Summer 8-15-2020

Domain Specific Computing in Tightly-Coupled Heterogeneous Systems

Anthony Michael Cabrera
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

James McKelvey School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Roger Chamberlain, Chair
Jonathan Beard
Jeremy Buhler
Ron Cytron
William Richard

Domain Specific Computing in Tightly-Coupled Heterogeneous Systems

by
Anthony Michael Cabrera

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2020
Saint Louis, Missouri

Table of Contents

List of Figures . . . . **v**

List of Tables . . . . **viii**

Acknowledgments . . . . **ix**

Abstract . . . . **xiii**

Chapter 1: Introduction **1**
1.1 Research Questions 5
1.2 Contributions 5
1.3 Outline 6

Chapter 2: Background and Related Work **7**
2.1 Domain Specific Computing 7
2.2 Data Integration 8
2.3 Quantitative Characterization Techniques 10
2.4 FPGA 11
2.4.1 Intel HARPv2 13
2.4.2 HARP for Acceleration 14
2.4.3 Designing Kernels with OpenCL 15
2.4.4 High Level Synthesis and Design 19

Chapter 3: DIBS: A Data Integration Benchmarking Suite **21**
3.1 Overview of Benchmark Suite and Integration Tasks 25
3.2 Benchmark Application Descriptions 26
3.2.1 Computational Biology 26
3.2.2 Image Processing 28
3.2.3 Enterprise 31
3.2.4 Internet of Things (IoT) 32
3.2.5 Graph Processing 34
3.3 Characterization of Data Integration Tasks 35
3.3.1 Locality 36
3.3.2 Determinism/Branch Entropy 38

3.3.3 Instruction Mix 38
3.4 Characterization Methods 39
3.5 Results of Characterization 41
3.5.1 Locality 42
3.5.2 Branch Entropy 45
3.5.3 Instruction Mix 48
3.5.4 Discussion 51
3.6 Conclusion 52

Chapter 4: Multi-spectral Reuse Distance: Divining Spatial Information from Temporal Data 54
4.1 The Data Movement Problem 55
4.2 Methods 57
4.2.1 Benchmark Applications 58
4.2.2 Reuse Distance 58
4.2.3 Earth Mover's Distance 60
4.2.4 Memory Footprint 62
4.3 Results and Discussion 63
4.3.1 Spatially Dense Memory Accesses 63
4.3.2 Page Sizing and Utilization 68
4.3.3 Data Layout Transformation 71
4.4 Conclusion 71

Chapter 5: Evaluating Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2 75
5.1 Methods 77
5.1.1 Needleman-Wunsch 78
5.1.2 Description of Each Kernel Version 79
5.1.3 Hardware Design Space Search 83
5.1.4 Shared Virtual Memory 86
5.2 Results and Discussion 87
5.2.1 FPGA Kernel Results 87
5.2.2 Hardware Design Space Search 91
5.2.3 SVM Performance 94
5.3 Conclusion 96

Chapter 6: Designing Domain Specific Compute Systems 98
6.1 Methods 100
6.1.1 Clustering of Domain Applications 100
6.1.2 Evaluating the Hardware 101
6.1.3 Kernel Development 102
6.1.4 Hardware Design Parameters 103
6.2 Kernels 104

6.2.1 ebcdic_txt 106
6.2.2 idx_tiff 108
6.2.3 fix_float 110
6.2.4 edgelist_csr 111
6.2.5 2bit_fa 113
6.2.6 fa_2bit 115
6.3 Other Design Considerations 117
6.3.1 Overlapping Data Transfer and Execution 117
6.3.2 Visualizing the Hardware 118
6.3.3 Widening the Data Type 121
6.4 Results 123
6.4.1 Design Space Search Sweeps 124
6.4.2 MWI versus SWI Implementations 135
6.4.3 Results of Widening The Datatype 141
6.5 Conclusion 144

Chapter 7: Conclusion and Future Work 145
7.1 Future Work 147

References 150

Vita 163

List of Figures

Figure 2.1: FPGA Block Diagram 12

Figure 2.2: HARPv2 vs. PCIe Card FPGA 14

Figure 2.3: NDRange vs. SWI Block Diagram 17

Figure 3.1: Spatial Locality Measure 42

Figure 3.2: Cumulative Sum of Memory References Across Strides 43

Figure 3.3: Temporal Locality Measure 45

Figure 3.4: Cumulative Sum of Memory References Across Reuse Distances . . . 46

Figure 3.5: Branch Entropy Measure 47

Figure 3.6: x86-64 Static Instruction Mix 48

Figure 3.7: x86-64 Dynamic Instruction Mix 49

Figure 3.8: AArch64 Dynamic Instruction Mix 50

Figure 4.1: Reuse Distance Basics 59

Figure 4.2: Reuse Distance Signatures 64

Figure 4.3: Intuition for Mass Shift Proof 67

Figure 4.4: Memory Footprint Results 72

Figure 4.5: Earth Mover's Distance Results 72

Figure 5.1: Dynamic Programming Example 79

Figure 5.2: Diagonal Parallelism Illustration 81

Figure 5.3: Shift Register Example 83

Figure 5.4: Staircase Shift Registers 84

Figure 5.5: Exploiting Diagonal Parallelism and Staircase Shift Registers 85

Figure 5.6: NW Kernel Hardware Design Space Search 91

Figure 5.7: Shared Memory Benefit 94

Figure 6.1: k-means Clustering of the DIBS Applications 101

Figure 6.2: Width vs. Depth Execution Models 102

Figure 6.3: Confirmation of Shared Memory Benefit 118

Figure 6.4: CDFGs and Cycle Schedules 118

Figure 6.5: `idx_tiff` SWI Results 124

Figure 6.6: `fix_float` SWI Results 125

Figure 6.7: `edgelist_csr` SWI Results 125

Figure 6.8: `2bit_fa` SWI Results 126

Figure 6.9: `ebcdic_txt` SWI Results 127

Figure 6.10: `fa_2bit` SWI Results 128

Figure 6.11: `ebcdic_txt` MWI Results 129

Figure 6.12: `fix_float` MWI Results 129

Figure 6.13: `edgelist_csr` MWI Results 130

Figure 6.14: `2bit_fa` MWI Results 130

Figure 6.15: `idx_tiff` MWI Results 132

Figure 6.16: `fa_2bit` MWI Results 133

Figure 6.17: CPU vs. MWI vs. SWI Results 136

Figure 6.18: Data Type Vectorization Design Space Search 139

Figure 6.19: Input File Size Sweep 141

List of Tables

Table 3.1: Data Integration Task Classification 26

Table 3.2: Experimental Machine Specifications 40

Table 3.3: Throughput Results 41

Table 5.1: HARPv2 vs. PCIe Card Results 88

Table 5.2: Hardware Design Space Search for NW Kernels 90

Table 6.1: MWI Design Knob Trend 131

Table 6.2: Most Performant MWI and SWI Configurations 137

Table 6.3: Utilization and Performance Results for Data Type Vectorization . . 140

Acknowledgments

Though this dissertation bears my name as sole author, this process has hardly been a solo endeavor. There are a number of people who have been in my corner throughout my life, and without them, this work would not have been possible.

First, I would like to thank my parents, Patrick and Annabelle Cabrera, and my brother, Chris Kramp, who have supported me throughout all of my endeavors. Through the baseball games, orchestra concerts, and all of my academic pursuits, they were always behind me, supporting me every step of the way and letting me know how proud they were. I could not have gotten to where I am today without their love and support.

My advisor, Dr. Roger Chamberlain, has been invaluable in so many ways during my PhD journey. He took me on as a graduate student in my second year, and trusted and respected me as if I had already been his student for years. He has been nothing short of extraordinary as a mentor, and I am thankful for his guidance. Even if it was staying up late for a paper deadline, Roger would always make time for me when I asked, and that is something that I will always appreciate.

There have also been a number of other faculty and colleagues that I would like to acknowledge. My former research advisor, Dr. Viktor Gruev, gave me my first research opportunity as an undergraduate and is what ultimately lead me to pursue a PhD. My committee members, Drs. Jonathan Beard, Jeremy Buhler, Ron Cytron, and William Richard have all played pivotal roles in my formation as a researcher, even before they agreed to serve on my committee. My colleagues in the CSE department at WUSTL, including Clayton Faber and Drs. Steve Cole, Missael Garcia, Tim York, and many others, have been incredibly insightful, and have helped shaped my development as a researcher.

I must also thank the faculty at Hendrix College, specifically Drs. Gabe Ferrer, Karen Griebling, Damon Spayde, and Todd Tinsley, for their enthusiasm for teaching. They made learning the hard stuff fun, and they showed me what happens when faculty truly invest in their students' success.

During the summers of my third and fourth years, I had the opportunity to intern and work with some great people. I want to thank Dr. Jonathan Beard for taking me on as a research intern at ARM, and for leading me through the nasty details of modern memory subsystems and the field of High Performance Computing. Thanks also to Dr. Christine Harvey, John Hilbing, Andrew van Overloop, Rose Schneider, and Dan Aiello for giving me the opportunity to work at MITRE, and their constant support during my time as an intern and beyond.

For all of the people I have had the pleasure of calling friend over the past few years, I am also thankful. To Marcus Brown, Emily Murphy, Katy O'Keefe, and Page Vick, I am glad to have gotten to know you from the third grade to the present, and I am thankful that we still continue to keep in touch and lift each other up. To my Hendrix friends, namely Stephen Borutta, Ernest Perez, and Drs. Sam Bondurant, William Haden Chomphosy, Sam Fullerton, Lance Riley, Spencer Sanson, and Erik Urban, our friendships are ones that I will continue to cherish.

Finally, I want to thank my wife, Jillian Smith, for whom I am eternally grateful. She has shown me a level of affection, patience, and support that I had not known was possible. She has been a wonderful cat mom to Lil' Sebastian and Ann Purrkins. She welcomed me into her life without hesitation, and I am grateful to have a second family in Jane, Ralph, Ben, and Theo Smith and Shelley Bryant. Jillian has been there for me during my highest highs and lowest lows, and she has never stopped professing how proud she is and how much she believes in me. This achievement is just as much hers as it is mine. I love you, Jillian, and I am fortunate to have been blessed with such a loving and supportive partner.

Anthony Michael Cabrera

Washington University in Saint Louis

August 2020

Dedicated to Mom, Dad, and Jillian.

ABSTRACT OF THE DISSERTATION

Domain Specific Computing in Tightly-Coupled Heterogeneous Systems
by
Anthony Michael Cabrera
Doctor of Philosophy in Computer Engineering
Washington University in St. Louis, 2020
Professor Roger Chamberlain, Chair

Over the past several decades, researchers and programmers across many disciplines have relied on Moores law and Dennard scaling for increases in compute capability in modern processors. However, recent data suggest that the number of transistors per square inch on integrated circuits is losing pace with Moores laws projection due to the breakdown of Dennard scaling at smaller semiconductor process nodes. This has signaled the beginning of a new "golden age in computer architecture" in which the paradigm will be shifted from improving traditional processor performance for general tasks to architecting hardware that executes a class of applications in a high-performing manner. This shift will be paved, in part, by making compute systems more heterogeneous and investigating domain specific architectures. However, the notion of domain specific architectures raises many research questions. Specifically, what constitutes a domain? How does one architect hardware for a specific domain?

In this dissertation, we present our work towards domain specific computing. We start by constructing a guiding definition for our target domain and then creating a benchmark suite of applications based on our domain definition. We then use quantitative metrics from the literature to characterize our domain in order to gain insights regarding what would be most beneficial in hardware targeted specifically for the domain. From the characterization, we learn that data movement is a particularly salient aspect of our domain. Motivated by this fact, we evaluate our target platform, the Intel HARPv2 CPU+FPGA system, for architecting domain specific hardware through a portability and performance evaluation. To guide the creation of domain specific hardware for this platform, we create a novel tool to quantify spatial and temporal locality. We apply this tool to our benchmark suite and use the generated outputs as features to an unsupervised clustering algorithm. We posit that the resulting clusters represent sub-domains within our originally specified domain; specifically, these clusters inform whether a kernel of computation should be designed as a widely vectorized or deeply pipelined compute unit. Using the lessons learned from the domain characterization and hardware platform evaluation, we outline our process of designing hardware for our domain, and empirically verify that our prediction regarding a wide or deep kernel implementation is correct.

Chapter 1

Introduction

Over the past several decades, researchers and programmers across many disciplines have relied on two trends in computing. The first is that the number of transistors on integrated circuits doubles roughly every two years. The other is that the relationship between a transistor's feature size and power consumption and switching speed are constant. Together, these insights have guided the semiconductor industry towards creating smaller transistors to lower power consumption, raise clock frequencies, and increase compute capability per unit area.

These two observations, known as Moore's law [115] and Dennard scaling [12], respectively, have been the driving force behind aggressively scaling semiconductor process technologies. This allows for increases in compute capability in modern processors. However, recent data and results show that the number of transistors per square inch on an integrated circuit is losing pace with Moore's law's projection, and that second-order effects of extreme transistor scaling are cancelling out the effect of lower power consumption for smaller transistors. The impending end of Moore's law due to the breakdown of Dennard scaling will mark the end of an era characterized by relying on transistor scaling and increased clock frequencies to improve performance gains.

In response to this, modern computing systems are becoming more architecturally diverse. Architectural diversity includes any type of processing element within a computing system other than the traditional processor core. The goal of introducing heterogeneity into a computing system is to accelerate compute tasks that would otherwise be handled by a traditional processor core. One of the earliest examples of this is Intel's 8087 numeric data processor that served as a co-processor dedicated to speeding up floating point computations [102]. In the spirit of this type of hardware acceleration, modern systems are increasingly adding heterogeneity through incorporating hardware accelerators i.e., digital signal processors (DSPs), graphics processing units (GPUs), or field-programmable gate arrays (FPGAs). GPUs in particular are gaining traction, finding use in applications such as convolutional neural networks [74], molecular simulations [1], and protein sequence alignment [132].

In their 2017 ACM A.M. Turing Award speech, John Hennessey and Dave Patterson espoused the importance of architecturally diversity. They address the challenges that arise from the end of Moore's law and Dennard scaling and use this to signal a new "golden age of computer architecture". However, the onus of research in a post-Moore's law landscape was not solely placed on computer architects. In order to create effective compute solutions, researchers must embrace working across the hardware and software stack through hardware-software co-design. Additionally, they suggest that the path to a post-Moore's law world is paved, in part, by domain specific computing. This key idea means shifting the paradigm of improving general purpose processors that are good at many compute tasks towards building hardware and surrounding infrastructure for processors that do fewer things but in a high-performing manner. Between a spectrum bounded by general purpose processors and application specific integrated circuits (ASICs), domain specific architectures would occupy the middle ground by developing an architecture to accelerate a given domain or class of applications.

One way to help facilitate this shift is by using field-programmable gate arrays (FPGAs) as a platform for domain specific computing. FPGAs are a special type of integrated circuit that can be programmed to implement a desired application in hardware. Historically, FPGAs have been used for prototyping hardware or microarchitectures or as a lower-cost solution to application specific integrated circuits (ASICs). The widespread use of FPGAs, though nascent, has been burgeoning in recent years due to increased interest in industry. This forward progress is reflected by companies like Amazon and Microsoft equipping their data center nodes with FPGAs [107, 25, 5] and Intel acquiring FPGA manufacturer Altera. Additionally, there is a growing research trend toward harnessing the reconfigurability of FPGAs towards accelerating applications like neural networks [23, 43, 143], biocomputation [62, 89, 94], and many other applications [88, 117, 125, 142, 150]. One of the key ideas we present in this thesis, though, is designing techinques and tools using FPGAs to accelerate *classes* of applications, rather than individual applications.

A common way to incorporate hardware accelerators like GPUs and FPGAs into a computer system is to attach them through a PCIe bus. Accelerators attached in this way, though, incur considerable overhead for data movement and keeping the memory between the host and accelerator systems coherent. Recently, Intel has developed a system to address these issues by incorporating both a multicore Xeon CPU and Arria 10 FPGA into the same chip package and connected via high-speed and coherent interconnect. This particular project is known as the Heterogeneous Accelerator Research Program, or HARP, and it is the platform we use in this dissertation for developing domain specific hardware. We describe the HARP hardware futher in Section 2.4.1.

One of the steepest barriers to using FPGAs, though, is expressing a design in the first place using traditional hardware description languages (HDLs) like VHDL and Verilog. This requires the ability to orchestrate a design at the logic gate level and at a clock-cycle granularity. The HARP system is no exception to this rule. A current research direction in lowering the barrier is High Level Synthesis (HLS), which allows a programmer to express a kernel of computation in a higher level language like C or C++ for deployment onto an FPGA. (HLS is described further in Section 2.4.4.) This circumvents the problem of having to learn an HDL to express a kernel and its low level interfaces, reduces the amount that a programmer has to understand about FPGA microarchitecture, and abstracts away the lower level details of using FPGAs. In addition to being able to author designs using an HDL, Intel has provided the infrastructure to use the Intel FPGA OpenCL SDK for HLS FPGA development [61]. While there have been recent publications targeting the HARP system with a traditional FPGA design flow [4, 28, 117, 122, 135, 144], not much is known about the experience, feasibility, and performance of targeting a HARP system using OpenCL. Our work towards this combination is one of the contributions of this thesis; we show how to effectively utilize properties of the Intel HARPv2 (the second iteration of the Intel Heterogeneous Architecture Research Platform) and evaluate its effectiveness as a domain specific computing solution.

The overarching goal of this thesis is to develop a methodology for identifying a domain and to architect performant hardware for that domain. We first define the domain of data integration as a case study and define this domain using both qualitative and quantitative methods. From there, we will evaluate the hardware design process and performance of the Intel HARPv2 system. We target the HARPv2 system using the Intel FPGA SDK for OpenCL for hardware development, which allows us to author designs in a higher level language. We develop hardware design strategies specific to our target domain, as well as strategies for generally using OpenCL to target the HARPv2 system. In order to make quantitative hardware design choices, we create a tool called multi-spectral reuse distance, whose outputs are used as features to cluster our applications and create sub-domains of our original domain. We posit that these sub-domains represent whether a particular application would benefit more from a widely vectorized or deeply pipelined implementation, and then empirically verify our position.

1.1 Research Questions

In this dissertation, we make progress towards answering the following questions:

- How does one define an application domain?
- How does one architect performant hardware on the Intel HARPv2 platform using OpenCL?
- How does one effectively architect domain specific hardware?

1.2 Contributions

In addressing these questions, we make the following specific contributions:

- The Data Integration Benchmarking Suite (DIBS), a suite of applications that represent data integration workloads across a variety of different application domains [22].
- Using known profiling techniques to quantitatively characterize the DIBS applications [22].
- A novel tool to measure the temporal and spatial locality of a given application [20].

- A performance and portability evaluation between OpenCL kernels synthesized for FPGAs attached via PCIe card and the Intel HARPv2 system [17].
- A method using control data-flow diagrams to inform design decisions at the OpenCL kernel level.
- A method for design space enumeration and search for the two OpenCL execution models [17, 19].
- Empirically-based techniques for designing kernels for the Intel HARPv2 platform [17].
- A set of OpenCL kernels designed to target the Intel HARPv2 platform [19].
- A method leveraging an unsupervised clustering algorithm to predict the most performant OpenCL execution model for a given kernel, and empirically validating the prediction [19].

1.3 Outline

The rest of this dissertation is structured as follows: Chapter 2 will cover related work and background information pertaining to domain specific computing, FPGAs, and HLS. Chapter 3 will introduce the domain of data integration, the Data Integration Benchmarking Suite, and our initial characterization of the applications. Chapter 4 will present multi-spectral reuse distance, a novel tool and methodology to quantitatively capture both spatial and temporal locality. Chapter 5 will present a performance and portability evaluation of the Intel HARPv2 system. Chapter 6 will outline our method of domain specific hardware design. Chapter 7 will conclude this dissertation and present our directions for future work.

Chapter 2

Background and Related Work

2.1 Domain Specific Computing

Early work in domain specific computing most resembling our approach has been done by Cong et al., in which they propose a heterogeneous processor consisting of both fixed cores and configurable fabric and perform a workload characterization on a set of domain applications in order to determine which components of those applications should be implemented on the various compute units [30]. Our work differs in that we are both targeting a different execution platform (the Intel HARPv2) and we are using HLS for application expression.

With the rise in machine learning, particularly neural networks, there has been hardware created for accelerating such workloads. The Tensor Processing Unit (TPU) [69] and the Intel Nervana NNP-T [140] are examples of such hardware for training and inference, respectively. Both feature custom multiply-accumulate units to handle GEMM operations that dominate convolutional neural network applications. Our work aims to address domains that aren't so clearly dominated by one aspect of the computation.

2.2 Data Integration

In this dissertation, the domain that we primarily focus on is that of data integration. The issue of data integration is universal to anyone working on data driven applications and can be troublesome to deal with, often taking time comparable to the actual computation of interest [90, 91]. In general, data integration is the process of taking input data in some initial form and shaping and preparing it into a suitable form required by downstream analyses, e.g., a genome sequence application that requires a *.fasta* file be converted to *.2bit*, or transforming the bounding box labels from the *MS-COCO* training dataset to the *KITTI* format because the front-end of the target neural network training system requires it. While definitions vary, we will use the definition presented in the Data Integration Benchmark Suite (DIBS) in Chapter 3.

The data integration problem has received considerable attention already in the research community. Quoting from Kandel et al. [70], "In spite of advances in technologies for working with data, analysts still spend an inordinate amount of time diagnosing data quality issues and manipulating data into a usable form. This process of 'data wrangling' often constitutes the most tedious and time-consuming aspect of analysis." Dasu and Johnson indicate that data reformatting and cleaning accounts for up to 80% of the development time and cost in data warehousing projects [34].

Customized Domain Specific Languages (DSLs) and graphical user interfaces (GUIs) exist that are designed explicitly for describing data transformation workflows. Examples from the ETL literature include AJAX [46], Potter's Wheel [108], ARKTOS [128], BPEL [39], Wrangler [71], and OptiWrangler [124]. Work has also considered finding the right transformations, helping address issues of data integrity and consistency. Guo et al. [53] describe a model that proactively suggests data transforms. In addition, there are commercial systems available both to specify the workflows and to execute them (either on traditional multicores or, more recently, on map-reduce clusters). Examples here include IBM's InfoSphere and Informatica.

There are also a host of systems aimed at scientific data (e.g., see [2, 13, 16, 36, 101]). While there is significant disparity of data formats in many disciplines, biology [101] for example, other disciplines, such as ecology [13, 95], have a stronger culture of data description via XML and semantic ontologies, enabling a higher degree of automation in the specification of data transformations.

