. Visualization of the run results of our approach using GPU performance data as estimation. Note this is the best case of multiple runs. Fig. 5.9 depicts the average case.

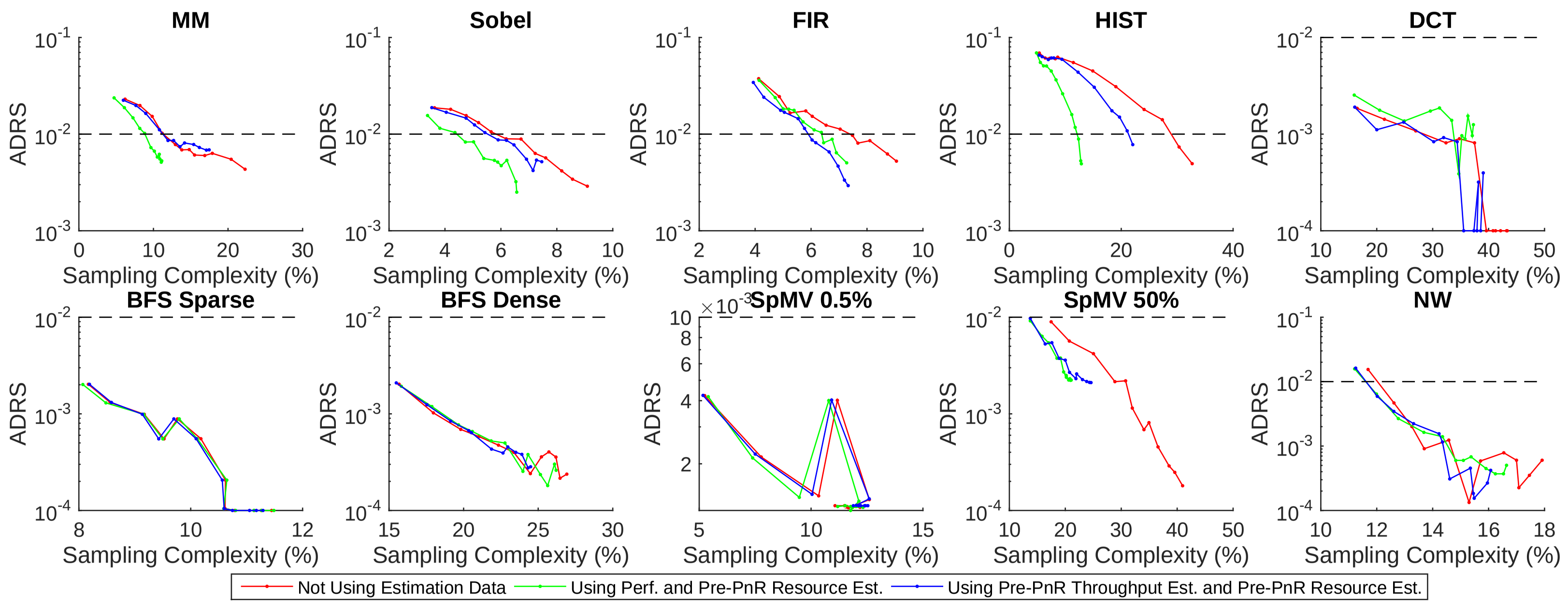


Figure 5.9. The evaluation results of using the GPU performance, pre-PnR throughput estimation and the baseline machine learning method. We tuned the parameter given in Chapter 4 to control the trade-off between the error and complexity and created a variety of different results for each benchmark.

5.8 Summary

We investigated the possibility of using estimation data such as GPU performance and pre-PnR results to aid FPGA DSE. We proposed a data mining method that extracts the useful information from the estimations with clustering and sampling techniques. We evaluated the effectiveness of our method with 10 benchmarks. Our GPU performance mining approach successfully achieves comparable or even better prediction errors in comparison with the approach not using estimations, while reducing the sampling complexity by $1.26\times$, i.e. hundreds of compute hours. Therefore, our answer to the question in the title is: Yes, with an intelligent data mining approach, one can use the GPU performance data for FPGA DSE.

Acknowledgements

This chapter contains material printed in the submission of "Can One Use GPU Performance Data for FPGA Design Space Exploration?", Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner, to International Conference On Computer Aided Design (ICCAD), 2016. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Conclusion

Optimizing the hardware accelerated designs requires the programmer to explore different implementations on different heterogeneous devices. This dissertation has presented several research results for this DSE problem. This dissertation has provided two heterogeneous hardware accelerated paradigms. The first paradigm of a hardware accelerated DNA sequence alignment technology showed a 115$\times$ speedup against the software. The second paradigm demonstrated a FPGA-GPU-CPU heterogeneous architecture which accelerates a biomedical imaging application by 237$\times$. According to our firsthand experience, these heterogeneous devices require different types of programming skills.