While there are any number of ways that data transformations can be specified, our general interest is in helping research groups compare implementations of systems that execute data transformations by providing a baseline implementation and its accompanying characterization. The classic way to do this is via a benchmark suite. Examples of benchmark suites in other fields include: the SPEC family[1], including SPEC CPU2017 and SPECjvm2008; MiBench [56], for embedded systems; PARSEC [10], for parallel applications; MediaBench [81], for multimedia computations; Rodinia [26], for heterogeneous computing with GPUs; HiBench [58], for map-reduce data processing; MachSuite [109], for accelerator architectures; and CommBench [138], for network processing. We will use this benchmarking suite as our target domain for architecting domain specific hardware.

Poess et al. [106] have developed an enterprise-centric data integration benchmark, but do not speak to the more general data integration audience. Additionally, the characterization of their benchmark suite is limited only to scalability and runtime. To the best of our knowledge, we present the first benchmark suite that broadly characterizes data integration tasks.

[1] `http://spec.org`

2.3 Quantitative Characterization Techniques

Characterization of both temporal and spatial locality has a long history [38]. Metrics from the literature include [31, 55, 75, 78, 119, 120, 121, 133].

Reuse distance–defined initially by Mattson et al. [92] as stack distance–is frequently used as a measure of temporal locality. For example, Weinberg et al. [136] define a temporal locality measure that is the area under the reuse distance curve, with the reuse distance expressed using a log scale. This formulation has been used for the characterization of various benchmarks [22, 27, 99, 109, 127]. Reuse distance has been compared with spatial locality by previous authors [52, 147]. All of these authors owe the gestalt of their works to the observations of Spirn and Denning [121] who made some of the earliest observations of program locality. Gu et al. [52] observed reuse distance to be a measure of both temporal and spatial locality. They used reuse distance as a measure of spatial locality as we do, by altering the granularity of the data block size. They reason that varying the block size leaves temporal locality unchanged, so distinctions between two block sizes are due to spatial locality. These authors also propose a spatial locality score SLQ. Gupta et al. [55] propose a statistical model based on the idea of "near-future windows sizes." In contrast to this work, our methodology uses Earth Mover's Distance (EMD) [110] to provide a metric that gauges spatial locality when moving histograms of multi-spectral temporal reuse data.

While the approach we espouse in Chapter 4 is driven by empirical data, others have taken a more theoretical approach, using the cache oblivious model to determine data locality [118] and graph theoretic approaches (interval graphs) [9]. These methods are of [9]) the search for multiple cliques over the entire stream of allocations and accesses of a program. While these methods are intended to inform cache behavior, our methods are intended to be more general. We also intend to be approximate; we feel that for many cases in real world decisions, a good fast answer is far better than a too-late exact answer.

Within Chapter 4, we make the claim that prefetching of data is a difficult problem. Mittal [96] provides an excellent overview of contemporary prefetching methods and results. Plainly speaking, the dynamic random access main memory (DRAM) of modern computers is yet another level of cache, managed by the operating system. This DRAM can be composed of many different types of memory technology, as well as having NUMA [79] characteristics. The authors make no claims of use directly as a model for prefetching, however, the proposed modeling methodology could be used to determine the optimal granularity of prefetch (in the case of memory systems) and also on the selection of cost function to drive the control process. Granularity of statistical prediction has a well known relationship with a prediction's accuracy [98] (e.g., very detailed predictions with more degrees of freedom often have more uncertainty) and we make no claim to this relationship, but we do hope that this method provides a means to more optimally use coarse grained prediction effectively (through better page sizing). The problem of data placement within a tiered and NUMA system is by no means new, and heavily related to data to disk optimization problems solved as examples in [80]. Regarding domain specific hardware design, we will use the technique outlined in Chapter 4 to generate feature data to identify sub-domains within our target data integration domain.

2.4 FPGA

Field Progammable Gate Arrays (FPGAs) are integrated circuits that include programmable logic blocks, hardened logic blocks such as Digital Signal Processors (DSPs) and floating point units (FPUs), block RAMs (BRAMs, and referred to as M20K blocks for Intel FPGAs), and reconfigurable routing circuitry to connect these components together and to hardened I/O logic in order to interface with external hardware. A block diagram of an FPGA is shown in Figure 2.1.

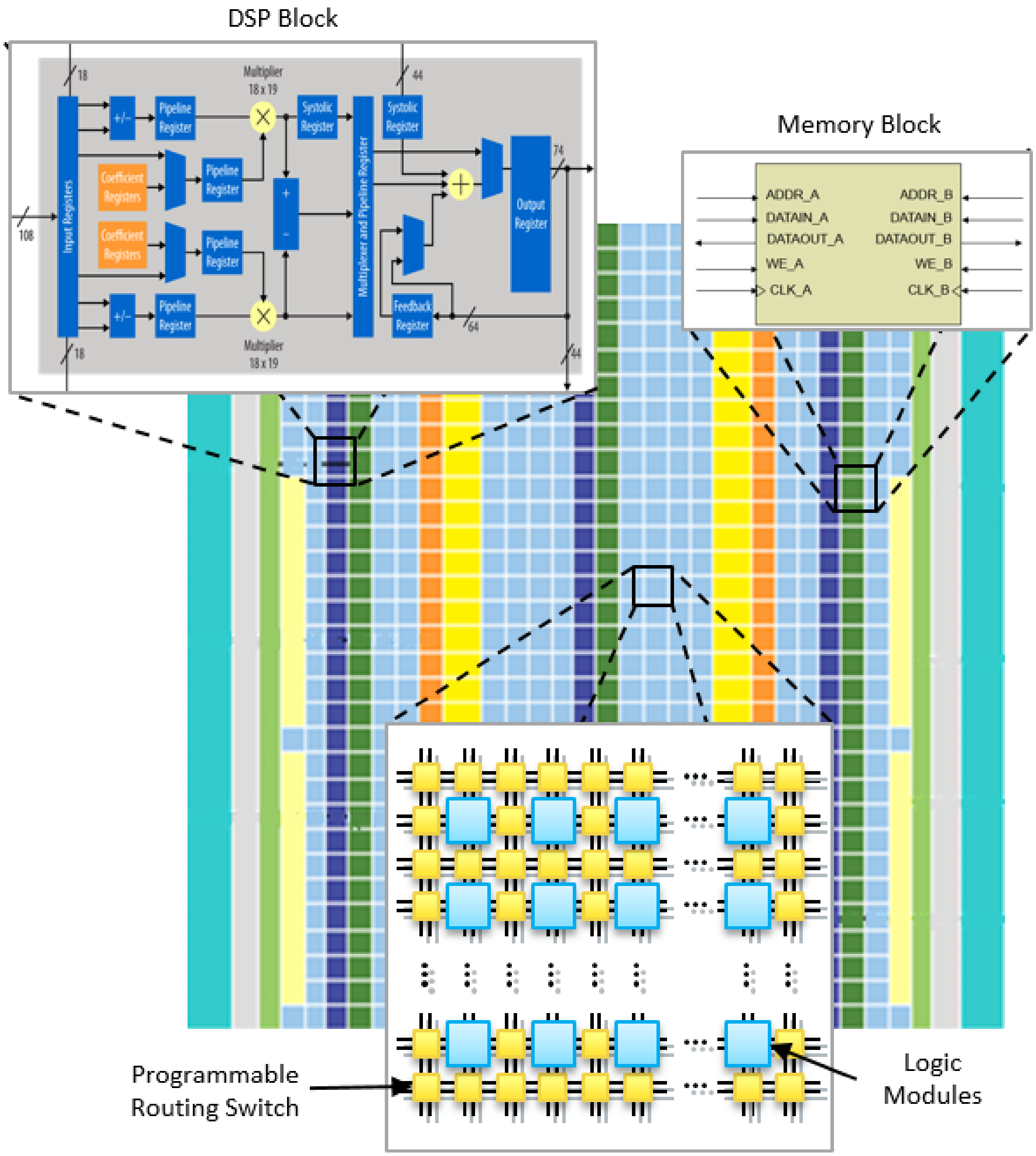


Figure 2.1: A block diagram of an Intel FPGA [60].

FPGAs tend to occupy the middle ground between general purpose CPUs and application specific integrated circuits (ASICs) in terms of programmabilty, performance, and power consumption. FPGA developers traditionally design hardware using Hardware Description Languages (HDLs) such as VHDL or Verilog. This allows them to tailor hardware to a specific application. This usually results in better performance than CPUs. The effective use of hardware that is specific only to the problem also leads to lower power consumption.

2.4.1 Intel HARPv2

The second iteration of the Heterogeneous Architecture Research Platform (HARPv2) system incorporates a 14 core Intel Broadwell Xeon CPU with an Intel Arria 10 GX1150 in the same chip package, where both the CPU and FPGA share the same memory. The HARPv2 sstem serves as the target platform in this work. Relative to the Stratix V GX A7 in the HARPv1 system, the FPGA in HARPv2 has 1.06 times more M20K blocks, 1.82 times more logic blocks and registers, 5.93 times more DSP blocks, and is located on the same chip package as opposed to a different socket. Integrating the CPU and FPGA on the same package is different from traditional FPGA accelerator solutions that are connected via PCIe slot or on their own development board. A block diagram comparing the Intel HARPv2 system and the traditional PCIe card version are shown in Figure 2.2.

The FPGA is connected to the CPU through three physical channels: one through Intel's QuickPath Interconnect (QPI), and the other two through PCIe lanes. Intel also provides the low level interface hardware for the FPGA through their Board Support Package (BSP). Faict presents an excellent overview of the HARP system in [41].

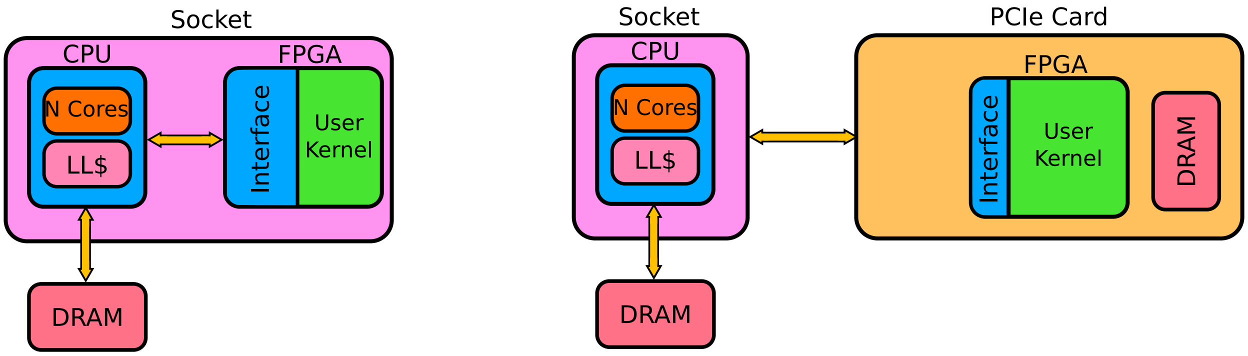


Figure 2.2: Block diagrams of (left) the Intel HARPv2 system and (right) a traditional FPGA solution using a PCIe card.

2.4.2 HARP for Acceleration

Since its inception, there have been many projects that have demonstrated the benefits of using the HARP system in a variety of different applications and domains. Podili et al. use the HARP as their experimental system in using Winograd FFTs to speed up convolutional layers in Convolutional Neural Networks [105]. Alves et al. utilize the low-latency QPI channel for collision detection algorithms to demonstrate the HARP's feasibility for real-time applications [4]. Sidler et al. exploit the shared memory feature of the HARP system to reduce superfluous data movement in pattern matching for databases [117]. Stitt et al. develop a scalable window generator architecture for sliding window applications, which are a common pattern in FPGA design, to take advantage of the increased memory bandwidth in the HARPv2 system and reported future memory bandwidth increases for FPGAs [122]. Wang leverages the tighter coupling of CPU and FPGA in the HARP system as a heterogeneous platform for accelerating graph processing [135]. In all of these cases, though, custom RTL is written to express the hardware and low-level interfaces for the HARP system, which is a skill not generally in the toolbox of the modern software developer. In Chapter 5, we leverage OpenCL to allow a description of accelerator functions in C, which is a higher level language than an RTL description. Specifically, we will evaluate the performance and portability of OpenCL kernels that were originally intended for FPGAs attached via PCIe card.

2.4.3 Designing Kernels with OpenCL

Traditionally, programming an FPGA requires domain specific knowledge of digital systems design, which is not a skill of most software developers. Also, the designs are historically expressed at the register-transfer level (RTL) using languages like VHDL, Verilog, or SystemVerilog. High level synthesis (HLS) addresses both of these issues. Specifically, we use the HLS framework provided by the Intel FPGA SDK for OpenCL [61].

This SDK is an implementation of the OpenCL standard API that allows for programmers to author both host and device code in a high level language. The SDK provides a runtime environment (RTE) that controls the execution of kernels on the FPGA. All of the low level interfaces and drivers that facilitate the interaction between the host and target device(s) are included in the BSP, traditionally provided by the board manufacturer. In the case of the HARPv2 platform, a pilot BSP is provided by Intel. The Intel OpenCL FPGA SDK provides an offline compiler that takes an OpenCL kernel, creates an HDL representation of that design in Verilog, synthesizes that into logical FPGA elements (RTL), maps that design into FPGA components (e.g. logic blocks, I/O blocks), places the mapped design onto the target FPGA, and routes the design.

HLS effectively allows a programmer to express a computational kernel at a higher abstraction level than RTL, allowing the programmer to focus on the functional specification. This kernel is then translated into an equivalent RTL description by the Intel tools which will be fed into the traditional FPGA synthesis flow. At this point, a bitstream to program the

FPGA is generated as if the design is written in Verilog to begin with, i.e., the resulting Verilog is synthesized, placed, and routed to generate the bitstream. From this point forward, we will refer to the tools that take the OpenCL specified kernel to perform the high level synthesis, logic synthesis, place, and route steps collectively as the *hardware compiler*.

OpenCL FPGA Execution Models

There are two main execution models for designing an OpenCL kernel to target synthesizable FPGA hardware: the Multiple Work-Item and Single Work-Item (SWI) models. MWI is also known as the NDRange (NDR) execution model. These models are pictorially described in Figure 2.3.

The MWI model expresses kernels through specifying a global amount of work (i.e. global work size) to do in (up to) a 3-dimensional space, and a local amount of work to do (i.e. local work size) in that same space that will be scheduled for execution on a processing element. In Figure 2.3, the NDRange kernel is specified in a 1-dimensional space. Kernel execution, then, must be enqueued from the host side to make sure all global work items will be executed. Each work item is then scheduled by a hardware scheduler on the FPGA side. This execution model is frequently used on GPUs, whose compute units are comprised of many SIMD vector units that are well suited to take advantage of data-level parallelism. The Single Work Item (SWI) model expresses kernels by setting the global and local work size to 1 in all dimensions so that all computation is handled by a single work item. In both cases, a custom pipeline is created for computation, as shown in Figure 2.3.

Intel recommends using the SWI model if the target kernel contains many loop and memory dependencies [60]. This allows the offline compiler to have a global view of all computation so it can account for dependencies when constructing a custom pipeline. Ideally, iterations can

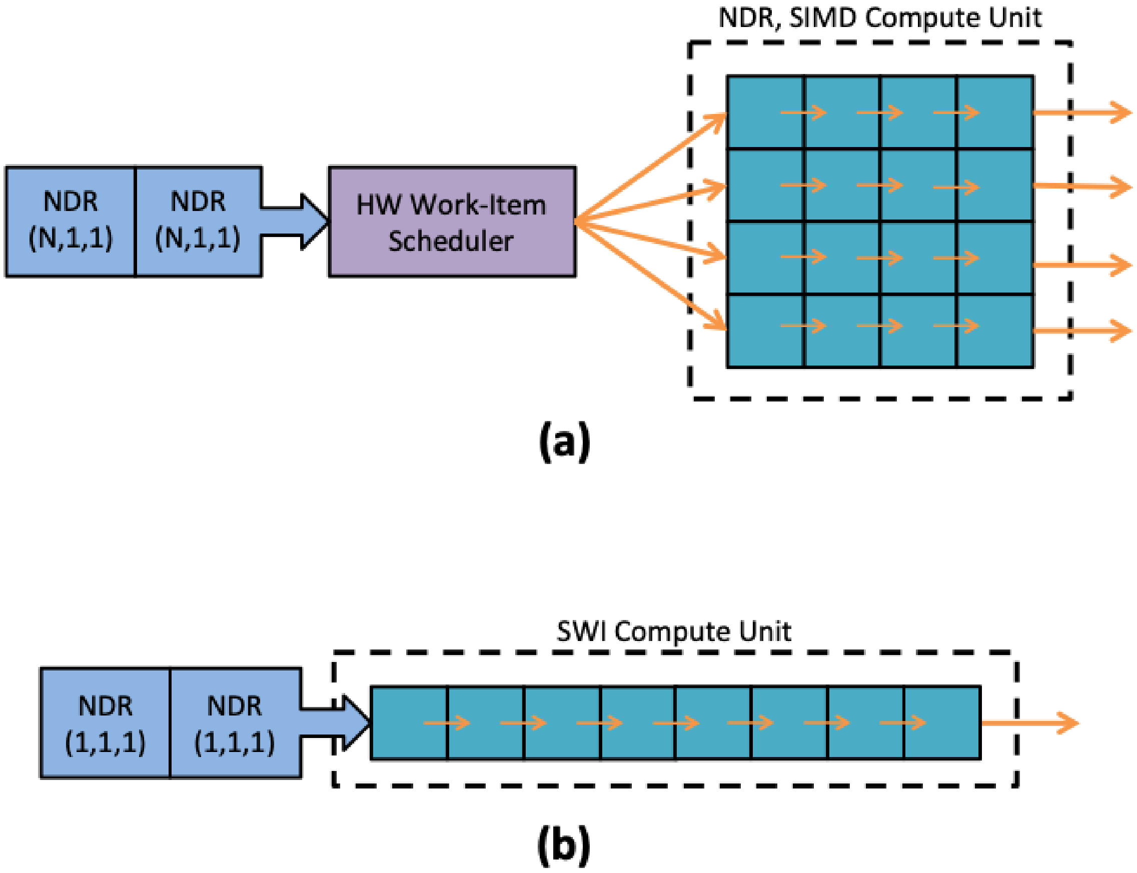


Figure 2.3: (a) The NDRange model relies on using multiple work items to perform kernel computations. Each of these work items must be scheduled for execution onto the compute unit by a hardware scheduler implemented on the FPGA. In this case, there are two instances of some 1D NDRange kernel enqueued for execution that each have N global work items that need to be executed. The pipelined compute unit in this case has been vectorized by a factor of 4. (b) The Single Work Item kernel uses only one work item and thus does not need a hardware scheduler. Single work items rely on pipelining to exploit instruction-level parallelism and resolving of loop and memory dependencies between iterations without the use of costly memory barriers.

then be launched every clock cycle. Additionally, fine-grained sharing between loop iterations in an NDR kernel requires an intricate mechanism that involves local memory and barriers, and this leads to suboptimal kernel performance. Zohouri [148] takes this further and says that NDRange kernels should only be employed if loops cannot be fully pipelined due to variable exit conditions, complex loop-carried dependencies, or random external memory accesses. For all other cases, they recommend that the SWI model should be employed. However, the choice between the two models is non-trivial, as evidenced by Jiang et al. [64].

OpenCL FPGA in the Wild

Though there are many examples in the literature of using HLS frameworks to program FPGAs, we highlight instances that are most relevant to this work. Jin and Finkel evaluate the performance of varying the number of replicated compute units for an OpenCL kernel that computes an MD5 hash [66]. Sanaullah and Herbordt use the Verilog created by the Intel FPGA OpenCL SDK offline compiler for an OpenCL kernel that describe a fast Fourier transform, apply code structure optimizations, and outperform vendor IP-based designs while also being able to fit this modified design into existing FPGA solutions that use FFTs [112]. Zohouri et al. focus on the portability aspect of using OpenCL kernels intended for GPUs on FPGAs [149]. Sanaullah et al. propose a framework for describing OpenCL kernels that relies on the stacking of optimizations that should apply generally to all kernels [114]. However, they prescribe that the most performant version of any OpenCL kernel will use the SWI design paradigm and ignore the MWI paradigm. In this work, we explore both paradigms and find that the design choice between the two is non-trivial. Jin and Finkel perform a hardware design space search [68] similar to this work, but do not show the effect of varying the vectorized data types and their interaction with the available coarse-grained knobs. Additionally, they do not show the impact of scaling the input size.

We will address both of these issues in Chapter 6. In all cases, none of these works target the Intel HARPv2 CPU+FPGA platform using OpenCL. This work specifically evaluates the portability and performance of OpenCL FPGA system on the HARPv2 system, in which the CPU and FPGA are located on the same chip package and share a common memory. There is a only a small body of literature showing case studies that use the Intel HARPv2 platform in this way [17, 40, 42, 123, 145], and we intend for this work to add to the existing literature in order to inform more performant OpenCL-based designs for the Intel HARPv2 system.

The kernels used in Chapter 5 to evaluate performance and portability are sourced from work done by Zohouri et al. that aims to evalute and optimize OpenCL kernels taken from the Rodinia suite [26] to evaluate the effectiveness of FPGAs in high performance computing applications [148, 149].

2.4.4 High Level Synthesis and Design

In addition to the Intel FPGA OpenCL SDK, there are other ways to leverage High Level Synthesis (HLS) and design to target FPGAs. One of the earliest HLS languages that preceeded OpenCL for FPGAs was the Streams-C language and compiler, implemented by Gokhale et al. [50], that allowed programmers to author streaming kernels in a C-based language. They also quantified the tradeoffs between performance and ease of programmability using HLS. LegUp, developed by Canis et al., takes a C program as input and automates the process of finding segments of code that can be accelerated on an FPGA [24]. Bachrach et al. develop a hardware construction language called Chisel in the Scala programming language in order to design hardware using object-oriented principles and functional programming [6]. It is important to make the distinction that Chisel is not a “C-to-Gates” form of HLS; this solution allows for a more expressive description of hardware using higher level ideas like object-oriented programming.

Chapter 3

DIBS: A Data Integration Benchmarking Suite

Generating and analyzing big data are tasks encountered by many scientists and researchers in various disciplines. Social networks, computational biology, sensor data, and entrepreneurial records are just a small sample of the range of applications that encounter various and extensive data streams. It is generally well understood that big data is voluminous and prevalent in the research and industrial communities alike, but what is less studied are all of the steps that must be completed even before the primary computation (e.g., number crunching and analysis of the data) can begin. Specifically, there is often a non-trivial amount of time, effort, and resources that are spent towards retrieving and preprocessing big data sources.

This problem of taking data in some initial form and transforming it into a desired one comes in several flavors. It might involve rearranging fields, changing the form of expression of one or more fields (e.g., translating one character set into another, such as ASCII to UTF or EBCDIC to ASCII), altering the boundary notation of records and/or fields (e.g., moving between comma-separated and fixed-length fields), encrypting or decrypting records and/or fields, parsing non-record data and organizing it into a record-oriented form, etc. We define this problem, collectively, as *data integration*.