Recently, there is a trend to use one unified programming language for different heterogeneous devices. The OpenCL-to-FPGA tools allow the designer to program FPGA with OpenCL. Along with GPU and CPU OpenCL SDKs, the OpenCL-to-FPGA tool enables the possibility of using one unified language to program all three heterogeneous devices. However, the compilation time of the OpenCL-to-FPGA tool creates the second bottleneck for DSE. Using the brute force method to explore the entire design space consumes months of compilation time.

To address this issue, this dissertation has provided an automatic approach that uses machine learning for efficient OpenCL-to-FPGA DSE. The machine learning approach can identify the optimal designs by learning from very few training samples. This machine learning approach reduces the amount of required compilations by 3.28$\times$ (hundreds of compilation hours) against the other state of the art machine learning frameworks for FPGA DSE. In addition, a data mining approach is proposed to utilize the estimation data for FPGA DSE. This data mining approach further reduces the amount of required compilations by 1.26$\times$.

Chapter 7

Future Directions

There exist many possible research directions in the hardware acceleration design space exploration problem. I will provide several directions that could be explored using the machine learning approaches proposed in this dissertation.

Quantify the Difficulty of Design Space Exploration: Chapter 4 presents the sampling complexities of different design spaces. For effective design space exploration, some design spaces require more sampling complexity than others do. This is due to the fact that some design spaces are sparse (i.e. containing more outliers) than the others. Therefore, the first possible direction is to develop a methodology that can identify the exploration complexity of a given design space. This study contains two aspects. The first aspect is to investigate the proper metric that can quantify the exploration complexity of the design space. The second aspect is to investigate the effective method that uses very few samples to estimate such a metric.

Quantify the Effectiveness of Estimation Data: The second possible direction is to extend the study of using Pre-PnR estimation data. There exist many different estimation data such as estimated LUT, DSP and Register utilizations as well as the clock frequency estimation. Moreover, in other Pre-PnR stages such as RTL emulation, there exists performance estimation such as number of clock cycles. These estimations predict the Post-PnR data with different accuracies. Quantifying how accurate these estimation data are for different applications will provide the designers a reference for using these estimation data. A robust data mining methodology that can automatically learn the accuracies of these different estimation data and use them effectively is highly desired.

Learning the Underlying Architecture Features for Input-Data Sensitive Operations: The performance of some applications is data sensitive. For example, the SpMV and BFS applications in Chapter 4. With different input data, the application produces a different design space. This requires the machine learning algorithm to re-sample the new design space. This fact creates a significant overhead especially in those systems where the input data change dramatically. The third direction is therefore to develop a methodology that can automatically learn the underlying architecture properties for the data sensitive design spaces.

Explore and Quantify the Effectiveness of Other Estimations: The fourth possible direction is the most ambitious: develop a machine learning approach that could learn a domain of applications. This idea is to imitate the human designer with machine learning. The human designer usually learns the tuning skills from other similar designs. The goal of this direction is to train a machine with the ground truth data of a type of application e.g. sliding window. The trained machine can later effectively tune a brand new sliding window application based on the knowledge from this domain. A possible method for this problem is to use deep learning.

Bibliography

[1] M. Jacobsen, S. Sampangi, Y. Freund, and R. Kastner, "Improving FPGA accelerated tracking with multiple online trained classifiers," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–7.

[2] J. Cho, S. Mirzaei, J. Oberg, and R. Kastner, "Fpga-based face detection system using haar classifiers," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 103–112.

[3] J. W. Tang, N. Shaikh-Husin, U. U. Sheikh, and M. N. Marsono, "FPGA-based real-time moving target detection system for unmanned aerial vehicle application," *International Journal of Reconfigurable Computing*, vol. 2016, 2016.

[4] J. Maria, J. Amaro, G. Falcao, and L. A. Alexandre, "Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems," *Neural Processing Letters*, pp. 1–14, 2015.

[5] D. Tasson, A. Montagnini, R. Marzotto, M. Farenzena, and M. Cristani, "FPGA-based pedestrian detection under strong distortions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, June 2015, pp. 65–70.

[6] S. N. Sinha, J. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," *Machine Vision and Applications*, vol. 22, no. 1, pp. 207–217, 2011.

[7] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 161–168.

[8] B. S. C. Varma, K. Paul, M. Balakrishnan, and D. Lavenier, "FAssem: FPGA based acceleration of de novo genome assembly," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 173–176.

[9] S. Aluru and N. Jammula, “A review of hardware acceleration for computational genomics,” *IEEE Design Test*, vol. 31, no. 1, pp. 19–30, Feb 2014.

[10] B. S. C. Varma, K. Paul, and M. Balakrishnan, “FPGA-based acceleration of protein docking,” in *Architecture Exploration of FPGA Based Accelerators for BioInformatics Applications*, 2016, pp. 39–54.

[11] J. Arram, K. H. Tsoi, W. Luk, and P. Jiang, “Reconfigurable acceleration of short read mapping,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 210–217.

[12] W. Tang, W. Wang, B. Duan, C. Zhang, G. Tan, P. Zhang, and N. Sun, “Accelerating millions of short reads mapping on a heterogeneous architecture with FPGA accelerator,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 184–187.

[13] C. Dennl, D. Ziener, and J. Teich, “Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 25–28.

[14] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer, “Accelerating join operation for relational databases with FPGAs,” in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 17–20.

[15] J. Casper and K. Olukotun, “Hardware acceleration of database operations,” in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 151–160.

[16] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, “Database analytics acceleration using FPGAs,” in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 411–420.

[17] P. Bakkum and K. Skadron, “Accelerating SQL database operations on a GPU with CUDA,” in *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 94–103.

[18] W. Wang and X. Huang, “FPGA implementation of a large-number multiplier for fully homomorphic encryption,” in *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, May 2013, pp. 2589–2592.

[19] A. Gielata, P. Russek, and K. Wiatr, “AES hardware implementation in FPGA for algorithm acceleration purpose,” in *Signals and Electronic Systems, 2008. ICSES ’08. International Conference on*, Sept 2008, pp. 137–140.

[20] J. Zutter, M. Thalmaier, M. Klein, and K. O. Laux, "Acceleration of RSA cryptographic operations using FPGA technology," in *2009 20th International Workshop on Database and Expert Systems Application*, Aug 2009, pp. 20–25.

[21] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography," in *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, Nov 2007, pp. 65–68.

[22] F. Grüll and U. Kebschull, "Biomedical image processing and reconstruction with dataflow computing on FPGAs," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–2.

[23] D. Lee, P. Meng, M. Jacobsen, H. Tse, D. D. Carlo, and R. Kastner, "A hardware accelerated approach for imaging flow cytometry," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[24] X. Gu, Y. Zhu, S. Zhou, C. Wang, M. Qiu, and G. Wang, "A real-time FPGA-based accelerator for ECG analysis and diagnosis using association-rule mining," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 15, no. 2, p. 25, 2016.

[25] X. Zhao, J. Hu, and P. Zhang, "GPU-based 3D cone-beam CT image reconstruction for large data volume," *Journal of Biomedical Imaging*, vol. 2009, p. 8, 2009.

[26] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the common unified device architecture (cuda)," in *2007 IEEE Nuclear Science Symposium Conference Record*, vol. 6, Oct 2007, pp. 4464–4466.

[27] N. Wilt, *The CUDA handbook: A comprehensive guide to GPU programming*. Pearson Education, 2013.

[28] D. Hefenbrock, J. Oberg, N. T. N. Thanh, R. Kastner, and S. B. Baden, "Accelerating Viola-Jones face detection to FPGA-level using GPUs," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 11–18.

[29] T. Nguyen, D. Hefenbrock, J. Oberg, R. Kastner, and S. Baden, "A software-based dynamic-warp scheduling approach for load-balancing the Viola–Jones face detection algorithm on GPUs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 677–685, 2013.

[30] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. Stone, D. B. Kirk, and W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, pp. 73–82.

[31] S. Kilts, *Advanced FPGA design: architecture, implementation, and optimization*. John Wiley & Sons, 2007.

[32] B. Zhou, Y. Peng, and D. Hwang, "Pipeline FFT architectures optimized for FPGAs," *International Journal of Reconfigurable Computing*, vol. 2009, p. 1, 2009.

[33] N. Kapre and D. Ye, "GPU-accelerated high-level synthesis for bitwidth optimization of FPGA datapaths," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 185–194.

[34] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A tradeoff analysis of FPGAs, GPUs, and multicores for sliding-window applications," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 1, p. 2, 2015.