In the business community, this is part of the Extract, Transform, and Load (ETL) process, specifically the transform step. Another phrase that is often incorporated into the scope of data integration is data cleansing, which includes notions of checking data integrity, (re)constructing missing fields, outlier detection, type checking (e.g., does a numeric field contain non-numeric contents?), etc. Yet another phrase that is relevant is the notion of pre-analytics. Here, various aggregations might be performed on the data (e.g., summation, histogram construction) the results of which are then used, downstream, during the analytics process. Beyond dealing with record-oriented data, modern data integration must deal with semi-structured and unstructured data as well [48]. Frequently, the challenges here include parsing of the data to extract what structure does exist (e.g., click streams) and text processing to address unstructured data (e.g., blog posts).

While the individual transforms are each (mostly) quite straightforward, the task is quickly complicated by the fact that individual data streams can be quite large and there are frequently many streams, each requiring a distinct transformation specification. Tens to hundreds of multi-Gigabyte data streams must be concurrently integrated, and this must be done prior to actually doing any of the real data analysis, the ultimate goal.

The issue of how to effectively achieve data integration is a pain point for enterprise data, sensor data, scientific data, financial data, etc. Data-driven public policy, economics, and journalism all rely on data from widely disparate sources that must first go through data integration prior to effective use [54]. In short, efficient data integration is crucial to effective use of big data resources.

Data integration manifests itself in many of the preliminary steps that researchers take before applying the analysis algorithms or processing steps to their data. An example is the effort expended in the effective usage of data provided by the US Virtual Astronomical Observatory

(VAO). Here, a team comprised of over 11 institutions provides access to astronomical data from the National Optical Astronomy Observatory (NOAO), the National Radio Astronomy Observatory (NRAO), the Sloan Digital Sky Survey (SDSS), the 2 Micron All Sky Survey (2MASS), the Hubble Space Telescope, the Chandra X-ray Observatory, the Spitzer Space Telescope, and others. Section 5.2 of the VAO Annual Report [131], entitled "Data Sharing and Publishing," describes updates to no less than 7 different tools for accessing the disparate data available, one of which is tasked with simply telling astronomy researchers where they can download the descriptions of the data sets themselves. Another tool provides information about the various data models.

Thus, a user runs a first tool to find the format that describes the data, runs a second tool to access the data model (our user at this point finally has metadata, but no actual data), and then runs yet a different tool to access the data set itself. However, if they wish to use data from more than one source, there is still the need to unify the (differently formatted) data from these sources into a common format for analysis.

As another example, consider the needs of a researcher in biosequence analysis. Genomic and proteomic data sets are available from a wide variety of sources in a large number of disparate formats (e.g., FASTA, FASTQ, SAM, BAM, AB1/SCF, PDB, GTF, etc.). Wikipedia lists 22 distinct file formats for molecular biology and bioinformatics[2]. The data volumes are sufficiently large that simply transforming the data from its original form into that needed for analysis is becoming time prohibitive (e.g., three days are required to perform duplicate marking, base score quality recalibration, and local realignment on a 250 GB BAM file at 30× coverage [91]).

[2] `http://en.wikipedia.org/wiki/List_of_file_formats#Biology`

While there are a number of ways in which one could attempt to organize data integration applications, we will consider an individual data integration job to be decomposed into one or more of the following three tasks:

- Parsing/Cleansing – the computation associated with recognizing the records, fields, and/or other components of the input data, including checking to see if it is well-formed and addressing any example inputs that aren't well-formed.
- Transformation – once parsed, the input data must be translated into the form that is expected by the primary computation, typically going from a file-oriented format to a memory-oriented format.
- Aggregation – any pre-analytics computations that result in aggregate information about the input.

While the boundaries between the tasks in each category above are not always completely clear, we will use the above tasks to help us reason about how representative and comprehensive is the set of applications ultimately chosen to be in the benchmark suite.

Here, we present the Data Integration Benchmark Suite (DIBS), a set of applications spanning several different application domains and the above three types of data integration tasks. DIBS tries to be reasonably comprehensive with respect to both applications and tasks. To help us address how comprehensive they truly are, the benchmarks are characterized through different measures in order to capture the properties (and idiosyncrasies) across the various data integration tasks represented in the suite. The goals of DIBS are to provide insight into the the nature of data integration tasks to guide research in this area, and to create a way in which different research groups can compare their work [82].

3.1 Overview of Benchmark Suite and Integration Tasks

The challenges in selecting candidate applications for any benchmark suite include whether or not the candidates that are ultimately included are both representative of the field and comprehensive in their coverage of the field. To help us assure that the selected applications are representative, we consider each application across two dimensions.

First, we want to capture the breadth of application domains that handle large volumes of data. Scientific data are quite commonly organized in either one-dimensional or a two-dimensional form. As a representative of one-dimensional data integration, we include biosequence data from the field of computational biology. For two-dimensional data, we chose several image processing data transformations. To reflect the importance of business applications (and corresponding data volumes), we include a pair of data integration applications from the enterprise space. We round out the set of application domains by including examples from IoT data and graph processing.

Second, we want to ensure that we include tasks that span the three composite parts of data integration. For parsing and data cleansing, in many cases we are reading a human-readable format and recognizing fields, records, etc., while at the same time separating the primary data set from associated metadata. The data transformations essentially define each specific application, putting the data into the form required by the computation that follows. Finally, the aggregation tasks include counts, summations, and histograms.

The relationship between the five application domains, types of integration tasks, and elements included in the benchmark are all summarized in Table 3.1. The applications themselves (shown in the Transformation column) are each described in the following section.

Table 3.1: Data Integration Task Classification.

	Data Integration Tasks		
Domain	**Parsing/Cleansing**	**Transformation**	**Aggregation**
Computational Biology	Separate bases and meta-data Handle non-A,T,G,C bases	fa→2bit 2-bit→fa	Track total size
Image Processing	Parse FITS tags	fits→tiff idx→tiff optdigits→tiff unipen→tiff	Pixel statistics Histogram Taking log of pixels
Enterprise	Adjust non-ASCII characters	ebcdic→txt fix→float	Count number of elements
Internet of Things	Tokenize input	tstcsv→csv gotrackcsv→csv plt→csv	Running total of file size
Graph Processing	Parse edge list	edgelist→csr	Get total vertex/edge count Compute vertex edge degree

To assess the extent to which the benchmark suite provides comprehensive coverage of the area, we will rely primarily on the distribution of the properties of the applications, described in Section 3.3.

3.2 Benchmark Application Descriptions

In this section, we will describe each of the benchmark applications, identified by their associated data transformation and organized by application domain. In all cases, the data integration applications are written in C and the input data set size is large enough such that any second-order effects caused by start-up transients can be ignored.

3.2.1 Computational Biology

In bioinformatics, DNA sequence alignment is the use of computing to compare sequences of DNA to determine the degree of similarity between them. Based on discovered similarities, researchers can determine structural relationships, pinpoint evolutionary mutations, and predict biological functions [51].

Currently, there are a number of different sequence alignment computing tools and software that, at their core, compare an input sequence against a known genomic sequence. For example, the popular Basic Local Alignment Search Tool (BLAST) from Altschul et al. [3] performs the comparison by approximating the Smith-Waterman dynamic programming algorithm to find the maximal segment pair between the two sequences. Kent et al. [72] developed the BLAST-like Alignment Tool (BLAT), which performs the comparison in a similar to BLAST. It sacrifices homology depth for speed. The inputs to these tools, however, are far from standardized. BLAST accepts sequences using the FASTA format [85], which was born out of an alignment tool that predates BLAST. BLAT uses a custom 2-bit format in which the four DNA bases are represented by two bits per base. There are also a number of other DNA sequence formats that exist, such as FASTQ [29], SAM/BAM [83], and AB1 [126].

Often, it is the case that researchers will want to perform analyses on DNA sequence data that require a format that differs from the format the data is originally in. For our suite, we have chosen two of the conversion utilities available with BLAT–namely, the conversion from the FASTA form to 2-bit form, and vice versa.

fa→2bit

In the FASTA to 2-bit form, the input FASTA bases are parsed line by line. The set of accepted bases is $\{a, A, g, G, c, C, t, T, n, N\}$. For each base in a given line, its 2-bit equivalent is found and packed into bytes, with four, 2-bit bases per byte. Newlines are not recorded. There is also some metadata to be stored alongside the sequence of bases. If one or more consecutive, non-A,T,G,C bases are found, the amount of them and their relative position are recorded. This is also computed for one or more consecutive blocks of lowercase bases. The two-bit data is then stored in a data structure that accounts for the raw data as well as metadata (e.g. name of sequence, size, count and position of non-A,T,G,C bases) associated with the raw data. A sample of FASTA human genome data[3] totaling 130 MB in size is used as input. When looking at results, this transformation will be under the label `fa→2bit`.

2bit→fa

In the reciprocal transformation, 2-bit to FASTA conversion, an input 2-bit file is parsed. Each byte is unpacked into its corresponding FASTA representation. Each character is then converted to its upper-case representation. The metadata from the 2-bit representation is used to restore the lower-case blocks as well as the blocks of the character '`N`'. The 2-bit representation of the input data from the previous transformation is used as input and is 34 MB. This transformation is labeled `2bit→fa`.

3.2.2 Image Processing

From consumers and smartphones, medical physicians and biomedical imaging systems, and astrophysicists and space telescopes, the proliferation of imaging data and has become one of the most voluminous sources of data today. As reported by Venter and Stein [129], images make up more than 80% of all corporate and public unstructured big data. For our benchmarking suite, we have selected four different image processing applications in which a non-traditional image format is converted into a more conventional one.

[3] `ftp://ftp.ensembl.org/pub/release-89/fasta/homo_sapiens/dna/`

fits→tiff

The Flexible Image Transport System (FITS) file format was developed specifically for storing data from scientific applications, and is the most common format for astronomical imaging data. FITS files contain one or more headers that contain metadata such as data types and image dimensions. Raw data immediately follows a metadata header. Our conversion consists of three parts. First, the metadata headers are parsed and written into an accompanying JSON file. Next, the raw data is copied into a buffer that will be written into a `TIFF` file. Finally, descriptive statistics are calculated and a histogram of the image is created. Input data for this transformation is of the globular cluster Messier 12 through the B band recorded by the Hubble Telescope[4], and totals 17 MB. This transformation is labeled `fits→tiff`.

Handwriting recognition is the process of taking a source of handwritten input and converting it into a machine-readable form. There are have been many unique formats developed by researchers to store handwriting data. In our suite, we present a conversion to `TIFF` for three of these formats.

idx→tiff

The first is the `IDX` file format, which is the format used by the MNIST handwriting database. `IDX` is comprised of two files. The raw data, which is stored in the first file (`.idx3-ubyte`), contains a compilation of handwriting samples. The offsets for each image's pixel data are computed using the meta-data encoded at the beginning of the `.idx3-ubyte` file. This metadata includes the total number of images in the file and the number of rows and columns

[4] `https://www.spacetelescope.org/projects/fits_liberator/m12data/`

in each image. The labels (e.g., image 701 is the number "4") associated with each image is stored in the second file (`.idx1-ubyte`), and can be indexed in a similar fashion as the image data. Both files are read into memory, and a different `TIFF` file is generated for each handwriting sample. The input to our transformation is 7.5 MB and is taken from the MNIST website[5]. This transformation is labeled `idx→tiff`.

optdigits→tiff

The `optdigits` format represents a set of handwritten digits in a bitmap format encoded in `ASCII`. The raw handwriting data is preceded by metadata that describes the uniform height and width of each digit, as well as the total number of digits in the set. This format is used in the Optical Recognition of Handwritten Digits dataset [84], which is 2.0 MB and serves as the input for our transformation. The transformation is performed by parsing each handwritten digit separately, converting strings of '1's and '0's into 8-bit pixel values to create a `TIFF` image. This transformation is labeled `optdigits→tiff`.

unipen→tiff

The `UNIPEN` format consists of handwriting digits that are described as a set of XY coordinates. Sets of coordinates are demarcated by keywords that denote pen strokes (i.e., it is in vector form). The transformation parses one set of coordinates associated with a given handwriting sample at a time. To construct a `TIFF` image from a set of coordinates, each consecutive pair of coordinates is treated as a line segment. Circles of a specified radius are drawn from the first coordinate to the second using the Midpoint Circle Drawing algorithm [44] and are then filled in after each circle is drawn. We use the 1.6 MB Pen-Based

[5] `http://yann.lecun.com/exdb/mnist/`

Recognition of Handwritten Digits dataset [84] as input. This transformation is labeled `unipen→tiff`.

3.2.3 Enterprise

Enterprise data transformations are classically considered to be part of ETL (Extract, Transform, and Load) processing. Poess et al. [106] include 18 different transformations in their benchmark suite, which includes the same tasks that we use for organization of the transformation (parse/cleanse, transform, aggregate). One thing we do not include, which is present in [106], is a join operation across two distinct inputs. A great many of enterprise transformations are motivated by businesses moving away from mainframe execution to using cloud-based machines. As such, data type transformations are quite prevalent, especially from older, legacy systems (often EBCDIC based and even not including floating-point hardware) to modern x86 platforms.

ebcdic→txt

In our benchmark suite, we include a simple `EBCDIC` to `ASCII` transformation. Our transformation uses the traditional 7-bit `ASCII` encodings. Once the total number of elements is calculated, the conversion executes by referencing a look-up table to find the corresponding `EBCDIC` character. If a particular `EBCDIC` character does not have a corresponding `ASCII` equivalent, the offending `EBCDIC` encoding is assigned an unused `ASCII` encoding specific to that `EBCDIC` encoding. A 9.2 MB `EBCDIC` file is used as input. This transformation is labeled `ebcdic→txt`.

fix→float

Additionally, we include a conversion for fixed-point data to floating point. The input dataset is a 10 MB, random binary file[6]. The input data is interpreted in 16-bit chunks with a user defined number of fractional bits. Each number is then converted into a 32-bit floating point number using a bit-shift, division, and typecasts. The value is then saved into a memory buffer completing the transformation. This transformation is labeled `fix→float`.

3.2.4 Internet of Things (IoT)

The Internet of Things (IoT) represents scores of devices with newly enabled connectivity [45]. While the promise of increased functionality provided by these devices is high, there is very limited commonality in how they provide the data that they all collect. For example, we use GPS data from multiple sources [32, 97, 146], yet different transformations are required for each input data set. In our benchmark suite, we transform three different types of GPS data in order to normalize them. The normalized format is a `CSV` file where each record takes the form `<ID>,<Latitude>,<Longitude>`.

tstcsv→csv

The `CSV` format used for the Taxi Service Trajectory (TST) Prediction Challenge of 2015[7] uses a `CSV` format containing data describing the trajectories of operating taxis in Porto, Portugal. The GPS trajectories are located in the last field of a data line, and are stored as a list of coordinates in the form `<Longitude>,<Latitude>`. Each list is record of the

[6] `http://rngresearch.com`

[7] `http://www.geolink.pt/ecmlpkdd2015-challenge/index.html`

periodically recorded GPS coordinates from start to end of a fare. In our transformation, we store the unique identifier for the trip, navigate to and parse the trajectories list for that trip, swap the order of each pair of coordinates to match the normalized form, keep a running total of the size, and create a new `CSV` file using the normalized format. A 437 KB data set from the TST challenge [84] is used as input. This transformation is labeled `tstcsv`→`csv`.

gotrackcsv→csv

The GPS data released by the GoTrack mobile application[8] is packaged in a `CSV` format that contains GPS coordinates, coordinate identifiers, and timestamps. In our conversion, we parse the input data line by line for the coordinate identifier and GPS coordinates. Then, we write the data using the normalized format to a new `CSV` file. We use a 1.1 MB set of GPS data from the GoTrack mobile application [84] as input. This transformation is labeled `gotrackcsv`→`csv`.

plt→csv

The `PLT` format is used in the GeoLife project[9] conducted by Microsoft Research Asia to store GPS trajectories. The `PLT` format is essentially in `CSV` format, except the raw data is preceded by six lines of metadata. The raw data includes GPS coordinates, altitude, and variations of timing data. In our transformation, we calculate the total number of coordinate pairs, parse the input data line by line, extract the coordinate identifier and its corresponding `<Longitude>,<Latitude>` pair, and write the data using the normalized format into a new `CSV` file. We use a 449 KB subset of the GeoLife dataset as input. This transformation is labeled `plt`→`csv`.

[8] `https://play.google.com/store/apps/details?id=com.numerex&hl=en`

[9] `https://www.microsoft.com/en-us/download/details.aspx?id=52367`

3.2.5 Graph Processing

Across many disciplines, mapping complex problems to graph structures is a common occurrence [116]. Mapping problems to graph structures is advantageous because there exists a wealth of graph theory information and research that can then be applied to the problem. Graph processing has become so important that there exists an intensive graphing application benchmark, known as the Graph500, that is used to compare the performance of world's supercomputers. One piece of the benchmark, Kernel 1, consists of a kernel in which an edge list is converted into a graph. In our benchmark suite, we have included the Graph500 reference implementation of this kernel[10].

edgelist→csr

Kernel 1 takes an edge list and creates a graph using the Compressed Sparse Row (CSR) [111] matrix representation. This is accomplished by first calculating the degree of each vertex and associating edges with their respective vertices. This information is used to create three arrays to fill buffers that describe the graph's non-zero elements, the number of non-zero elements in each row, and the column position of each non-zero element. The input data size is generated during run time, and is approximately 280 MB in size. This transformation is labeled `edgelist→csr`.

[10] `http://graph500.org/?page_id=47`

3.3 Characterization of Data Integration Tasks

In determining what attributes to choose to characterize our benchmark suite, we want to address two specific things. The first is choosing an analysis that enables a comprehensive look at the benchmark suite through many characteristic dimensions. Thus, the attributes we use to characterize the data integration tasks in DIBS are chosen to exhibit the overall behavior of each task and capture any idiosyncrasies associated with them. In most cases, data integration tasks take the shape of looping through each data element in a set and performing the required integration task. There are some qualitative properties that are possibly intuited from looking at data integration tasks in this form, and our characterization works to quantify these intuitions. Additionally, there are things that may not be so intuitive, and our characterization selection also addresses such properties. In our characterization, we declare data ingestion beyond the scope of our analysis, and focus instead on memory access and compute behavior of the tasks. All of the characterization methods that follow are profiling the data integration tasks themselves, and not the execution before and after the data integration tasks.

Second, we wish to craft an analysis that is independent of the system that our suite is deployed on. Current systems are comprised of many differing components and features—for example, differing instruction paradigms like RISC and CISC, hardware accelerators, and distributed systems—that are tasked with handling the load of data integration. This necessitates that we create a characterization that is independent of the test system so that our analysis can focus on the applications themselves and not the idiosyncrasies of the execution platform.

With the aforementioned objectives, along with characterizations from benchmarking suites listed in Section 2.2 as guides, we have chosen the following characterizations.

3.3.1 Locality

Measures of locality allow us to examine the behavior of a program's memory access patterns. To this end, we present measures for spatial and temporal locality. Our interest here is limited to data access patterns, leaving instructions to be addressed below.

Spatial Locality

Qualitatively, a program's spatial locality is described by whether or not subsequent memory references will be located near previous memory accesses [15]. Programs in which future memory references are near previous memory locations references are said to exhibit high spatial locality. Higher spatial locality is generally beneficial to programs because it allows contiguous chunks of data to exist in caches with less thrashing and evictions.

From a quantitative perspective, we need a method to express the degree of a program's spatial locality. In our characterization, we draw from work done by Weinberg et. al [136] to quantify spatial locality in an architecturally independent manner. In this metric, used also by Reagen et al.[109], they describe the stride of a memory access as the difference between two memory reference addresses in units of a 64-bit word size. They present the following equation to quantify spatial locality:

$$L_{Spatial} = \sum_{i=1}^{\infty} \frac{stride_i}{i} \tag{3.1}$$

where $stride_i$ is the total number of memory accesses that are of stride length i. The result of this expression is a normalized score in the range [0,1] that can be used to compare the spatial locality between programs.

Temporal Locality

Temporal locality is a characteristic of a program's memory access pattern that describes the frequency of memory accesses to the same memory location. Higher temporally local programs reference the same memory locations numerous times, whereas lower temporally local programs do not exhibit as much data reuse. Programs with higher temporal locality have similar behavior to programs with higher spatial locality relative to memory and cache behavior, thus they are afforded similar benefits at that level.

To quantify temporal locality, we again draw from work done by Weinberg et al. [136]. They describe temporal locality through data reuse. Given a particular memory address, data reuse is the number of unique memory addresses that have been accessed before that particular memory address is referenced again. The formula that they proposed to quantify temporal locality is shown below:

$$L_{Temporal} = \frac{\sum_{i=0}^{\log_2(N)-1} [\ (reuse_{2^{i+1}} - reuse_{2^i})\ \times\ (\log_2(N) - i)\ ]}{\log_2 N} \tag{3.2}$$

where $reuse_{2^i}$ is the number of dynamic memory accesses with reuse distance less than or equal to 2^i and N is the largest reuse distance used. This metric also produces a score within the range [0,1] with which to compare to the temporal locality scores of programs.

3.3.2 Determinism/Branch Entropy

The predictability of a program's control flow can be characterized by the regularity of the program's branching behavior during execution. Regularity in a program's control can dictate the performance of a program on an underlying architecture. Strong regularity in control behavior allows for more confident branch predictions, while irregular branching decreases prediction confidence. To quantify a program's branching behavior, we draw from work done by Yokota et al. [141]. Inspired by Claude Shannon and information theory, they define a measure called branch entropy, which quantifies program regularity through branching behavior. Specifically, we will be using their formulation for table reference entropy, based on the values that a pattern history register assumes. The pattern history register acts as a shift register that either shifts in a 1 or 0, for a branch that is taken or not taken, respectively. In this case, we will make the shift register 16 bits in length. A table reference entry, then, is a resulting 16-bit value that the pattern history register takes after it is updated by a branching decision. The formula for the branch entropy metric is shown below:

$$BE = -\sum_{i} p(E_i) \log_2 p(E_i) \tag{3.3}$$

where E_i is the i-th entry of the table, and $p(E_i)$ is the probability of E_i occurring.

3.3.3 Instruction Mix

The instruction mix of a program is a measure of the unique instruction classes the program contains and the distribution of those instructions during execution. In our benchmark suite, we examine both the static and dynamic instruction mix, and we classify instructions in three categories: compute, control flow/branch, and data movement.

Static Instruction Mix

The static instruction mix of a program is a static count of the unique machine instructions present in the program image, post compilation. This metric shows the ratios of differing classes of instructions which give insight into what operations are necessary for program execution.

Dynamic Instruction Mix

The dynamic instruction mix of a program is the count of how many times the aforementioned classes of machine instructions were executed. This metric can reveal hotspots of a program's execution and characterize the execution-time leanings of the program, for example, mostly compute or an equal combination of data movement and compute.