[35] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2012, pp. 47–56.

[36] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on FPGA," in *2013 First International Symposium on Computing and Networking*, Dec 2013, pp. 112–121.

[37] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 9–16.

[38] A. R. Hastie, L. Dong, A. Smith, J. Finklestein, E. T. Lam, N. Huo, H. Cao, P. Kwok, K. R. Deal, J. Dvorak, M. Luo, Y. Gu, and M. Xiao, "Rapid genome mapping in nanochannel arrays for highly complete and accurate de novo sequence assembly of the complex Aegilops tauschii genome," *PloS one*, vol. 8, no. 2, p. e55864, 2013.

[39] S. Iravanian and D. J. Christini, "Optical mapping system with real-time control capability," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 293, no. 4, pp. H2605–H2611, 2007.

[40] H. Pak, Y. Liu, H. Hayashi, Y. Okuyama, P. Chen, and S. Lin, "Synchronization of ventricular fibrillation with real-time feedback pacing: implication to low-energy defibrillation," *American Journal of Physiology-Heart and Circulatory Physiology*, vol. 285, no. 6, pp. H2704–H2711, 2003.

[41] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: from prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[42] D. Chen and D. Singh, "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms," in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, Jan 2013, pp. 297–304.

[43] V. M. Morales, P. Horrein, A. Baghdadi, E. Hochapfel, and S. Vaton, "Energy-efficient FPGA implementation for binomial option pricing using OpenCL," in *2014 Conference on Design, Automation & Test in Europe*, 2014, pp. 208:1–208:6.

[44] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, July 2009, pp. 35–42.

[45] D. Navarro, O. Lucia, L. A. Barragan, I. Urriza, and O. Jimenez, "High-level synthesis for accelerating the FPGA implementation of computationally demanding control algorithms for power converters," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1371–1379, Aug 2013.

[46] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: productivity, performance, and software constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, p. 1, 2012.

[47] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2012, pp. 531–534.

[48] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, p. 24, 2013.

[49] J. Matai, A. Irturk, and R. Kastner, "Design and implementation of an FPGA-based real-time face recognition system," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 97–100.

[50] J. Matai, P. Meng, L. Wu, B. Weals, and R. Kastner, "Designing a hardware in the loop wireless digital channel emulator for software defined radio," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 206–214.

[51] J. Zhang, Z. Zhang, S. Zhou, M. Tan, X. Liu, X. Cheng, and J. Cong, "Bit-level optimization for high-level synthesis and FPGA-based acceleration," in *18th annual ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2010, pp. 59–68.

[52] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Luján, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–8.

[53] J. Monson, M. Wirthlin, and B. L. Hutchings, "Optimization techniques for a high level synthesis implementation of the Sobel filter," in *Reconfigurable Computing and FPGAs (ReConFig), 2013 International Conference on*, Dec 2013, pp. 1–6.

[54] J. Matai, D. Lee, A. Althoff, and R. Kastner, "Composable, parameterizable templates for high-level synthesis," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2016, pp. 744–749.

[55] M. R. Garey, D. S. Johnso, and L. Stockmeyer, "Some simplified NP-complete graph problems," *Theoretical computer science*, vol. 1, no. 3, pp. 237–267, 1976.

[56] T. G. Szymanski, "Dogleg channel routing is NP-complete," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, no. 1, pp. 31–41, January 1985.

[57] S. Hong and H. Kim, "An integrated GPU power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2010, pp. 280–289.

[58] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 382–393.

[59] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU workloads and systems," in *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 31–42.

[60] S. Hong and H. .Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, Jun 2009.

[61] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *ACM Sigplan Notices*, vol. 45, no. 5, 2010, pp. 105–114.

[62] K. Dohi, K. Fukumoto, Y. Shibata, and K. Oguri, “Performance modeling and optimization of 3-D stencil computation on a stream-based FPGA accelerator,” in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013, pp. 1–6.

[63] S. Skalicky, S. López, and M. Lukowiak, “Performance modeling of pipelined linear algebra architectures on FPGAs,” *Computers & Electrical Engineering*, vol. 40, no. 4, pp. 1015–1027, 2014.

[64] M. Zuluaga, G. Sergent, A. Krause, and M. Püschel, “Active learning for multi-objective optimization,” in *Proc. 30th Int. Conf. on Machine Learning (ICML)*, 2013, pp. 462–470.

[65] H. Liu and L. P. Carloni, “On learning-based methods for design-space exploration with high-level synthesis,” in *Proc. 50th Annu. Design Automation Conf. (DAC)*, 2013, pp. 50:1–50:7.

[66] I. Birol, S. D. Jackman, C. B. Nielsen, J. Q. Qian, R. Varhol, G. Stazyk, R. D. Morin, Y. Zhao, M. Hirst, J. E. Schein, D. E. Horsman, J. M. Connors, R. D. Gascoyne, M. A. Marra, and S. J. M. Jones, “*De novo* transcriptome assembly with abyss,” *Bioinformatics*, vol. 25, no. 21, pp. 2872–2877, 2009.

[67] A. Zuccolo, A. Sebastian, J. Talag, Y. Yu, H. Kim, K. Collura, D. Kudrna, and R. A. Wing, “Transposable element distribution, abundance and role in genome size variation in the genus Oryza,” *BMC Evolutionary Biology*, vol. 7, no. 1, p. 152, 2007.

[68] E. T. Lam, A. Hastie, C. Lin, D. Ehrlich, S. K. Das, M. D. Austin, P. Deshpande, H. Cao, N. Nagarajan, M. Xiao, and P. Kwok, “Genome mapping on nanochannel arrays for structural variation analysis and sequence assembly,” *Nature Biotechnology*, vol. 30, no. 8, pp. 771–776, 2012.

[69] M. Baday, A. Cravens, A. Hastie, H. Kim, D. E. Kudeki, P. Kwok, M. Xiao, and P. R. Selvin, “Multicolor super-resolution DNA imaging for genetic analysis,” *Nano letters*, vol. 12, no. 7, pp. 3861–3866, 2012.

[70] B. S. C. Varma, K. Paul, and M. Balakrishnan, “Accelerating genome assembly using hard embedded blocks in FPGAs,” in *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*, Jan 2014, pp. 306–311.

[71] A. M. Aji, L. Zhang, and W. C. Feng, “GPU-RMAP: Accelerating short-read mapping on graphics processors,” in *Computational Science and Engineering (CSE), 2010 IEEE 13th International Conference on*, Dec 2010, pp. 168–175.

[72] C. Liu, T. Wong, E. Wu, R. Luo, S. Yiu, Y. Li, B. Wang, C. Yu, X. Chu, K. Zhao, R. Li, and T. Lam, "SOAP3: ultra-fast GPU-based parallel alignment tool for short reads," *Bioinformatics*, vol. 28, no. 6, pp. 878–879, 2012.

[73] A. Valouev, "Shotgun optical mapping: A comprehensive statistical and computational analysis," Ph.D. dissertation, 2006.

[74] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators," in *2013 23rd International Conference on Field programmable Logic and Applications*, Sept 2013, pp. 1–8.

[75] Xilinx, "Xilinx Products," http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html, 2014, [Online].

[76] D. Sung, J. Somayajula-Jagai, P. Cosman, R. Mills, and A. D. McCulloch, "Phase shifting prior to spatial filtering enhances optical recordings of cardiac action potential propagation." *Ann Biomed Eng*, vol. 29, no. 10, pp. 854–61, 2001.

[77] R. J. Butera, C. G. Wilson, C. A. DelNegro, and J. C. Smith, "A methodology for achieving high-speed rates for artificial conductance injection in electrically excitable biological cells," *IEEE Transactions on Biomedical Engineering*, vol. 48, no. 12, pp. 1460–1470, Dec 2001.

[78] A. D. Dorval, D. J. Christini, and J. A. White, "Real-time linux dynamic clamp: a fast and flexible way to construct virtual ion channels in living cells," *Annals of biomedical engineering*, vol. 29, no. 10, pp. 897–907, 2001.

[79] B. Echebarria and A. Karma, "Spatiotemporal control of cardiac alternans," *Chaos: An Interdisciplinary Journal of Nonlinear Science*, vol. 12, no. 3, pp. 923–930, 2002.

[80] G. M. Hall and D. J. Gauthier, "Experimental control of cardiac muscle alternans," *Physical Review Letters*, vol. 88, no. 19, p. 198102, 2002.

[81] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D. J. Lee, "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, April 2008, pp. 173–182.

[82] K. Pauwels, M. Tomasi, J. D. Alonso, E. Ros, and M. M. V. Hulle, "A comparison of FPGA and GPU for real-time phase-based optical flow, stereo, and local image features," *IEEE Transactions on Computers*, vol. 61, no. 7, pp. 999–1012, July 2012.