3.4 Characterization Methods

In this work, we are only considering program execution on a single core and that the working set size fits in the experimental machine's RAM. Though many of these benchmarks can be extended to deployment on multiple cores, hardware accelerators, heterogeneous systems, or multiple nodes, we leave this extension to future work. The specifications of the experimental machine are listed in Table 3.2.

Table 3.2: Experimental machine specifications.

CPU	Intel Core i7 930
Clock rate	2.8 GHz
L1 d-cache size	32 KB
L1 i-cache size	32 KB
L2 cache size	256 KB
L3 cache size	8192 KB
RAM size	24 GB
Compiler	GCC 4.8.4
ISA	x86-64

To measure locality, branch entropy, and instruction mix, we use Pin, which is a dynamic binary instrumentation framework for IA-32, x86-64, and MIC instruction-set architectures that allows us to dynamically instrument our applications [87]. Thus, instrumentation is performed at run time on the compiled binary files, which captures the behavior of the applications as they are executed. The Pin framework allows us to perform architecturally independent analyses of each program in our benchmark suite.

It is apparent, however, that instruction mix is inherently architecturally dependent. We address this issue in several ways. First, we compile all programs with the default level of optimization in GCC (`-O0`) in order to prevent the exploitation of architecturally specific features at compile time. Second, we use a coarse categorization of instruction types, as detailed in Section 3.3.3, to abstract away the particular details of differing architectures. Since x86-64 is a CISC instruction set, some instructions do have implicit data movement within an instruction. In these cases, we categorize the instruction based on its main function, i.e. an `ADD` instruction counts as a compute instruction. Third, for a subset of the applications, we provide instruction mix data for the ARM AArch64 instruction set in addition to the x86-64 instruction set. Since the AArch64 instruction set is not supported by the Pin framework, we use the gem5 simulation environment [11] and cross-compile our applications to make this characterization.

Table 3.3: Throughput results.

Application	Throughput (MB/s)
fa→2bit	23.4
2-bit→fa	12.2
fits→tiff	43.8
idx→tiff	13.2
optdigits→tiff	133
unipen→tiff	0.113
ebcdic→txt	139
fix→float	204
tstcsv→csv	75.8
gotrackcsv→csv	104
plt→csv	123
edgelist→csr	3310

We present throughput rates ($\frac{input\ data\ size}{execution\ time}$) in Table 3.3 for our benchmark applications executing on a single core of the machine described in Table 3.2. In each case, the application was run 100 times and the value represented is the average across the runs.

While data integration tasks can be limited by I/O, we are interested in their computational limitations. As can be observed in the data of Table 3.3, these data rates are below that achievable in a modern I/O subsystem, therefore warranting investigation of their computational performance properties.

3.5 Results of Characterization

In this section, we present the findings of applying our characterizations to the applications of the benchmark suite.

3.5.1 Locality

As mentioned in Section 3.3, the structure of most of our applications is a sequential loop over all of the data records of a given input and performing the integration task. Since most data records are located next to each other in memory, we expect that most programs exhibit high spatial locality. Furthermore, the successfully processed data record is no longer needed. Thus, we expect that the amount of data reuse and temporal locality in such data integration tasks to be low.

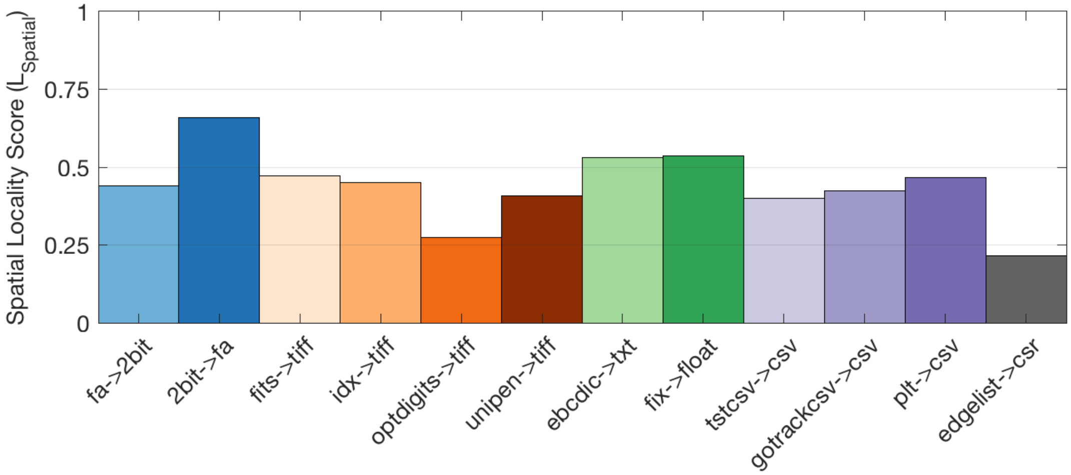


Figure 3.1: Spatial locality measure.

Spatial Locality

The results of the spatial locality characterization are shown in Figure 3.1. Although the spatial locality scores of the data integration applications are not as high as we originally posited, the level of spatial locality is consistent across applications. To better understand this, we can decompose the cumulative sum used to calculate $L_{Spatial}$ for each application as shown in Figure 3.2. In this figure, we separate the applications into the domains from which they originated, so that any trends that are domain-specific can be readily identified.

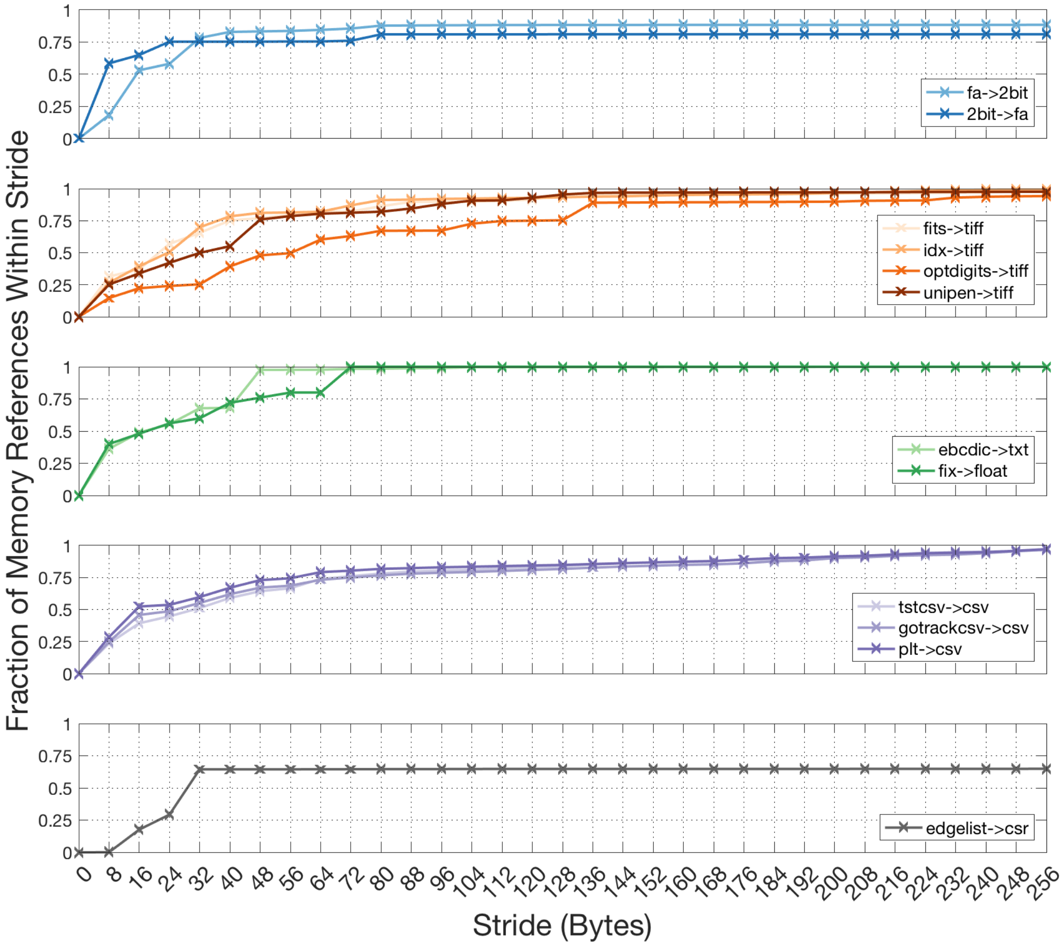


Figure 3.2: Cumulative sum of memory references across strides.

From the figure, we observe that 75% of memory references occur within a stride of 80 bytes for 10 of the 12 data integration applications, independent of their domain.

The `2bit→fa` application reaches 75% at a stride of 16 bytes. This results in a much higher spatial locality score than the other data integration tasks. However, the `optdigits→tiff` and `edgelist→csr` applications exhibit much lower spatial locality than the other applications. We examine the outlying applications to explain why this is the case.

Recall that smaller strides between subsequent memory accesses yield higher spatial locality scores. The 2bit→fa transformation has the highest locality score because in this transformation, 32 bases are packed into a single, 8-byte word. Since the data are packed so tightly, sequential accesses in the 2bit→fa transformation experience high spatial locality. Looking at Figure 3.2, we observe the result of the tightly packed data through the fact that over 50% of memory references are within 8 bytes of each other. In the edgelist→csr application, creating the sparse representation of the edge list is still sequential in nature, but the base stride distance is longer, as seen in Figure 3.2. In the optdigits→tiff case, each image is parsed and written line by line, effectively increasing the stride distance of the data integration task's memory accesses.

Temporal Locality

The results of the temporal locality characterization are shown in Figure 3.3. Similar to the spatial locality scores, the accompanying temporal locality scores of the benchmark suite are not as low as originally posited, but are relatively consistent relative to each other. Eight of the 12 data integration applications have higher spatial locality than temporal locality. This observation is the most pronounced in 2bit→fa and the two Enterprise integration tasks. All three of these applications are marked by memory references of small stride distances with minimal data reuse, which aligns with our original first principles intuition. Decomposing the temporal locality scores in Figure 3.4, we observe that the 2-bit→fa and the two enterprise integration tasks show minimal data reuse at smaller reuse distances. This explains the lower temporal locality scores since our metric favors earlier data reuse.

Of the four applications for which temporal locality is higher than spatial locality, the optdigits→tiff, the tstcsv→csv and the gotrackcsv→csv transformations are observed to have a similar levels of spatial locality as temporal locality. However, the difference is the most pronounced in the `edgelist→csr` applications. The main reason is that each edge of the edgelist is stored in a structure that packs inside it the two vertices that the edge connects. Counting the number of vertices and the degree of each vertex both take the form of iterating over a group of edges. In each of these integration tasks, each edge structure must be reference and unpacked, and extracting both vertices from each structure accounts for the higher temporal locality exhibited by the `edgelist→csr` application relative to its level of spatial locality.

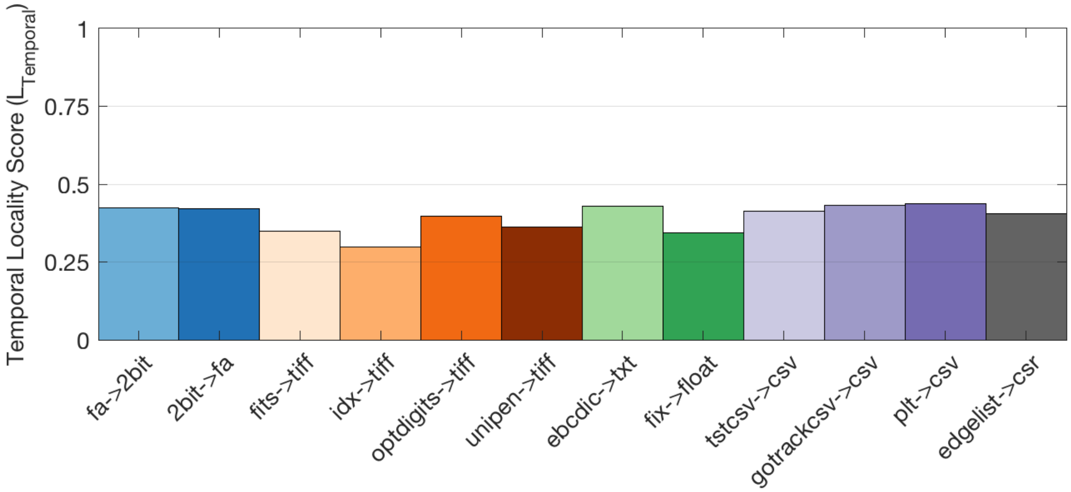


Figure 3.3: Temporal locality measure.

3.5.2 Branch Entropy

Figure 3.5 shows the result of the branch entropy characterization of our benchmarking suite. Looking at the distribution of the branch entropies of our benchmark suite, we observe that there is a wide variety of branch entropy results, which speaks to the variability of each application with respect to control flow. The `unipen→tiff` and `tstcsv→csv` transformations exhibit the highest branch entropy. In the `unipen→tiff` case, the branch entropy stems

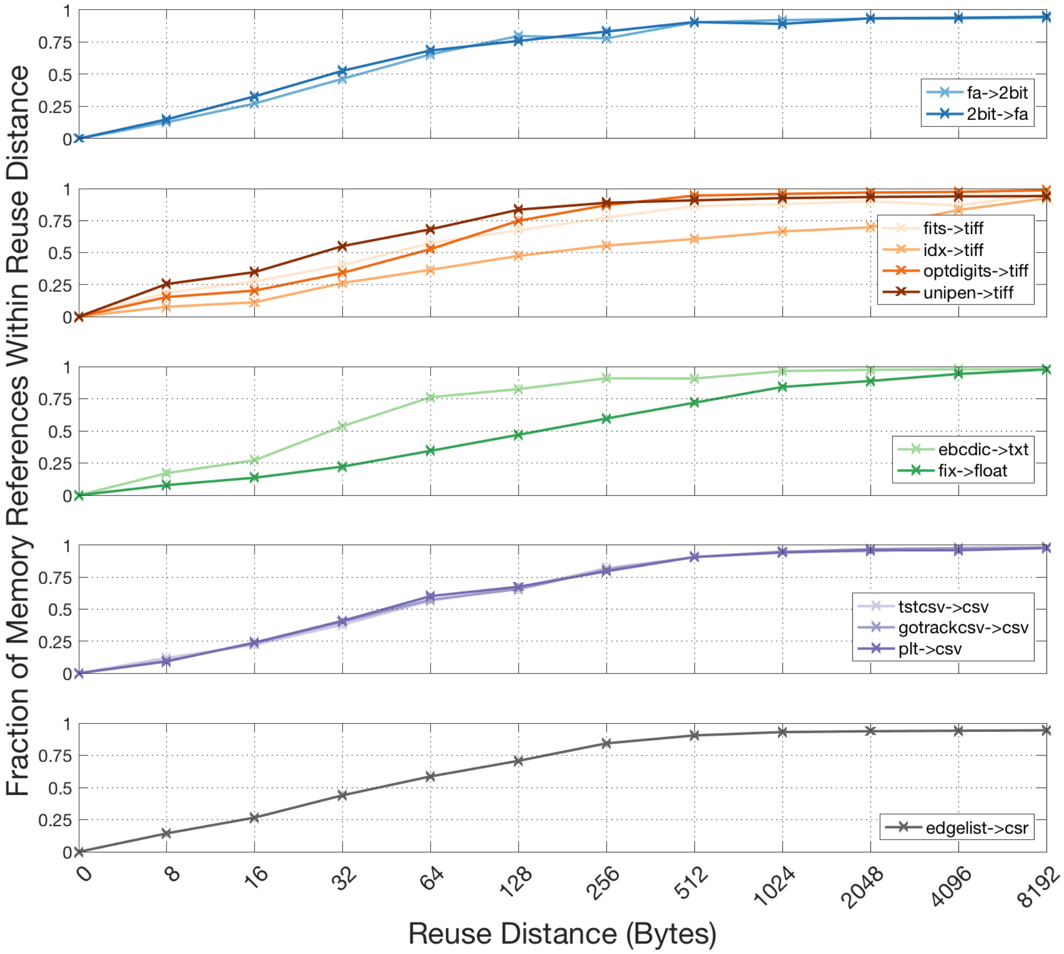


Figure 3.4: Cumulative sum of memory references across reuse distances.

from each pair of input coordinates and their relationship to each other yielding different ways of redrawing the digit in the `TIFF` format. In the `tstcsv`→`csv`, the number of coordinates varies in each list, which results in a variety of possible entries in the pattern history register. The two enterprise integration tasks, `ebcdic`→`txt` and `fix`→`float`, exhibit the lowest branch entropy. In both cases, the only branching decision to make is terminating the loop if all of the data records have been processed. This corresponds to the pattern history register value that represents always taking the branch (all 1's) occurring at almost every branching decision.

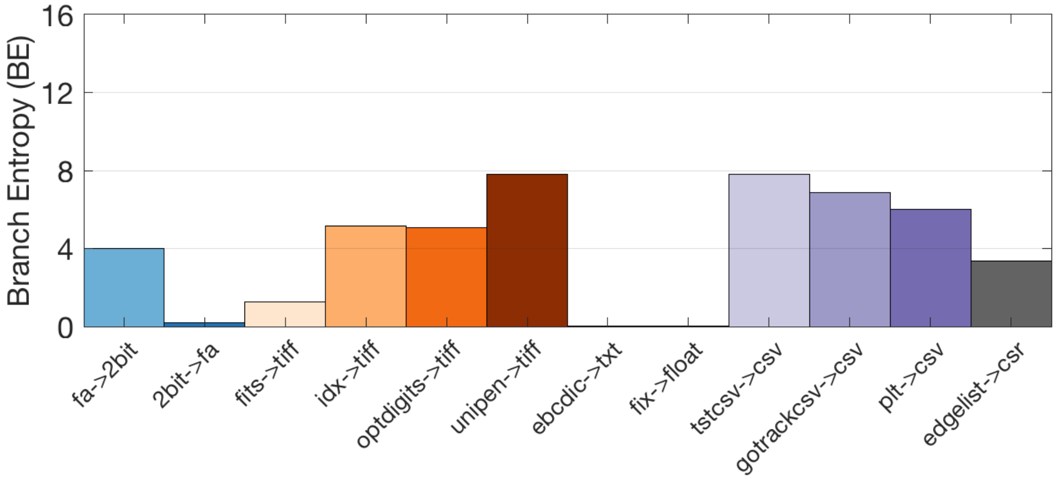


Figure 3.5: Branch entropy measure.

Although there exists variability among the integration tasks in our suite, all of them can be classified as executing with a certain level of control flow regularity. To contextualize this observation, we see that the maximum value of branch entropy metric is 16 bits, while no applications in the benchmark suite exceed 8 bits. To achieve the maximum value of branch entropy, there must be an equal probability of occurrence of all possible entries that the pattern history register can take. This equal probability results in the highest degree of irregular application control flow because all branching decisions are equally likely. This is not the case in our application suite because at their core, all integration tasks take the form of iterating over a set of data records. In general, this form exhibits relatively high control flow regularity.

3.5.3 Instruction Mix

x86-64 ISA

Figures 3.6 and 3.7 show the results of the static and dynamic instruction mix characterization, respectively, on the benchmark suite. The static instruction mix characterization shows the prominence of unique data movement instructions in each integration task. Out of 12 tasks, 10 of them are comprised of more than 50% unique data movement instructions. This follows from the fact that our benchmark suite is based on data-related operations. This line of reasoning also explains the low percentage of unique compute and control flow instructions in each benchmark since most of the data transformations are relatively simple.

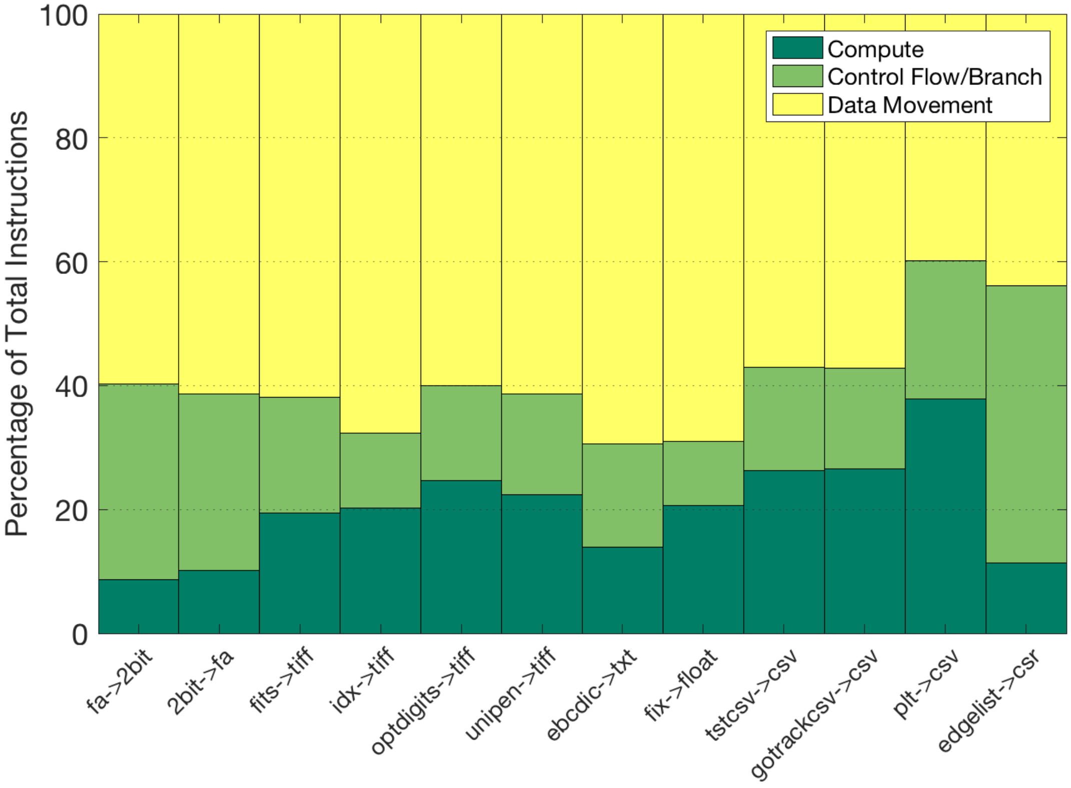


Figure 3.6: x86-64 static instruction mix.

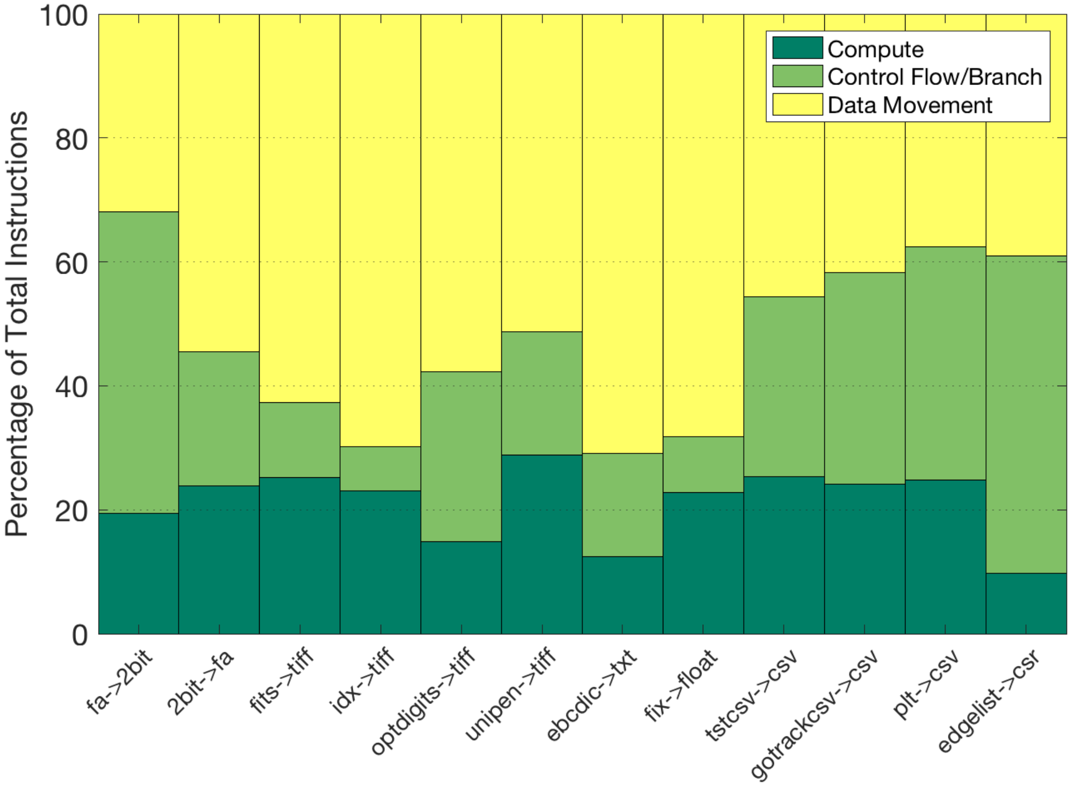


Figure 3.7: x86-64 dynamic instruction mix.

In the dynamic instruction mix characterization, we observe that data movement still exhibits a significant presence during execution time in most of the applications, noting that this presence is even larger but not represented since we binned complex instructions that contain implicit data movement as their main function only. We also observe that the presence of control flow and branching instructions during execution time becomes salient. There is at least one branching condition (the terminating condition) executed every time a data record is processed. Additionally, there are branching decisions associated with the way in which each record is transformed. Finally, the instruction mix characteristics vary fairly significantly across the benchmark suite; one indication that the choice of applications is reasonably comprehensive.

ARM ISA

Figure 3.8 shows the dynamic instruction mix characterization for 7 of the 12 benchmark applications when compiled to the ARM instruction set. There are a few observations worth making here. First, the results are distinct from those reported in Figure 3.7, confirming our earlier observation that the instruction mix characterization is inherently not architecture independent (i.e, it is clearly architecture dependent). Second, the fraction of data movement instructions is noticeably smaller (especially for a few of the applications), which could easily be due to the distinction between the RISC and CISC instruction set styles. Finally, we continue to see significant variation across the suite, further indication that our applications are reasonably comprehensive.

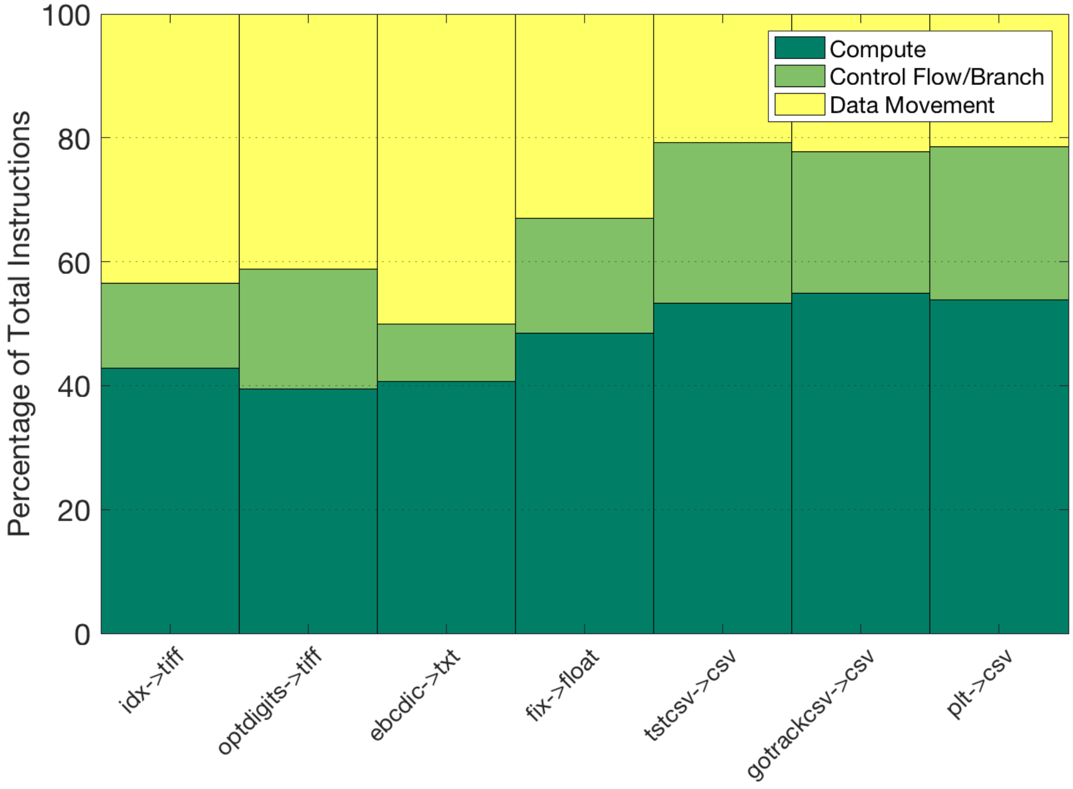


Figure 3.8: AArch64 dynamic instruction mix.

3.5.4 Discussion

For most of the characterizations considered, our first principles arguments held. Though the spatial locality was not as high as originally intuited, the degree of spatial locality was consistent among applications, which supports the idea that most data integration applications have a certain degree of spatial locality. Temporal locality was lower than spatial locality for most of the integration tasks. Additionally, the applications assumed a consistent level of temporal locality. The applications in which they did not fully hold allows this characterization to capture the idiosyncrasies associated with such data integration tasks with regard to locality, which adds to the insight gathered by the characterization and adds to its comprehensiveness. Modern memory subsystems are designed to exploit applications with high data reuse, and while we have shown that there is a consistent level of locality, future solutions addressing the data integration problem could be tailored to exploit the specific level of locality present in the data integration tasks.

The treatment of branch entropy on the benchmark suite revealed that there is indeed variability among the different applications in that regard, but that data integration tasks as a whole generally exhibit a high level of control flow regularity. This result has implications for how complex the branch predicting algorithms and hardware of a particular system needs to be for data integration tasks.

Examining both the dynamic instruction mixes of the applications for the x86-64 and AArch64 architectures, we show that the there is a non-trivial variation in the mix. This necessitates a review of performance when porting and compiling data integration codes across different architectures because optimizations for one architecture may not exhibit the same gains on another architecture. Through examining static and dynamic instruction mixes of x86-64, we show that there is a significant amount of data movement instructions in all applications for the x86-64 instruction set. Since the energy required to move data across the memory hierarchy is proportional to performing multiple double-precision floating point operations, this presents an opportunity to investigate ways to optimize execution to address this, such as minimizing the amount of data movement or rearranging data to take account of the existing memory subsystem that the data integration task is deployed on.

3.6 Conclusion

The Data Integration Benchmark Suite (DIBS) is a set of data integration applications that is representative of many different disciplines and integration tasks. We explore the general qualities as well as idiosyncrasies of the suite by applying a comprehensive and (mostly) architecturally-independent set of characterizations to each application. Based on the characterization, we observe that most data integration tasks have a consistent level of both spatial and temporal locality, and that they usually exhibit a higher degree of spatial locality. Our applications are also characterized by a high level of control flow regularity and, in their x86-64 versions, an emphasis on data movement. The insight gained from our characterizations will guide both software and hardware research in exploring and exploiting the qualities associated with data integration tasks. From the results of all of the characterizations, we have satisfied the objectives of creating a comprehensive characterization of the applications through a battery of different metrics.

Regarding domain specific computing, this benchmarking suite has, in part, addressed our intial question of identifying a domain. Specifically, it addresses how to qualitatively identify the domain. Towards this end, we crafted a specific definition of our domain of interest, i.e., data integration, and then searched the literature for applications that fit our definition, as well as creating our own applications that aligned with our definition. From there, we used characterization methods from the literature that allowed us to generate qualitative insights that will inform what a hardware accelerator would look like for this particular domain. In the next chapter, we present a novel locality characterization tool that enables us to build on this qualitative representation and lay the groundwork for a quantitative characterization of our domain. From this quantitative description, we can use the results from this tool to make data-driven hardware design choices.

Finally, we have made these applications and datasets publicly available so that researchers can compare data integration-specific solutions and systems [21].

Chapter 4

Multi-spectral Reuse Distance: Divining Spatial Information from Temporal Data

Equipped with insights regarding the consistency in locality, predictable branching behavior, and instruction mixes defined by a prevalence of data movement instructions, we now have a better understanding of what features would be beneficial in hardware accelerated versions of these applications. Specifically, we want to design hardware that focuses on the data; we want to exploit the locality of the data integration and make data movement efficient. However, these insights, though based off of quantitative measures, are qualitative in nature. Our aim in designing domain specific computing solutions is to use quantitative measures to inform quantitative decisions. The aim of this chapter, then, is to enable quantitative hardware design choices through the development of a novel locality tool to capture data movement measures. In general, this tool is applicable to quantitatively access the locality of any application.

4.1 The Data Movement Problem

At present, data movement is far more expensive than compute (i.e., an off-chip DRAM access will use 1000× more energy, comparatively, than the 64-bit floating-point multiply-add that results from it when calculated using a 28nm process node [33, 73]). It follows that superfluous data movement should be reduced as much as possible as a means to improve system efficiency. Efficiently modeling the spatial and temporal locality of data has a direct impact on multiple facets of the data movement problem [76]. This includes optimal page sizing, data to memory technology placement, data page prefetching (related to placement) [103, 130], and when (and where) to use various forms of data gather/scatter.

Page sizing is often not associated with changes in data movement, though it should be. Whether using a disk controller or the main central processing unit, when data is paged-out and new data paged-in, all the contents of that page must be written to persistent storage if modified. That write-back and subsequent reloading with a new 4KiB page requires 128-256b coherence bus transactions for just one direction of movement (e.g., controller to physical DRAM). If that page isn't fully utilized once it is moved, then much of that data movement is likely wasted. Consider the case when a 2MiB page is loaded to DRAM but only half of the page is used before that physical memory is needed for another application. We will potentially have wasted 2^{15} bus transactions for loading the page, and another 2^{15} transactions (only considering wastage for the portion of the page that was not used, the full page would take 2^{16} bus transactions with a 256b bus). Even when the core is not actively participating in the transfer, cache line tag RAMs will be accessed, as will snoop/directory filters within the cache coherence network. Every access for superfluous data movement is an access taken away from useful data movement. Choosing the correct size of page is also important for copy-on-write memory systems (which most modern operating systems implement). If super (huge) pages are chosen where page utilization is low, much additional data must be copied. For example, any time a write to a child page (the virtual page pointing to a parent original page) occurs, the entire contents of that page must be copied. Simply choosing a smaller page would have been desirable. The model described in this dissertation could be used for online prediction for page size based on actual spatial/temporal reuse patterns, potentially with relatively low overhead.

Modern computer systems often integrate multiple memory technologies into a computer system. As an example, some GPGPU devices incorporate static random-access memory (SRAM), high bandwidth memory (HBM), and nonvolatile memory (NVM) all within the same device, and often byte addressable. The decision on where to place data within this physical memory space has direct system performance implications. Placing data on an HBM device provides very high bandwidth but intermediate latency, whereas placing data in an SRAM scratchpad could provide very low latency and high bandwidth at the expense of lower capacities (relative to other options such as NVM). Current industry practice for placing data on these memories is either to do it manually (user driven) or to treat the memory as a cache with some suitable replacement policy. The model described in this dissertation could be used to determine dynamically what granularity page should be used and if a predictor would be effective. Our model could do this by simplifying complex patterns, which is a side effect of the multi-spectral reuse distance approach (i.e., patterns often are easier to determine at a larger granularity versus small). Evidence presented within this work suggests that by using larger pages, the page placement prediction policy would be easier to derive due to the coarser granularity. Our model could be used as a means to decide between a caching policy or a prediction counter policy that would attempt to proactively fetch the next page.

Tightly related to data and memory technology placement is the choice of where to gather or scatter (and also compress and decompress) data. Currently the best way to decide is through extensive offline profiling on the target system. Evidence suggests that future systems will be equipped with DMA-like gather/scatter engines at multiple locations within the memory hierarchy [7, 86]. Just like the data placement decision and page sizing decisions previously mentioned, gathering data at the network interface controller (NIC) or NVM versus bringing all the data into the coherence network can pay dividends for efficiency [49]. If a system is equipped with multiple gather/scatter units, how is the system to choose between gathering at one location or another? If a reorganization function exists (provided by either the user or compiler), then using the spatial and temporal locality data provided through our described model a system could decide based on a heuristic if less data movement and tighter spatial locality could be gleaned from data reorganization.

This chapter makes two primary contributions. First we demonstrate how to use a well known statistical technique (Earth Mover's Distance, EMD) in a novel way to inform the relationship between spatial and temporal locality. Second, we show empirically the application of our method using a set of industry standard benchmarks as a proof of concept and how multi-spectral reuse distance analysis can inform various facets of memory management. In Chapter 6, we use this technique to guide the design of data integration applications on the Intel HARPv2.

4.2 Methods

This section outlines preliminary information and methodology for measuring multi-spectral reuse distance, using EMD, and how we calculate memory footprint.

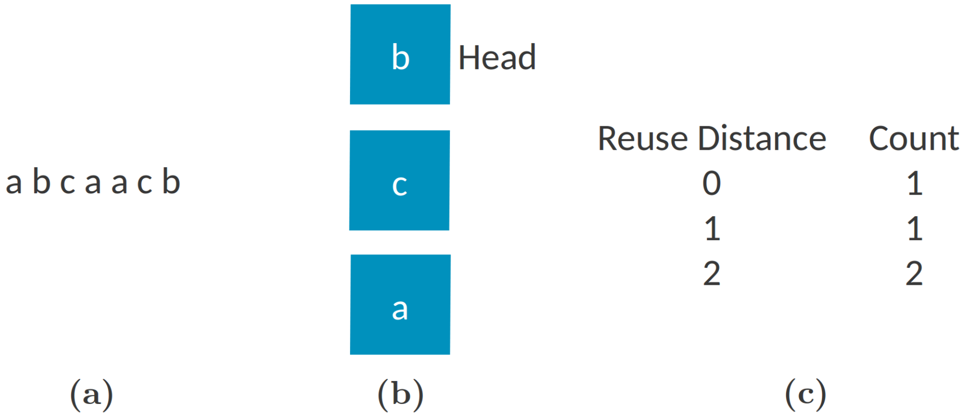
4.2.1 Benchmark Applications

The applications used in this work are a subset of the SPEC2006 benchmark suite [57]. However, profiling the entirety of a given benchmark proved too prohibitive. Generating reuse data for any application compiled with the *-size=train* option (i.e., the largest input size option) took several hours in the worst case. In the case of the *433.milc* benchmark compiled with the *-size=ref* option, the instrumented application took 26 days to complete. Thus, functions within this subset that have been shown to take a large share of the total execution time [104] were characterized. Additionally, the *MEGA-STREAM* benchmark [35], is used to demonstrate behavior of codes with very high memory access to computation ratios (itself derived from stencil computations).

Trying to save the traces of instrumented functions of the applications for post-processing also proved to be problematic because traces easily exceeded terabytes in size. Sampling reuse distances was also a possibility, but we did not want to risk aliasing a reuse distance pattern or miss unique cache line accesses. Thus, the characterization has been limited to 1 trillion references while the target function was executing.

4.2.2 Reuse Distance

In a trace of memory references, given a unique distance is the number of unique references that are made before it is referenced again. Traditionally, memory references take on a cache line (64B) granularity. To calculate reuse distances for an application, a stack is employed to maintain ordering of the memory references as they are encountered. The most recently used memory reference is always at the head of the stack. There are two main operations of the reuse distance stack: encountering either new or previously seen memory references.



Reuse Distance	Count
0	1
1	1
2	2

Figure 4.1: (a) Reference trace. (b) Reuse distance stack. (c) Reuse distance histogram.

A memory reference is added to the stack if it has not been seen during execution. When a memory reference has been encountered before, its index in the stack is isolated and the distance between its index and the head of the stack becomes the reuse distance. This reuse distance is the index into a histogram that keeps track of how many elements have a particular reuse distance. Reuse distance analysis was performed by dynamically instrumenting loads and stores using the *drcachesim* tool of *DynamoRIO* [14].

An example of calculating reuse distance is shown in Figure 4.1. The end result of the reuse distance analysis, i.e., the reuse distance stack in Figure 4.1(b) and histogram in Figure 4.1(c), is shown after processing the reference trace in Figure 4.1(a). Exploring the memory reference named a, it contributes to the reuse distance histogram as follows: the first time it is seen, it is added to the stack. The second time it is encountered, its reuse distance is calculated to be 2, and the reuse distance is 0 when it seen for the third time.

A reuse distance signature is the probability mass function (PMF) for the reuse distances of a given application across a range of bins. In this work, the bins represent groupings of reuse distances on a logarithmic scale.

Reuse distance analysis has traditionally been performed at cache line granularities, i.e., data blocks are set to 64B. However, our particular method uses *multi-spectral* reuse distance, which is to say that we sample reuse distance at 64B, 4KiB, and 2MiB. The 'multi-spectral' character of our methodology is what enables us to yield additional spatial locality information.

4.2.3 Earth Mover's Distance

Earth Mover's Distance (EMD) is a metric described by Rubner et al. [110] that quantifies the similarity of two histograms by finding the minimum amount of work necessary to transform the mass of one histogram into the other. In keeping with the spirit of the nomenclature, the two histograms can intuitively be viewed as a supplier and consumer of dirt (mass) that make up the two disjoint sets of a complete bipartite graph with weighted edges. The nodes of the supplier set can be viewed as piles of dirt, where the amount of earth in the pile corresponds to the value of that bin. The nodes of the consumer set can be regarded as holes, where the depth of each hole corresponds to the value of that bin. The weights are the distances between a given pile and hole. The amount of work to fill a given hole with dirt from a given pile is a function of the amount of dirt to be moved from the pile to the hole and the ground distance between the two.

More formally, bins are formed by grouping reuse distances into ranges of exponentially increasing reuse distances, with the exception of the first bin which has a range of [0,4). The bins used in this work can be observed as the labels of the x-axis in Figure 4.2. Mass is the value of a given bin of a reuse signature. Ground distances refer to the distance between the indices of the supplier and consumer bin. Though bin ranges grow exponentially, their indices are linear (e.g., bin with range [0,4) has index 0, bin with range [4, 8) has index 1, bin with range [8, 16) has index 2). As an example of ground distance in the context of EMD, the distance between bin [0,4) in one histogram and bin [32, 64) in the other histogram would be:

$$\begin{aligned} abs(\quad index(\ [0,4)\ ) - index(\ [32,\ 64)\ )\quad) &= abs(\ 4 - 0\ ) \\ &= 4 \end{aligned}$$

The amount of mass located at each bin is defined by $X = x_1, \ldots, x_n$ and $Y = y_1, \ldots, y_n$, for the supplier and consumer distributions, respectively.

From this, EMD can be solved for by applying polynomial time linear programming methods [110] to minimize the following equation:

$$EMD = \min_{\forall f_{ij}} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} c_{ij} \tag{4.1}$$

where c_{ij} is the distance (cost) of moving mass from bin i to bin j and f_{ij} is the amount moved from bin i to bin j.

The minimization of EMD is subject to the following constraints:

$$f_{ij} \geq 0 \tag{4.2}$$

$$\sum_{j=1}^{n} f_{ij} = x_i, \quad x_i \in X \tag{4.3}$$

$$\sum_{i=1}^{n} f_{ij} = y_j, \quad y_j \in Y \tag{4.4}$$

In our case, we quantify the similarity between reuse distance signatures X and Y (e.g., reuse distance signatures for 64KiB and 4KiB granules), where f_{ij} is the amount of mass that will be moved from bin x_i to y_j and the cost of moving that mass is defined by c_{ij}. The amount of mass in both X and Y is normalized to 1, and our cost function is simply the difference between the given indices, i.e.,

$$c_{ij} = j - i$$

4.2.4 Memory Footprint

The memory footprint is derived from the final state of the reuse distance stack after performing reuse distance analysis at a given data block granularity. For each granularity, the memory footprint is calculated as follows:

$$S_{block\_granularity} \times N_{unique\_blocks} \tag{4.5}$$

where $S_{block\_granularity}$ is the size of the granularity used for reuse distance analysis and N_{unique_blocks} is the number of unique data blocks accessed at that granularity. Calculating the memory footprint yields a measure of how much data (in bytes) is paged in for the profiled application's region of interest.

As an example, consider the final state of the reuse distance stack in Figure 4.1(b). If we assume that the granularity of each block is 2MiB,

$$S_{block\_granularity} = 2MiB$$

$$N_{unique\_blocks} = 3$$

$$Memory\ Footprint = 6MiB$$

This calculation shows that 6MiB of data were paged when profiled in a given region of interest.

4.3 Results and Discussion

The reuse signatures for each benchmark are shown in Figure 4.2. Isolating any one granularity shows typical temporal locality information such as how a particular memory subsystem will handle the memory reference access pattern of a given application (e.g., how many off-chip memory references to expect based on the *PMF* past the capacity of the last-level cache). Analyzing the reuse signatures of different granularities (a.k.a., multi-spectral reuse distance) provides valuable insight on the spatial locality of an application.

4.3.1 Spatially Dense Memory Accesses

When comparing the different signatures, there are two prototypical behaviors as the granularity of the reuse distance analysis is increased.