[83] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *SC 2006 Conference, Proceedings of the ACM/IEEE*, Nov 2006, pp. 6–6.

[84] K. Pereira, P. Athanas, H. Lin, and W. Feng, “Spectral method characterization on FPGA and GPU accelerators,” in *2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Nov 2011, pp. 487–492.

[85] B. Cope, P. Y. K. Cheung, W. Luk, and S. Witt, “Have GPUs made FPGAs redundant in the field of video processing?” in *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, Dec 2005, pp. 111–118.

[86] R. Inta, D. J. Bowman, and S. M. Scott, “The ”Chimera”: An off-the-shelf CPU/GPGPU/FPGA hybrid computing platform,” *Int. J. Reconfig. Comput.*, vol. 2012, pp. 2:2–2:2, Jan 2012.

[87] S. Bauer, S. Köhler, K. Doll, and U. Brunsmann, “FPGA-GPU architecture for kernel svm pedestrian detection,” in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, June 2010, pp. 61–68.

[88] P. Meng, A. Irturk, R. Kastner, A. McCulloch, J. Omens, and A. Wright, “GPU acceleration of optical mapping algorithm for cardiac electrophysiology,” in *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, Aug 2012, pp. 1558–1561.

[89] M. Jacobsen, Y. Freund, and R. Kastner, “RIFFA: A reusable integration framework for FPGA accelerators,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 216–219.

[90] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[91] J. Teubner, R. Mueller, and G. Alonso, “FPGA acceleration for the frequent item problem,” in *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, March 2010, pp. 669–680.

[92] C. Ttofis and T. Theocharides, “Towards accurate hardware stereo correspondence: A real-time FPGA implementation of a segmentation-based adaptive support weight algorithm,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2012, pp. 703–708.

[93] E. Gudis, G. van der Wal, S. Kuthirummal, and S. Chai, “Multi-resolution real-time dense stereo vision processing in FPGA,” in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 29–32.

[94] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *International Workshop on OpenCL 2013 & 2014*, 2014, pp. 4:1–4:9.

[95] O. Segal, M. Margala, S. R. Chalamalasetti, and M. Wright, "High level programming framework for FPGAs in the data center," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–4.

[96] J. Vasiljevic, R. Wittig, P. Schumacher, J. Fifield, F. M. Vallina, H. Styles, and P. Chow, "OpenCL library of stream memory components targeting FPGAs," in *Field Programmable Technology (FPT), 2015 International Conference on*, Dec 2015, pp. 104–111.

[97] E. Rucci, C. García, G. Botella, A. D. Giusti, M. Naiouf, and M. Prieto-Matias, "Smith-Waterman protein search with OpenCL on an FPGA," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, Aug 2015, pp. 208–213.

[98] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "Real-time 3D reconstruction for FPGAs: A case study for evaluating the performance, area, and programmability trade-offs of the Altera OpenCL SDK," in *Field-Programmable Technology (FPT), 2014 International Conference on*, Dec 2014, pp. 326–329.

[99] M. K. Papamichael, P. Milder, and J. C. Hoe, "Nautilus: Fast automated ip design space search using guided genetic algorithms," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[100] N. Kapre, H. Ng, K. Teo, and J. Naude, "InTime: A machine learning approach for efficient selection of FPGA CAD tool parameters," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 23–26.

[101] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: A response surface-based Pareto iterative refinement for application-specific design space exploration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 12, pp. 1816–1829, Dec 2009.

[102] K. Yu, J. Bi, and V. Tresp, "Active learning via transductive experimental design," in *Proc. 23rd Int. Conf. Machine Learning (ICML)*, 2006, pp. 1081–1088.

[103] S. Dasgupta, "Two faces of active learning," *Theor. Comput. Sci.*, vol. 412, no. 19, pp. 1767–1781, Apr 2011.

[104] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.

[105] H. Su, N. Wu, M. Wen, C. Zhang, and X. Cai, “On the GPU-CPU performance portability of OpenCL for 3D stencil computations,” in *Parallel and Distributed Systems (ICPADS), 2013 International Conference on*, Dec 2013, pp. 78–85.

[106] D. Huang, M. Wen, C. Xun, D. Chen, X. Cai, Y. Qiao, N. Wu, and C. Zhang, “Automated transformation of GPU-specific OpenCL kernels targeting performance portability on Multi-Core/Many-Core CPUs,” in *Euro-Par 2014 Parallel Processing*, 2014, pp. 210–221.

[107] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, “Portable performance on heterogeneous architectures,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*.