The first is the shift of mass in the *PMF* towards the bins of shorter reuse distances. An example of this is the result from `464.h264ref -- 2719` in Figure 4.2. When the granularity

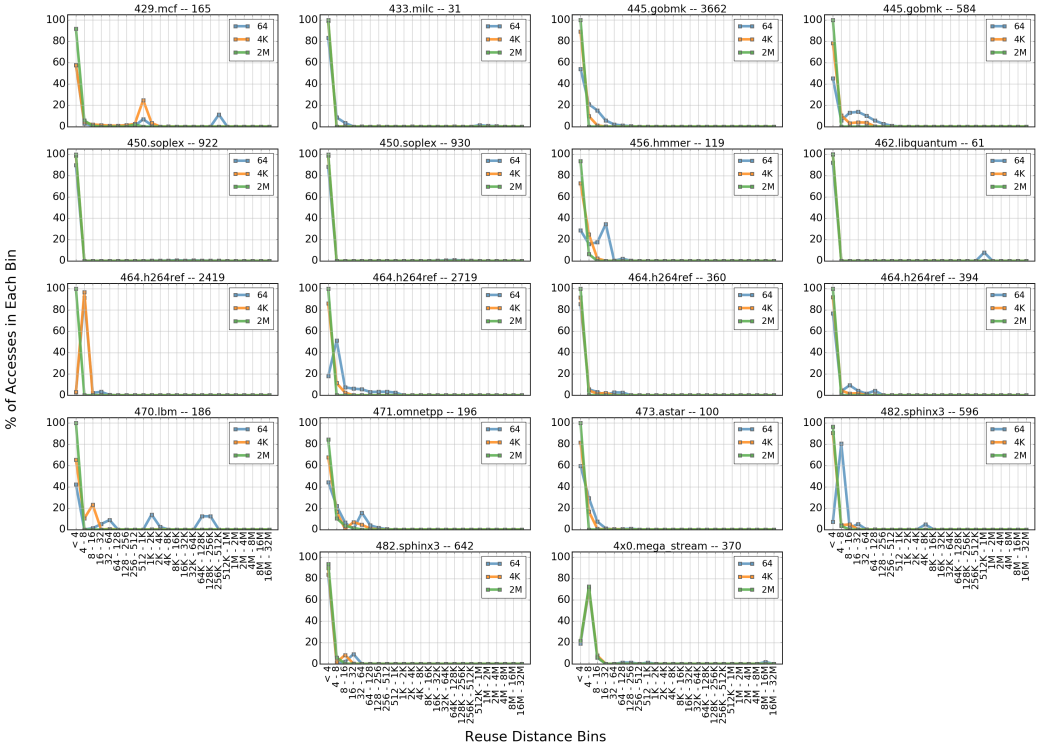


Figure 4.2: Reuse distance signatures for all benchmarks. The numbers following the name of each benchmark are the line numbers on which the regions of interest for that application start.

is 64B, almost a third of all memory references exhibit reuse distances greater than or equal to 8. At the 4KiB granularity, all memory references exhibit reuse distances no greater than 16. In the 2MiB case, virtually all reuse occurs within a reuse distance of 3.

The second behavior is the shape of the *PMF* remaining largely the same as the granularity is increased. There are two manifestations of this behavior. One is when the mass of each of the reuse signatures are contained mostly in the first bin. The result from `450.soplex -- 930` in Figure 4.2 shows almost identical reuse signatures for all 3 granularities, where 90% of the memory references happen within a reuse distance of 3 when the granularity is 64B, and 100% for 4KiB and 2MiB. The other manifestation is shown in the result from from `4x0.mega_stream`. For the 64B granularity, 70% of the *PMF*s mass is located in the [4,8) bin. While increasing the granularity to both 4KiB and 2MiB captures some of the mass to the right of this bin in the 64B case, the shape of the distribution remains largely unchanged.

The shifting (or not) of the *PMF* from higher to lower reuse distances bins as the granularity increases serves as a measure for how spatially dense the memory references are. A shift is indicative of memory references that reside on different data blocks at one granularity but reside on the same data block at a larger granularity. For example, refer back to the example reference trace in Section 4.2.2 and assume the granularity to be 64B. If references a, b, and c all reside on the same 4KiB data block, then when the reuse distance analysis is conducted at 4KiB granularity, then the reuse distance becomes 0 for all references. This is because the 64B data blocks that a, b, and c resided on were subsumed by the same 4KiB block. This is representative of the first prototypical behavior. If references a, b, and c reside on different 4KiB data blocks, then the reuse distances remain the same because they will not be subsumed by the same 4KiB block. Thus, we are able to observe the spatial locality for memory references by performing reuse distance analysis at different granularities.

Directionality of Mass Shift

Additionally, it is possible to formally prove the directionality of the mass shift that occurs when comparing the reuse signature of a smaller granularity to a larger one. In general, if we view the virtual address space of a process divorced from the physical address space underlying it, then we can view it as a contiguous space $\mathbb{A}$. Realistically this space has a natural range from 0 to $(2^{64}-1)$ for most 64-bit architectures. Calculating the reuse distance as previously defined in Section 4.2.2, with a single bin size of $\mathbb{A}$ would result in a distance of zero and nothing else. Consider dividing this single space $\mathbb{A}$ into two spaces (as illustrated in Figure 4.3), denoted as set $\mathbb{B}$, $\frac{\mathbb{A}}{2} \rightarrow \{\mathbb{B}_0, \mathbb{B}_1\} = \mathbb{B}$. There are two spaces and two possible reuse distances: zero and one. Each of these spaces has the relation (when comparing the size of each space, or granularity of reuse bin) of: $|\mathbb{A}| > |\mathbb{B}_0| = |\mathbb{B}_1|$. It follows, then, that regardless of the the reuse bin within set $\mathbb{B}$, when superimposed over the larger set $\mathbb{A}$, the reuse distance will be zero with regards to that set. Dividing the subsets of $\mathbb{B}$ yet again yields four spaces, which we denote as set $\mathbb{C}$ corresponding to four reuse distance bins. All valid programs must fit within the space of $\mathbb{A}$. The same cannot be said of the subsets of $\mathbb{B}$ or $\mathbb{C}$. It is expected, and required, that the next larger set will subsume smaller ones. These sets are equivalent to the reuse distance granularities we have chosen, as an example, $\mathbb{B}$ could equal 2MiB, $\mathbb{C}$ could equal 4KiB, etc. If, as we have described with the multiple sized sets, we instead have multiple fixed sizes of reuse distance bins, then the bin widths should exhibit the same pattern and directionality. That is, if the distributions of each granularity are ordered with the smallest granularity bin widths in front and the largest granularity widths in back (if on a three-dimensional axis, the *PMF* of each reuse distance measurement would have the probability on the y-axis, the bin count on the x-axis, and the z-axis would be ordered from smallest to largest), then we would expect the mass when moving from front to back (with respect to the z-axis) to slide towards the zero bin of the largest granule.

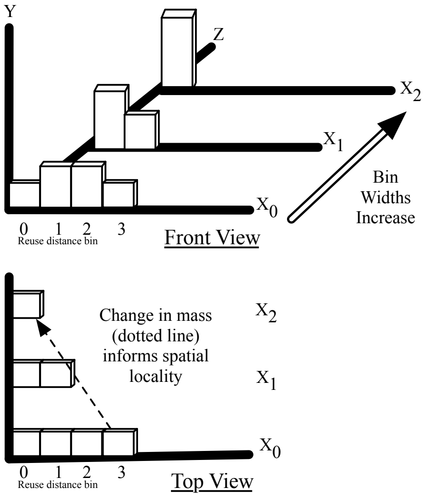


Figure 4.3: Visual representation of trend described in Section 4.3.1. X_0 corresponds to set $\mathbb{C}$, X_1 corresponds to set $\mathbb{B}$, X_2 corresponds to $\mathbb{A}$. The bottom graph is the view from "above" of the x and z axis showing the trend of changing mass that is expected of all applications as the bin size of each X_i approaches infinity. The rate of change in the mass (essentially slope of the line along this axis) informs the spatial locality, quantitatively measured in this work as EMD.

When ordered in this way, taking the multi-spectral reuse distance measurement has two immediate consequences we can exploit: when moving along the z-axis, we can qualitatively assess spatial density and the degree by which larger granules subsume (or do not subsume) smaller ones based on changes along the x- and y- axes. Second, with sufficiently large reuse distance bins, the mass will always converge to a zero reuse distance bin when moving in a positive direction along the z-axis (smaller reuse distance widths to larger ones).

EMD as a Spatial Locality Measure

The amount of mass that is shifted from one distribution to another is empirically shown by computing the Earth Mover's Distance between them, as described in Section 4.2.3. The results of comparing the 64B and 4KiB distributions and the 4KiB and 2MiB ones are shown in Figure 4.5. The closer the EMD is to zero, the more similar the distributions are.

It follows that EMDs that approach zero demonstrate behavior in which larger data block granularities do not subsume smaller ones (within the range of granularities measured, as proven previously, eventually they will always be subsumed), and that their memory reference patterns are less spatially dense (i.e., having parts close together) than two distributions that express a large EMD.

For example, the `470.lbm -- 186` benchmark has the highest EMD score among all of the 64B vs. 4KiB comparisons. From Figure 4.2, at the 64B granularity, over 20% of all reuse distances are at least 4MiB away. However, we observe qualitatively in the shifting of mass from 64B to 4KiB in Figure 4.2, and quantitatively with Figure 4.5 an EMD that is much greater than zero, that much of the necessary data for computation is resident on the same 4KiB data blocks. The implications of these observations will be explored in the following subsections.

4.3.2 Page Sizing and Utilization

Page Sizing

The reuse signatures and their respective EMD results also have implications for selecting the page size for a given computer system. In many system architectures, it is possible to alter the page size from 4KiB or 8KiB to something larger to try and exploit spatial locality and reduce translation overhead. From the spatial locality information that results from Figures 4.2 and 4.5, it is possible to evaluate whether there are any performance benefits to increasing page size.

Referring to the `464.h264ref -- 2719` benchmark, we observe mass shifting in its reuse signatures and EMD scores that are greater than zero. In fact, at the 2MiB granularity, all of the data required for this computation is resident within strides of 0 to 8MiB, i.e., all of the mass is located in the first bin. This suggests an extremely dense spatial locality access pattern, which would benefit from larger pages.

Antithetical to this are the results from the `4x0.mega_stream -- 370`, which qualitatively in Figure 4.2 shows no shift in mass and has a very small EMD at all granularities. Specifically, it is shown that at least 75% of all reuse distances are occurring between 8 and 16 at all granularities. At the largest granularity, 75% of all accesses are touching data resident on at least 4 different 2MiB pages before that data is reused again. Larger page sizes are not subsuming the memory references from smaller granularities. Thus, larger page sizes cannot extract spatial locality from applications in which that spatial locality does not exist.

Page Utilization

The memory footprint data, calculated using 4.5, for each benchmark is presented in Figure 4.4. Each granularity is normalized to the 64B case. From this, it is possible to determine how much extraneous data, if any, is paged in when larger pages are used. When looking at Figure 4.4, any bar that extends past the black dotted line indicates that more memory was paged in than was necessary. We will investigate this idea further in the remainder of this section.

The `462.libquantum -- 61` benchmark results from Figures 4.2 and 4.5 show benefits for increasing larger page sizes, while also fully utilizing the data that is paged in. This is evidenced by the amount of data paged in at the 2MiB granularity being almost equal to the amount paged in for the 64B case. Referring to Equation 4.5, the $S_{block\_granularity}$ term will be larger in the 2MiB case than for the 64B case, but the spatially local accesses at the larger granularity decrease the N_{unique_blocks} term such that the memory footprint of the two cases are almost equal. We will now examine applications for which non-spatially local accesses result in bigger discrepancies in memory footprint at their respective measured granularities.

Looking at `464.h264ref -- 2419` and `464.h264ref -- 2719`, however, we observe that, although the 2MiB page size subsumes the smaller granules, the 2MiB page size actually pages in 10× and 100× more data, respective to each function, than is actually necessary, assuming that every byte of each 64B data block pulled in is fully utilized (*note:* this is a strong assumption given the previous characterizations of *Dark Bandwidth*[8]). Thus, using a 2MiB page size for this application puts undue stress on the coherence bus, and wastes a considerable amount of energy since it has to move 10× and 100× more data than is actually necessary.

The `4x0.mega_stream -- 370` benchmark is particularly interesting because it has been previously shown that its spatial locality access pattern is not dense, and that larger pages do not subsume the smaller data block granules and help with spatial locality. However, virtually all of the data that is paged in, even at the 2MiB granularity, is used as shown in Figure 4.4. Thus, the page utilization is very good for this application. This result indicates that it may be a prime candidate for a data layout transformation in order to reduce the amount of data movement and increase the amount of available physical at any given instant. The spatial and temporal locality patterns of this benchmark indicate that multiple values are pulled from each page at any given instant. However, streaming them in a packed fashion would improve the utilization over any given time window (recall that the overall utilization is large, but only after the entire application has executed).

4.3.3 Data Layout Transformation

The layout of the data necessary for the computation directly impacts the spatial locality characterization of an application. Recent work such as [7] shows that data movement can be reduced by transforming the layout of data near memory to better exploit spatial locality for current memory subsystem and reduce superfluous data movement. Given that a data layout transformation is possible at multiple levels of the memory hierarchy, it is possible to better determine at which level to perform the data layout transformation. We can identify the levels to perform the data layout transformation using the memory footprint analysis performed in this work.

In the case of `4x0.mega_stream`, the memory footprint data shows that, even at the largest page size, all of the data that gets paged in eventually gets used. Since even at such a large granularity the spatial access is not dense, it would be beneficial to perform the data layout transformation nearer the data, so that the data that gets paged in is densely packed, which will reduce the amount of fast physical memory that must be utilized, improve cache utilization, and lastly reduce the overall energy of computation. The last improvement would primarily be due to the reduced need to refresh DRAM rows [63] compared to a non-data layout transformation case (as less physical DRAM need be provisioned). When using a data layout transformation mechanism such as SPiDRE [7], the data could be streamed as needed potentially reducing the need to store data in DRAM.

4.4 Conclusion

The problem of efficiently feeding processing elements and finding ways to reduce data movement is a pervasive problem in computing. Efficient modeling of both temporal and spatial

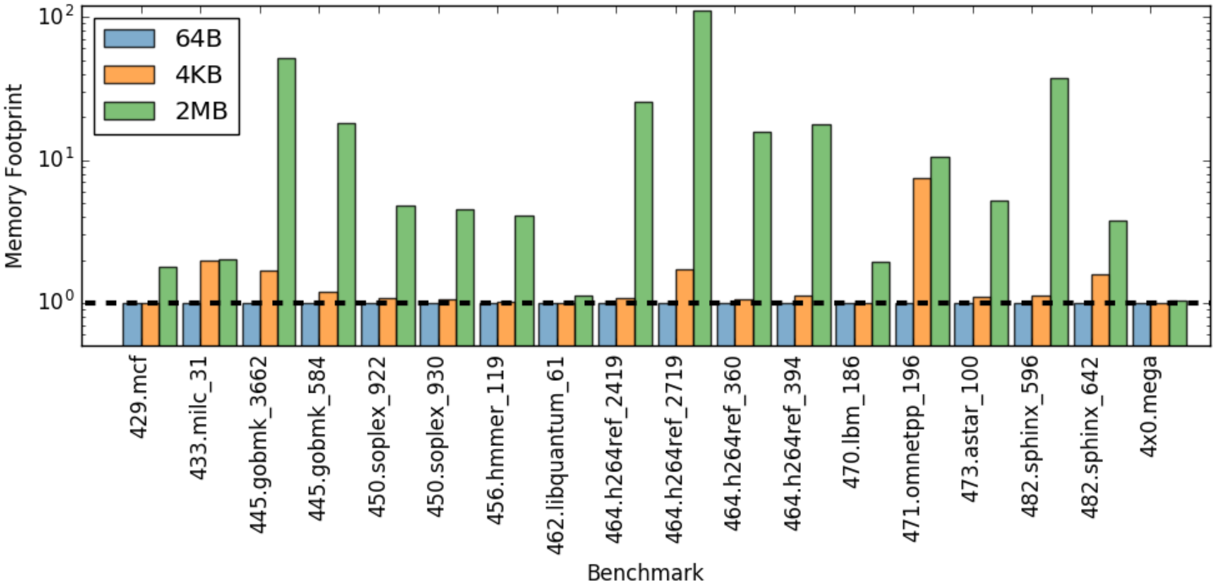


Figure 4.4: Memory footprint normalized to 64B granularity.

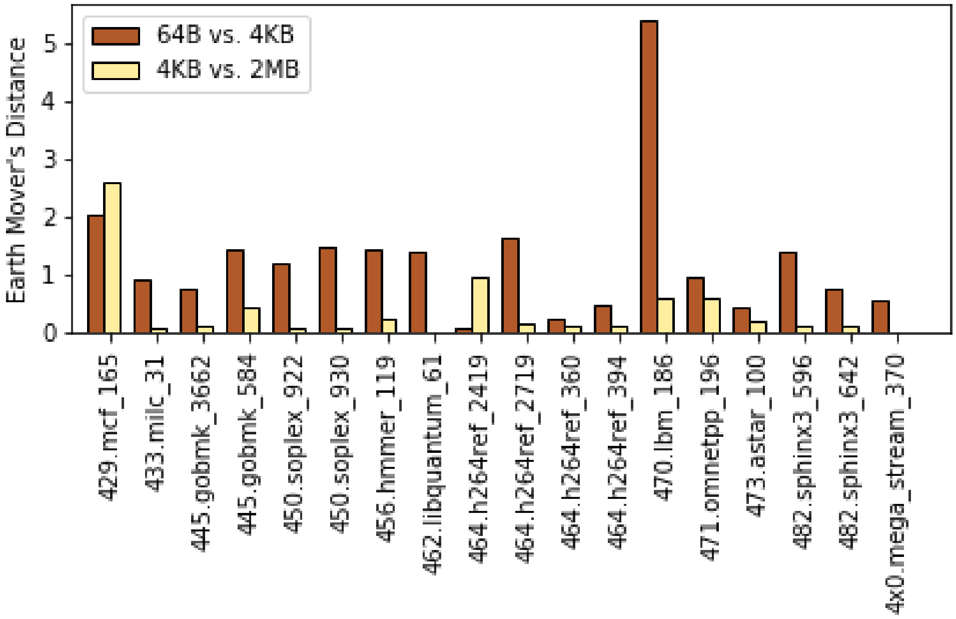


Figure 4.5: Comparing (64B, 4KiB) and (4KiB, 2MiB) reuse signatures using Earth Mover's Distance.

locality of memory references is invaluable in identifying superfluous data movement in a given application.

In this work, we have presented a way to model both spatial and temporal locality using what we term “multi-spectral reuse distance,” derived from classic reuse distance analysis. Reuse distance is a metric is traditionally used to determine the temporal locality of an application. Multi-spectral reuse distance is measured by performing reuse distance measurement at differing reuse distance granularities, in example, 64B, 4KiB, and 2MiB sizes. This approach allows for a qualitative observation of spatial locality, through observing the shifting of mass in an application’s reuse signature at different granularities. Furthermore, this be quantified through the Earth Mover’s Distance between ordered sets (ordered on reuse distance bin size) of probability mass functions of an application. It is these sets of PMFs that define the multi-spectral reuse distance. This characterization was performed on a subset of the SPEC2006 benchmark, as well as a streaming mini-application characteristic of stencil calculations.

From the multi-spectral characterization, it is possible to determine how spatially dense the memory references of an application are based on the degree to which the mass has shifted (or not shifted) and how close (or far) the Earth Mover’s Distance is to zero as the data block granularity is increased. It is also possible to make inferences based on this information as to the appropriate page size, and whether or not a given page is being fully utilized. From the applications profiled, it is observed that not all applications will benefit solely from having a larger page size. Additionally, larger data block granularities subsuming smaller ones suggest that larger pages will allow for more spatial locality exploitation, but examining the memory footprint will show whether those larger pages are fully utilized or not. Finally, it is possible to infer where in the memory hierarchy a data layout transformation could be beneficial in order to more efficiently move data by observing the data utilization within given data page.

In Chapter 6, we will measure multi-spectral reuse distance on the DIBS application, and use the generated locality measures as an input to an unsupervised learning technique to quantitatively inform a hardware design choice regarding width versus depth. Before we explore that, though, we present in the following chapter an evaluation of the Intel HARPv2 CPU+FPGA system, how to architect designs for it, and frame the design of domain specific hardware in Chapter 6.

Chapter 5

Evaluating Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2

As the end of Moore's law draws nearer, researchers across disciplines are looking beyond relying on performance increases through packing more transistors into CPUs and scaling CPU clock frequencies. Outside of multicore CPUs, people are turning towards heterogeneous computing solutions that incorporate hardware coprocessors such as Graphics Processing Units (GPUs) and Field Progammable Gate Arrays (FPGAs) to accelerate computation. The former has become ubiquitous in desktop, server, and cloud environments and has an established and mature ecosystem.

The widespread use of FPGAs, however, is still nascent while their presence is burgeoning. This forward progress is reflected in industry with companies like Amazon and Microsoft equipping their data center nodes with FPGAs [107, 25, 5] and Intel acquiring FPGA manufacturer Altera. Additionally, there is a growing research trend toward harnessing the reconfigurability of FPGAs towards accelerating salient applications like neural networks [23, 43, 143], biocomputation [62, 89, 94], and many other applications [88, 117, 125, 142, 150].

A common way to incorporate hardware acclerators like GPUs and FPGAs into a computer system is to attach them through a PCIe card, which keeps hardware costs relatively low. In spite of this, the use of FPGAs has not experienced the widespread adoption that GPUs have seen, in part because of all the difficulties inherent in their use. Historically, FPGA developers have needed to be well versed in electronic circuits and digital logic design. This includes knowledge of low-level hardware interaction at the register-transfer level (RTL) and handling timing constraints at a clock cycle granularity, as well as domain-specific knowledge of computer-aided design tools and workflows specific to FPGA design and development. This is generally outside of the skillset of most software developers.

One of the steepest barriers to using FPGAs is expressing a design in the first place using traditional hardware description languages (HDLs) like VHDL and Verilog, which requires the domain specific knowledge previously mentioned. A current research direction in lowering the barrier is High Level Synthesis (HLS), which allows a programmer to express a kernel of computation in a higher level language like C or C++ for deployment onto an FPGA. This circumvents the problem of having to learn an HDL to express a kernel and its low level interfaces, reduces the amount that a programmer has to understand about FPGA microarchitecture, and abstracts away the lower level details of using FPGAs.

One way that companies that build PCIe cards around FPGAs enable the use of HLS is through making their solutions OpenCL compliant. This involves providing a Board Support Package (BSP) that provides the interface between host and device, as well as parameters that are used by an offline compiler to synthesize, place, and route a design onto whatever FPGA is used on the card. In addition to PCIe cards, Intel has also developed a system that incorporates both a multicore Xeon CPU and Arria 10 FPGA into the same chip package (as described in Section 2.4.1). This particular project is known as the Heterogeneous Accelerator Research Platform, or HARP. In addition to being able to author designs using an HDL, Intel has provided the infrastructure to use the Intel FPGA OpenCL SDK for FPGA development. While there have been recent publications targeting this system with a traditional FPGA design flow [4, 28, 117, 122, 135, 144], not much is known about the experience, feasibility, and performance of targeting a HARP system using OpenCL.

In this chapter, we will target the second iteration of the HARP CPU+FPGA processor (HARPv2) through HLS using the Intel OpenCL SDK for FPGA to evaluate the portability and performance of OpenCL FPGA kernels. Specifically, we use OpenCL kernels authored for an FPGA attached via PCIe card that perform the Needleman-Wunsch algorithm [149] and port them to the HARPv2 system. Then, we evaluate and compare our results to ones previously reported, present our findings of portabilty through exploring the hardware design space, and show the benefit of using the Shared Virtual Memory (SVM) abstraction implemented for the HARP system.

5.1 Methods

This section describes the kernels built for and deployed on the Intel HARPv2 system. First, we describe the Needleman-Wunsch algorithm used in our study. We then describe each of the kernels synthesized for the HARPv2 system. We use the same kernel version enumeration from [148]. The kernels are built using the offline compiler provided in the Intel FPGA OpenCL SDK and a custom release of the 16.0.2 version of the Intel Quartus Prime tool suite that accomodates the HARPv2 system. This is the most recent version of the Quartus tools that is supported by the test system. Minimal changes were made to the original host source code during the porting process. The only change made to the kernel code was correcting an indexing error in Kernel Version 5 that left the last column and the last two rows unprocessed. The process of exploring the hardware design space is detailed next. Finally, we detail how we enable the HARPv2 system, including using the Shared Virtual Memory (SVM) abstraction.

5.1.1 Needleman-Wunsch

The workload targeted in this paper is the Needleman-Wunsch algorithm. It is a dynamic programming algorithm used for globally aligning a pair of protein or nucleotide sequences. The end result of the algorithm is a substitution matrix that is used to trace back the optimal global alignment of the two sequences. The general structure of the implementation (without any notion of blocking or parallelism) is outlined in Algorithm 1.

Algorithm 1: Needleman-Wunsch Algorithm

```
 1: new int subst_matrix[N+1][N+1], score_matrix[N+1][N+1]
 2: new int gap_penalty
 3: initialize first row and column of subst_matrix
 4: initialize score_matrix
 5: initialize gap_penalty
 6: for $i \leftarrow 1$ to $N+1$ do
 7:    for $j \leftarrow 1$ to $N+1$ do
 8:       top = subst_matrix[$i-1$][$j$] - gap_penalty
 9:       left = subst_matrix[$i$][$j-1$] - gap_penalty
10:       top_left = subst_matrix[$i-1$][$j-1$] + score_matrix[$i$][$j$]
11:       subst_matrix[$i$][$j$] = max(top, left, top_left)
12:    end for
13: end for
```

The size of the substitution matrix is determined by the length of the two strings to be compared. In this case, the two strings are both of size N, and an extra row and column are added for initial conditions. The substitution matrix is populated by iterating across all elements in each row as detailed in the nested `for` loop starting in line 6. Each element *subst_matrix*$[i][j]$ is calculated as a function of the elements to the left, top, and top left from the current element, as well as a similarity score from *score_matrix*$[i][j]$ and a predetermined penalty constant for alignment gaps. An example of calculating $subst\_matrix[2][2]$ in a 3×3 substitution matrix is shown in Figure 5.1.

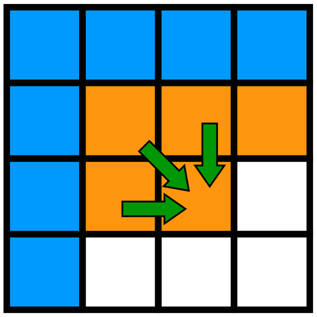


Figure 5.1: An example of the left, top, and top left dependencies for calculating $subst\_matrix[2][2]$ in a 3×3 substitution matrix.

The Needleman-Wunsch algorithm is included as part of the Rodinia benchmarking suite [26] developed by Che et al. In Rodinia, the Needleman-Wunsch algorithm has 3 different implementations: an OpenMP implementation that targets multicore CPUs, a CUDA implementation that targets NVIDIA GPUs, and an OpenCL implementation for any accelerator that is compliant with the OpenCL standard. Zohouri et al. extended this work by authoring kernels using the Intel FPGA OpenCL SDK to target FPGAs connected as a card on the PCIe bus [149, 148]. They do this for a subset of the Rodinia suite and show the advantages and disadvantages of FPGAs as accelerators compared to GPUs and multicore CPUs. We use the Needleman-Wunsch FPGA kernels from Zohouri et al. for experimentation in this work, and detail each version in Section 5.1.2.

5.1.2 Description of Each Kernel Version

The kernel versions used in this paper are from [26, 148, 149] and are described in the following subsections. Versions 0 and 2 are designed using the NDRange (NDR, and also referred to as MWI) paradigm, and Versions 1, 3, and 5 use the SWI paradigm. Versions 2 and 3 apply basic level optimizations to their preceding versions, and Version 5 implements a new design using the SWI model. Each kernel was built for and deployed on the Intel HARPv2 system for comparison to the prior work that evaluates performance with FPGAs that are connected via PCIe card.

Version 0

This kernel takes the OpenCL implementation from [26], which uses 2D blocking to subdivide the problem with no modifications and is used as the performance baseline. Its implementation follows the NDR paradigm. The kernel is divided into two separate kernel functions that perform the same computation but are indexed differently to compute the upper and lower trangular, respectively, of a given 2D block. Each function takes advantage of diagonal parallelism in two ways: thread- and block-level parallelism. Once an element or block of index (i, j) is computed, it satisfies the dependencies for the $(i, j+1)^{th}$ and $(i+1, j)^{th}$ elements or blocks, and allows them to be computed in parallel. An illustration of this diagonal parallelism is shown in Figure 5.2. The size of the 2D block is determined by a user-defined variable named `BSIZE`.

Version 1

This kernel uses a doubly nested `for` loop as outlined in Algorithm 1 and takes no steps to guide the synthesis tools on how to better achieve computational parallelism in the resulting custom pipeline.

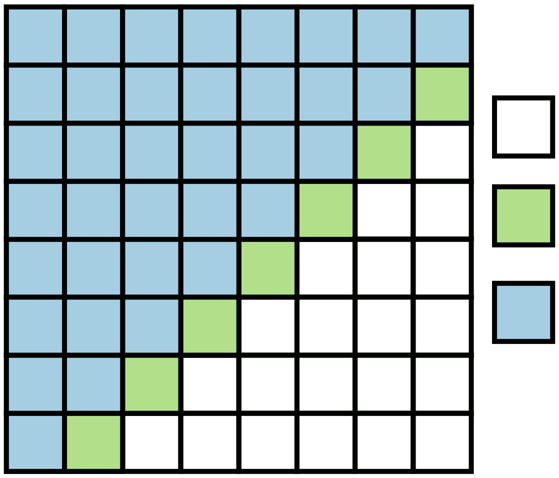


Figure 5.2: An illustration of the diagonal parallelism available at the thread and block levels in the Needleman-Wunsch algorithm. The blue squares are elements that have already been computed. The green squares are ones that are available to be computed because their top, left, and top left dependencies have been satisfied. The white sqaures are elements that have yet to be computed because their dependencies have not yet been satisfied.

Version 2

This kernel applies basic compiler-level optimizations [149] to Version 0 in two ways. The first is through setting the maximum work group size. This constraint allows the compiler to perform more aggressive optimizations without wasting precious hardware resources [60]. Additionally, setting the size also enables the second optimization: kernel vectorization. This is achieved by adding the `SIMD` attribute to the kernels in order to vectorize them. This allows work items to execute more data. In this work, the kernels are vectorized by a factor of 2.

Version 3

Version 3 improves on Version 1 in two ways. The first is by adding a register to cache the result of the current element so that it can satisfy the left dependency for the next iteration and avoid an external memory access. The second is by adding the compiler pragma `ivdep` on the substitution matrix to prevent compiler from assuming false load/store dependencies on that global buffer (since the current element being computed depends on previously computed values in that buffer) and to decrease stall cycles per loop iteration.

Version 5

Version 5 is a kernel programmed using the SWI model. Instead of iterating across elements left to right as in previous SWI implementations, Version 5 takes advantage of diagonal parallelism similar to the NDR kernel versions. It divides the substitution matrix into groups of rows, i.e. 1D blocks. The number of rows in each 1D block is set by a tunable hardware parameter named `BSIZE`. Each 1D block of the matrix is processed by dividing the block into column chunks. Specifically, there is a hardware parameter named `PAR` that sets how many columns are in each chunk. The chunks of columns are processed in a diagonal fashion, and wrap around to the next chunk of columns once the current one is finished. The kernel is done processing once all of the columns of the 1D chunk have been computed. The exit condition is precomputed on the host side.

While there are other optimizations employed as described in [148], the main optimization is use of shift registers as local storage to satisfy dependencies. This is done in two ways. The first is by using shift registers, like the one featured in Figure 5.3, to hold onto computed elements of the substitution matrix between iterations of the loop.

This is similar to the idea of caching a computed element in Version 3, but the shift registers act as buffers that satisfy dependencies across multiple rows and columns instead of just the next element to be computed in the substitution matrix. The size of these shift registers are a function of `BSIZE` and `PAR`. The second way is by creating 2D shift registers and utilizing

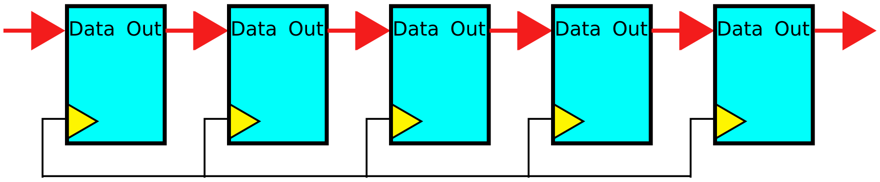


Figure 5.3: An example of a shift register comprised of 5 D-Flip Flops.

them in a staircase fashion as shown in Figure 5.4. Because each 1D block is traversed in a diagonal fashion, the access patterns of global memory are not spatially local. To this end, the staircase shift registers are employed such that reads and writes to global memory can still be coalesced, but are buffered until they are needed to compute an element in the substitution matrix. An example of this is shown in Figure 5.5.

5.1.3 Hardware Design Space Search

In [148], Zohouri reports the optimal parameter settings for `BSIZE` in kernel Versions 0 and 2 and the optimal settings for `BSIZE` and `PAR` for Version 5 for the PCIe-connected Stratix V and Arria 10 FPGAs that he uses in his experimentation. These parameters are effectively hardware design knobs that are exposed by the kernel designer. `BSIZE` controls how much of the substitution matrix was computed for the NDR kernels, and is a parameter for sizing some shift registers in kernel Version 5. The `PAR` parameter controls the degree of parallelism for kernel Version 5, i.e. how many substitution matrix elements can be processed at the end of a loop iteration, and determines how large to make the staircase shift register array, as shown in Figure 5.4. In order to find the parameter configurations for the Intel HARPv2 system that produce optimal performance, we define a hardware design space by creating a range of values that `BSIZE` can take for Versions 0 and 2, and a range of values that `BSIZE`

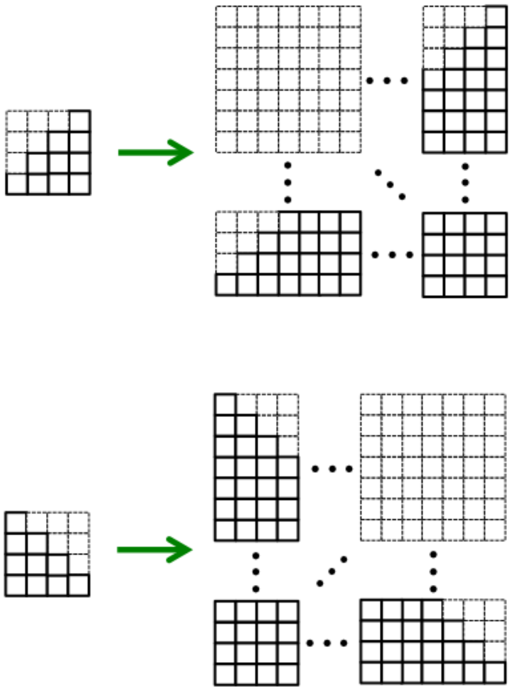


Figure 5.4: Hardware effect of staircase shift register when sweeping the PAR parameter. A 2D array of shift registers is allocated in the OpenCL kernel, but only half of the structure is used. This is represented by boldening the lines of the utilized shift registers and using dashed lines to represent the shift registers that are unused. The top figure is the shape of the arrays used to buffer data and coalesce reads from global memory, and the bottom is the shape used to buffer data and coalesce writes to global memory.

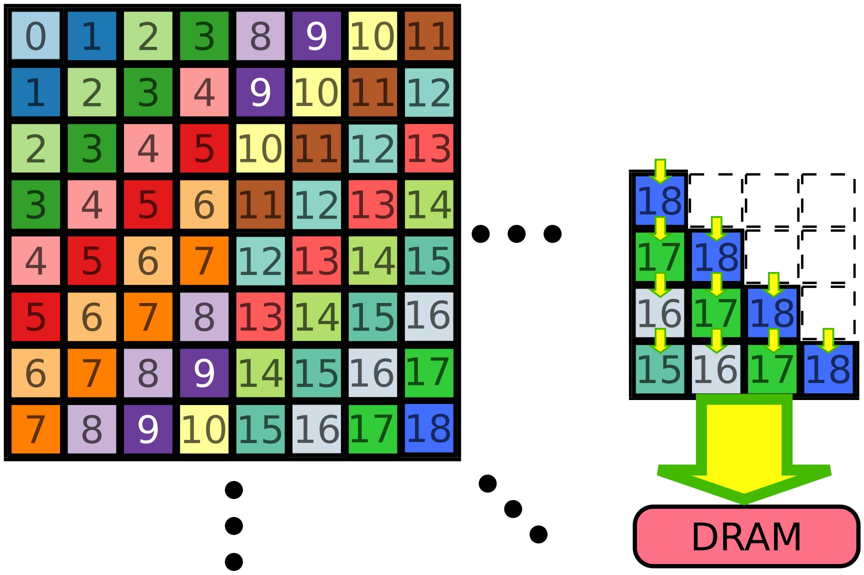


Figure 5.5: An illustration of exploiting diagonal parallelism paired with a staircase shift-register whose purpose is to queue writes to global memory until a contiguous chunk of memory can be written. The numbers represent the index of an iteration in the computational loop. Squares that share the same index (and subsequently the same color) imply that those particular elements are computed during the same iteration. Because of the shift-registers though, elements computed during the same iteration will be written back at different times in order to achieve more spatially local writes.

and PAR can take for kernel Version 5. This range is defined, in part, by what configurations the tools are able to successfully build.

5.1.4 Shared Virtual Memory

Though the Intel HARPv2 nodes used in this paper are technically only OpenCL 1.0 compliant (as reported by querying the CL_DEVICE_VERSION parameter of the device), they do support the feature of using Shared Virtual Memory (SVM) implemented as an extension to the OpenCL 1.0 API. It is worth noting, though, that devices compliant with versions of OpenCL 2.0 and up are required to support SVM.

Instead of having to explicitly enqueue writes and reads to and from the HARPv2 FPGA, shared memory is allocated on the host side and then is pointed to as a special SVM kernel parameter from the host code. In order to utilize this feature in the HARPv2 system, the host code needed to be edited in the following ways: all previously created cl_mem objects created and freed for the device were removed and replaced with shared memory allocated by clSVMMallocAltera() and clSVMFreeAltera(), respectively. Enqueueing writes and reads to cl_mem objects were removed. Finally, calls to clSetKernelArg() that pointed to cl_mem objects were replaced with calls to clSetKernelArgSVMPointerAltera() that pointed to shared memory buffers. Conveniently, no changes need to be made to the kernel code to accomdate using SVM instead of explicit reads and writes. Thus, kernels do not need to be rebuilt to accomodate using the SVM feature.

5.2 Results and Discussion

5.2.1 FPGA Kernel Results

Table 5.1 shows the results of building and evaluating each kernel version described in Section 5.1.2 on the Intel HARPv2 system and compares them to previously reported results in [148]. Specifically, any row that contains ”HARP” in the ”FPGA” column contains our results for the Intel HARPv2 platform, and all other data is from [148]. The percentages reported are how much of that particular FPGA resource is utilized relative to the amount available. The execution times presented are the lowest times over 100 runs of the respective kernels. The *Speedup* column is the calculated speedup relative to the Stratix V result from [148] for kernel Version 0.

Comparing results from [148] to those observed from the HARPv2 system, the trends regarding *NDR* and *SWI* kernels reported in [148] also appear here. The applied optimizations to Versions 0 and 1 result in decreases in execution time relative to each kernel's respective runtime. Version 2 executes 5.25 faster than Version 0, and Version 3 executes 270.49 times faster than Version 1. However, our HARPv2 system results, except for Version 2, execute slower than those from [148], despite the Arria 10 FPGA having more resources to use than the Stratix V FPGA.

The biggest contributing factor to this is the amount of resources needed to implement designs. This causes the place and route process of the FPGA to be more difficult, involving more complex routing solutions that drive the maximum possible clock speed down. In almost all cases, the Arria 10 FPGA HARPv2 system uses a larger percentage of its available resources than the Stratix V FPGA does, which has less resources to begin with. In all cases, f_{max} for the Arria 10 is lower than the those for the Stratix V.

Version	Opt. Level	Kernel Type	FPGA	Time (sec)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Speedup
v0	None	NDR	Stratix V, PCie	9.937	267.52	27%	16%	30%	6%	1.00
			Arria 10, HARP	13.367	211.77	25%	39%	25%	1%	0.74
v1	None	SWI	Stratix V, PCie	203.864	304.50	20%	5%	17%	$< 1\%$	0.05
			Arria 10, HARP	830.131	256.6	26%	9%	18%	$< 1\%$	0.01
v2	Basic	NDR	Stratix V, PCie	3.999	164.20	38%	68%	100%	8%	2.48
			Arria 10, HARP	2.545	162.865	50%	47%	81%	1%	3.90
v3	Basic	SWI	Stratix V, PCie	2.803	191.97	19%	8%	18%	$< 1\%$	3.55
			Arria 10, HARP	3.069	178.12	25%	10%	19%	$< 1\%$	3.24
v5	Advanced	SWI	Stratix V, PCie	0.260	218.15	53%	7%	28%	2%	38.22
			Arria 10, PCie	0.176	201.06	28%	8%	25%	$< 1\%$	56.46
			Arria 10, HARP	0.290	186.81	40%	19%	30%	$< 1\%$	34.27
Dummy	N/A	N/A	Arria 10, HARP	N/A	350.26	23%	6%	14%	0%	N/A

Table 5.1: Results of executing the Needleman Wunsch kernel versions on the HARPv2 System and how they compare to results in [148]. The values in the *Speedup* column are relative to the kernel Version 0 Stratix V result. The first two rows for kernel Version 5 are both results from [148]. The last row of the table is the result for building a “Dummy” kernel that is simply a kernel that contains no computation.

This is because of all of the resources necessary to implement the BSP components that interface to the host CPU to the FPGA. The last row in Table 5.1 shows the resource utilization for a "Dummy" kernel, which is an OpenCL kernel that contains no computation in its function body. We use this as a proxy for the resources required to implement the interface BSP components. As a comparison, consider the *Arria 10, PCIe* result for kernel Version 5, which uses the same FPGA. The percentage of total logic blocks used is 28%, compared to the 23% of logic blocks used just to implement the BSP for the HARPv2 system. Though addressing this shortcoming is compounded by the opacity of the toolflow for the Intel FPGA OpenCL SDK, work done by Sanaullah and Herbordt describe a methodology to isolate the HDL generated from the toolflow [113]. This is done, in part, to classify the common interfaces generated by the tools and either remove unnecessary parts or modify unoptimized parts of the OpenCL-generated HDL to reduce the amount of FPGA resources necessary to build the design and increase performance.

Another inefficiency that is specific to the implementation of kernel Version 5, but applies to both the PCIe and HARPv2 systems, is the way the staircase shift registers are implemented. In the kernel source, this is done by allocating local space for a 2D array and then inferring a shift register from it. Though they are synthesized as a 2D shift register, as shown in Figure 5.4, only half of it is used. A more efficient approach would be to allocate `PAR` shift registers that are the exact size needed to achieve the buffering effect explained in Section 5.1.2. However, this is more complex than just allocating a 2D array and inferring a shift register because it involves further tweaks to the OpenCL kernel such as manual unrolling of loops to account for boundary conditions in the algorithm. This problem exemplifies the tradeoff of productivity versus performance.

Kernel version	PAR	BSIZE	Time (sec)	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	DSP	Build Time (hr:min:sec)
v0	N/A	64	20.530	232.665	30%	16%	28%	1%	11:43:22
		128	13.367	211.77	31%	25%	39%	1%	9:8:36
		256	15.836	153.985	31%	59%	81%	1%	12:56:10
v2	N/A	8	2.545	162.865	50%	47%	81%	< 1%	12:24:00
		16	16.735	182.415	35%	40%	58%	< 1%	7:26:49
v5	8	256	1.011	215.4	27%	12%	21%	< 1%	5:29:05
	8	512	1.035	216.26	27%	12%	21%	< 1%	5:34:20
	8	1024	1.156	213.67	27%	12%	21%	< 1%	14:35:26
	8	2048	1.210	215.26	28%	12%	21%	< 1%	14:22:57
	8	4096	1.227	214.17	27%	12%	22%	< 1%	13:35:46
	8	8192	1.270	213.44	27%	13%	22%	< 1%	6:23:05
	16	256	0.417	200.8	30%	13%	23%	< 1%	9:48:13
	16	512	0.410	209.16	30%	13%	23%	< 1%	9:27:11
	16	1024	0.414	197.86	30%	13%	23%	< 1%	15:25:55
	16	2048	0.437	205.42	30%	13%	24%	< 1%	9:53:23
	16	4096	0.449	196.54	30%	14%	24%	< 1%	6:00:06
	16	8192	1.267	199.84	30%	14%	24%	< 1%	15:21:19
	32	256	0.338	171.02	40%	19%	30%	< 1%	22:30:28
	32	512	0.312	180.27	40%	19%	30%	< 1%	22:18:48
	32	1024	0.298	179.01	40%	19%	30%	< 1%	21:41:14
	32	2048	0.292	186.81	40%	19%	30%	< 1%	18:38:15
	32	4096	0.296	179.5	40%	19%	30%	< 1%	8:04:14
	32	8192	0.297	187.37	40%	19%	30%	< 1%	8:27:33
	64	2048	0.363	117.85	66%	30%	47%	< 1%	31:10:3
	64	4096	0.330	129.04	67%	30%	47%	< 1%	44:54:53
	64	8192	0.332	128.22	67%	31%	48%	< 1%	38:16:34

Table 5.2: Results for sweeping the `BSIZE` parameter for kernel versions 0 and 2 and the `BSIZE` and `PAR` parameters for kernel version 5.

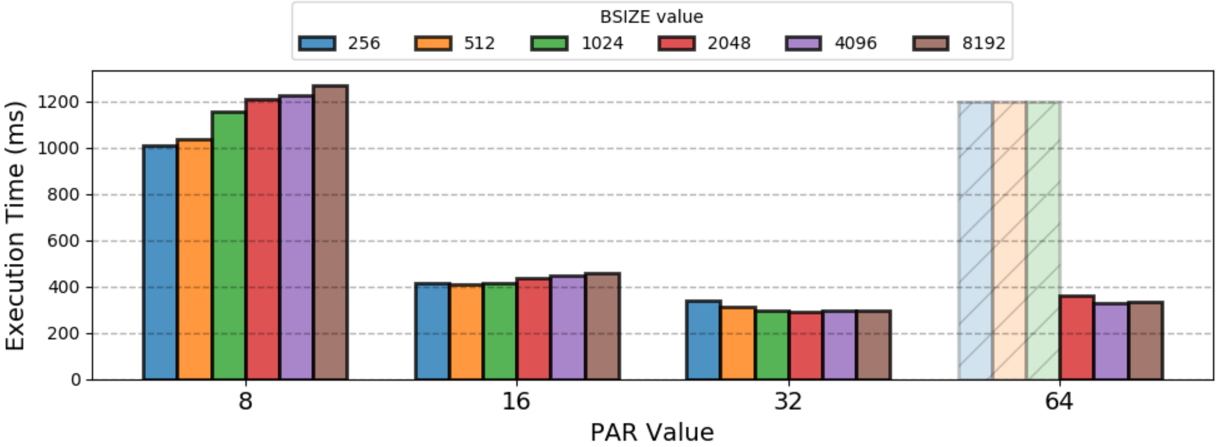


Figure 5.6: Graphical depiction of execution times for sweeping across hardware parameters `BSIZE` and `PAR` in kernel Version 5. The bars with greyed-out and diagonal lines represent parameter configurations for designs that were unable to be fitted for the FPGA.

5.2.2 Hardware Design Space Search

Table 5.2 shows the results for sweeping `BSIZE` for kernel Versions 0 and 2, as well as the results for sweeping `BSIZE` and `PAR` for kernel Version 5. As in the previous section, the execution times presented are the lowest times over 100 runs of the respective kernels.

In our experimentation, we define the kernel version design search space for kernel Version 0 as

$$BSIZE = \{64, 128, 256\}.$$

For kernel Version 2, it is

$$BSIZE = \{8, 16\}.$$

For kernel Version 5, the search space is the Cartesian product between

$$BSIZE = \{256, 512, 1024, 2048, 4096, 8192\}$$

and

$$PAR = \{8, 16, 32, 64\}.$$

In the case of kernel Versions 0 and 2, the upper bound of the search space was determined by the largest value that `BSIZE` could assume while still being synthesizable by the Intel FPGA OpenCL SDK offline compiler. The upper bounds for kernel Version 5 were determined by the amount of time required to synthesize a design. It must be noted, though, that for `BSIZE = 256, 512, 1024` and `PAR = 64`, the compiler was not able to synthesize a design. Investigating the logs revealed that despite multiple attempts at fitting the design, the routing was too congested and the fitting phase ultimately failed. Slightly larger designs fit on the FPGA, i.e. kernels with `BSIZE = 2048, 4096, 8192`, so we attribute these failures to shortcomings with the offline compiler.

The optimal settings for `BSIZE` in kernel Versions 0 and 2 reported in [148] were 128 and 64, respectively, for the Stratix V FPGA. The optimal HARPv2 `BSIZE` for Versions 0 and 2 were found to be 128 and 8, respectively. Thus, the setting matches the optimal setting in [148] for Version 0 but not for 2. The design space for Version 2 on HARPv2 did not include the optimal setting from [148], yet it outperformed [148] by a factor of 1.57 and with a smaller `BSIZE`. For kernel Version 5, `BSIZE` and `PAR` are set to 4096 and 64, respectively, for both the Stratix V and the Arria 10 FPGA to achieve optimal performance in [148]. However, the configuration that was optimal in [148] was not the most performant configuration for the HARPv2 system; the result in [148] is 1.64 times faster. While we expect the best configuration not to align for different FPGAs, this speaks to the portability of kernels designed for FPGAs connected through a PCIe slot versus the HARPv2 system even when the FPGAs are the same. Some of this performance difference can be attributed to the large amount of resources used to implement the CPU/FPGA interface as previously discussed. While the performance difference is relatively small, this result also suggests that further consideration must be given when authoring kernels specific to the HARPv2. This is similar to the claim that OpenCL kernels intended for one type of accelerator will not be the most performant for another type made in [149] when describing GPUs and FPGAs.

The build times of the different configurations of the kernels are also shown in Table 5.2. Perhaps the most startling result is the amount of time spent buiding kernels for Version 5. The longest build time was for `BSIZE = 4096` and `PAR = 64`, which took nearly two days. In total, it took 14 days to build all the kernels in order to search the design space and find the most optimal kernel. The amount of time it takes to search the design space by brute force necessitates the need for performance models, using facets of the kernel and its estimated resources as inputs, that can be more intelligently searched. To this end, work done by Wang et al. has demonstrated progress in this area by modeling OpenCL workloads on FPGAs for the NDR model [134]. Additionally, it would be beneficial to isolate the parameters of such analytic models which affect performance the most in order to prune the search space of lower weight parameters. Consider Figure 5.6, which graphically shows the execution times for all the configurations of kernel Version 5.

When `PAR` is small (i.e., `PAR` = 8), execution time increases as `BSIZE` increases because the effects of the inefficient staircase register allocation outweighs the benefits of processing a small number of the substitution matrix in a pipeline-parallel fashion. However, as `PAR` grows, the effects of processing more and more columns in parallel has a greater impact on performance than sweeping the `BSIZE` parameter. Holding `BSIZE` to some constant and sweeping the `PAR` parameter, which has only 4 discrete values, would lead to finding the optimal setting of 32 for PAR. In this case, the range of execution times for the different values of BSIZE is 46 *ms* at 4 days of kernel build time, while the global range is 978 *ms* at 14 days. This then becomes a tradeoff between an approximate answer found quickly versus a precise answer found slowly.

5.2.3 SVM Performance

Since the kernel built for the runs with explicit reads and writes is the same one used for the runs using SVM, the FPGA resource utilization remains the same between the two. Figure 5.7 shows the benefit in modifying the host code to use the SVM abstraction, as described in Section 5.1.4, for kernel Version 5 at the best performing parameter configuration for the HARPv2 system: BSIZE = 2048 and PAR = 32. The execution time reported is the smallest out of 100 runs.

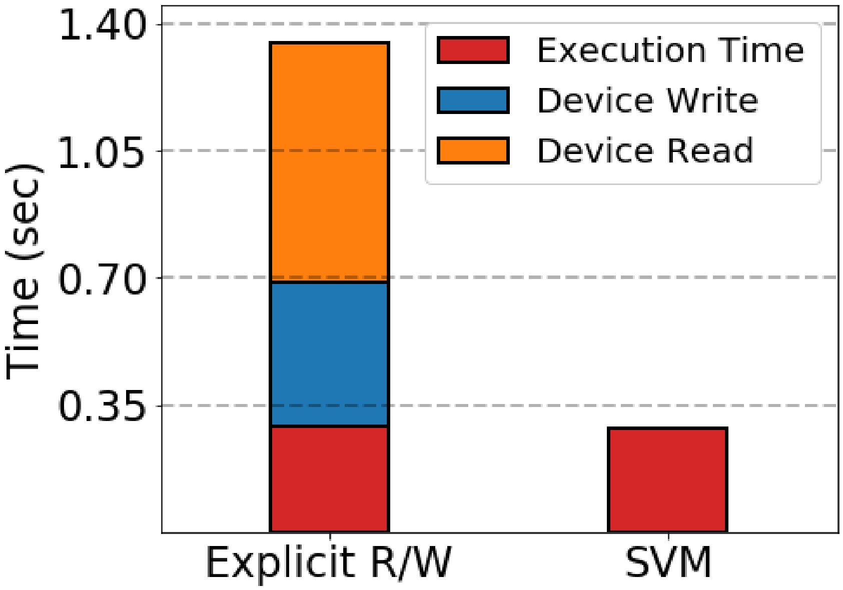


Figure 5.7: Execution times for kernel Version 5 with BSIZE = 2048 and PAR = 32 for host code that enqueues reads and writes explicitly to the device and host code that uses the SVM abstraction.

The left bar shows total amount of time the kernel took to execute, as well as the explicit reads and writes to global memory. Both the explicit reads and writes take longer than the execution of the kernel and increase the running time by a factor of 4. The right bar shows the execution time using the modified host code that uses the SVM abstraction. The time taken to allocate shared buffers was also recorded, but takes 10s of milliseconds and is negligible relative to the execution time. The time taken to explicitly read and write buffers was not recorded in [148], but conservatively assuming that explicit reads and writes execute in one-eighth the time that it does on the HARPv2 system would still have the HARPv2 outperforming the Arria 10 FPGA connected via PCIe slot.

This coherent, low-latency access to shared memory has important implications that require rethinking current paradigms of offloading computation to accelerators. Most commonly for compute-intensive tasks marked for accelerator offload, all data necessary for the computation is moved from from host to device. Since data movement is such an expensive operation, it is beneficial to perform as much computation as possible on the accelerator before shipping the results back to the host. This is the model used in all of the Needleman-Wunsch kernel versions in this work when using explicit reads and writes. The initial state for the substitution matrix and the entirety of the score matrix are moved to the FPGA. Once all of the substitution matrix has been computed, the updated substitution matrix is moved back to the host for reading.

The tighter integration present in the HARPv2 system, however, would allow for more fine-grained interactions between host and device without the overhead currently of explicitly moving data from CPU to FPGA memory. Huang et al. have investigated the tradeoffs associated in partitioning tasks and data in heterogeneous systems that collaboratively use CPUs and FPGAs attached via a PCIe bus [59]. They find that both partitioning schemes improve the execution time over systems that do not use any kind of collaborative execution.

Future OpenCL FPGA kernels targeting the HARPv2 system, then, should take advantage of the low latency communication between shared memory and the FPGA. Additionally, application designers should find ways to collaboratively use the CPU and FPGA for a computation region of interest, instead of relying on one or the other to perform the entirety of that region.

5.3 Conclusion

FPGAs offer a heterogenous compute solution to the problem of diminishing returns and physical limits of transistor scaling by enabling the creation of application-specific hardware that accelerates computation. While the barrier to entry has historically been steep, advances in High Level Synthesis (HLS) are making FPGAs more accessible. Specifically, the Intel FPGA OpenCL SDK allows software designers to abstract away low level details of architecting hardware on an FPGA and allows them to author computational kernels in higher level languages. Furthermore, Intel has developed a system that incorporates both a multicore Xeon CPU and Arria 10 FPGA into the same chip package, as part of the Heterogeneous Accelerator Research Program (HARP), that can be targeted by their SDK.

In this work, we targeted the second iteration of the HARP platform (HARPv2) using HLS through porting OpenCL kernels written for FPGAs connected via PCIe card. We evaluate their performance against previously reported results, explore the portability of kernels intended for PCIe-connected FPGAs through a hardware design space search, and empirically show the benefits of using the SVM abstraction over explicit reads and writes to the FPGA. Additionally, all artifacts associated with this chapter (code and data) are available through WashU OpenScholarship [18].

Having completed this evaluation of the Intel HARPv2 system, we take the lessons learned and apply them to the design of domain specific hardware in Chapter 6.

Chapter 6

Designing Domain Specific Compute Systems

As mentioned in Chapter 1, John Hennessey and David Patterson use the slowing of Moore's law to signal a new "golden age of computer architecture" and suggested that the path to a post-Moore's law world is paved, in part, by domain specific computing. This key idea means less emphasis on the paradigm of improving general purpose processors and more towards hardware and surrounding infrastructure for processors that focus on a class or domain of applications in a high-performing manner. The hardware flexibility of FPGAs, paired with incrementally easier programmability through HLS, can be used to realize the vision of post-Moore systems that incorporate heterogeneous compute components.

At present, domains are comprised of applications that align with the dictionary definition of the area, e.g., support vector machines and convolutional neural networks are types of machine learning applications so they fall under the domain of machine learning. While we gleaned valuable insights from our definition and characterization of the domain of data integration, we aim to further our understanding of the domain by making quantitative design choices.

Specifically, the quantitative decisions we want to make are about designing domain specific hardware. The question is:

How do you architect hardware for a domain?

In the previous chapter, we made our initial evaluation of using OpenCL to design hardware for the HARPv2 system. OpenCL enables the design of hardware at a much higher level of abstraction than RTL, but this increase in programmability is not without its own challenges. While FPGA hardware can be described using a higher level of abstraction, it can be unclear what hardware results from a specified kernel of computation. Often, the inclusion or exclusion of one line or even a keyword can imply a non-trivial amount of hardware and can have a large impact on the design that is inferred. In particular, we learned in Chapter 5 how impactful the execution model and the tuning of hardware parameters can be towards performance.

In this chapter, we take the lessons learned from Chapters 3, 4, and 5 in order to refine our knowledge of the data integration domain and architect domain specific hardware. We apply our multi-spectral reuse distance technique to the DIBS applications in order to generate outputs that will serve as features to an unsupervised clustering technique. We then use these clusters to create sub-domains of our original domain that inform the hardware design choice of width versus depth, i.e., should a design be architected as a wide vectorized compute unit that executes multiple threads or a deeply pipelined compute unit that is controlled by a single thread? Each paradigm, additionally, comes with its own coarse-grained design knobs that are specific to that paradigm. Even when the best execution model is chosen, the knobs must be tuned for optimal performance along with other optimizations that may be applicable. We will show that, even for seemingly simple kernels, there are design choices and optimizations to be made whose interaction and performance are not immediately obvious.

The situation is, in fact, analogous to the need to optimize codes for good cache performance in the HPC community, which is primarily an empirical task [137], even today [77] Additionally, we present our methodology for overcoming limitations of the currently available tools for OpenCL kernel development on the platform and justifying design decisions through this methodology.

6.1 Methods

6.1.1 Clustering of Domain Applications

In our initial analysis of DIBS, we observed that data movement operations comprised a large fraction of the dynamic instruction mix. We will explore this notion further, and leverage multi-spectral reuse distance to quantitatively assess both the temporal and spatial locality of data references within the benchmark suite, and then use the outputs of this tool as features to an unsupervised learning technique.

Specifically, we use k-means clustering, with $k = 2$, to cluster the DIBS applications into two groups. The results are shown in Figure 6.1, where the two axes are the Earth Mover's Distance (EMD) measure [110] separating two different granularities of reuse distance. The x axis is EMD for 64-byte vs. 4-KiB granularity and the y axis represents 4-KiB vs. 2-MiB granularity. The resulting two clusters are shown as distinct colored points on the graph.

Based on this result, we see that there is a division of the applications within the originally specified domain of the benchmarking suite. We posit that these clusters might reasonably represent sub-domains of the initial data integration domain, and the cluster that a given

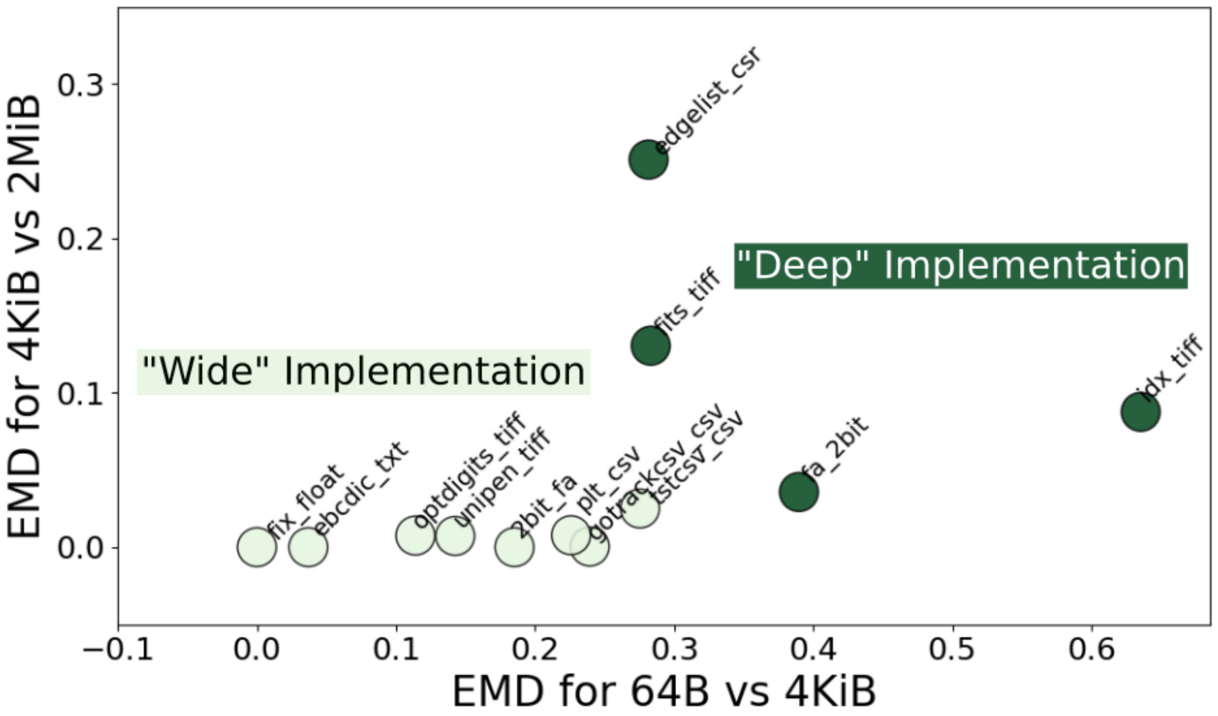


Figure 6.1: k-means clustering of the DIBS applications.

application is in will allow us to determine whether it will benefit from a wide or deep implementation.

6.1.2 Evaluating the Hardware

Once the sub-domain identification phase is complete, the target hardware platform must be selected, which, in our case, is the Intel HARPv2 CPU+FPGA system. The benefits of using this platform are threefold. The first is that we are able to take advantage of the reconfigurable nature of FPGAs. This functionality provides the basis of being able to hardware architect specific to the domain. The second is that the location of the FPGA on fabric alongside the Xeon cores allows for lower latencies and higher data transfer rates. This is beneficial because the workload characterization of DIBS shows a prevalence of data movement. Finally, the system can be targeted using the Intel FPGA OpenCL SDK. As opposed to using an HDL, this allows hardware designers to think about hardware design in a way that is semantically closer to the application, which lowers the technical barrier to using FPGAs. Moreover, it allows for an easier parameterization of the hardware design, enabling a more user-friendly way of tuning the design for optimal performance.

6.1.3 Kernel Development

When authoring FPGA designs using OpenCL, an important design choice is whether to architect a hardware kernel as a widely vectorized compute unit or deep pipeline, as shown in 6.2.

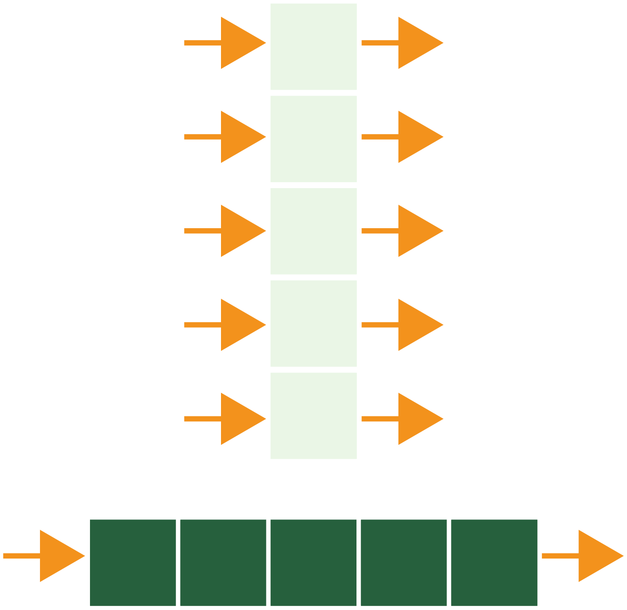


Figure 6.2: A block diagram showing (top) a design using the MWI execution model with multiple threads executing on multiple processing elements and (bottom) a deep pipeline orchestrated by one thread.

Insights and heuristics from the literature contend that the most performant design paradigm is to opt for the deeply pipelined approach. However, there are applications for which the

SIMD choice is more performant, e.g., a tiled matrix-multiply unit for convolutional neural networks. One of the intended contributions of this work is to be able to inform this decision based on the quantitative assessment from Section 6.1.1.

Specifically, we use the application groupings resulting from the k-means analysis to make this choice. Noting that applications with EMD scores closer to 0 exhibit a higher degree of spatial locality, we posit that the applications in the lighter colored group will benefit from a wide SIMD architecture. This is because spatially local memory accesses are easier to coalesce to take full advantage of widely vectorized architectures. In order to evaluate this hypothesis, we select three applications from each cluster.

6.1.4 Hardware Design Parameters

When designing OpenCL FPGA kernels, there are coarse-grained design knobs associated with each design paradigm. Each of these knobs has a set of assumable values that creates a hardware design space. The configuration of these knobs ultimately determines how much hardware is generated and how much of the FPGA's on-chip resources are utilized.

Deeply pipelined architectures are referred to as single-work item (SWI) kernels in the OpenCL literature. This is because the entirety of the computation is authored as a single-threaded task that is contained within a loop or set of loops. As such, the coarse-grained knob associated with SWI kernels is the unrolling of loops within the task, or the *loop unrolling factor*.

The loop unrolling factor allows for more iterations of a loop to be completed once the pipeline is fully saturated. Additionally, it gives the Intel FPGA OpenCL SDK compiler more opportunities to create optimizations between loop iterations.

Widely vectorized architectures are referred to as multiple-work item (MWI) kernels. This compute paradigm is more aligned with massively-threaded SIMD architectures like GPUs, in contrast to the single-threaded tasks in the SWI case. Kernels designed in this paradigm generally benefit from little to no dependencies between loop iterations, because synchronization can be costly. Additionally, irregular applications characterized by an abundance of conditional execution are also not suitable for this type of design paradigm.

In order to evaluate the performance between paradigms, we take the Cartesian product of all knobs, and synthesize hardware for each configuration. The description of the hardware design spaces of the kernels and their design are outlined in the following section.

6.2 Kernels

To determine the most performant execution model, we implement an MWI and SWI version for our chosen subset of the DIBS applications and perform a design space search using the coarse-grained hardware knobs specific to each execution model. As stated in Chapter 3, most of our applications consist of a sequential loop over all of the data records of a given input and performing the integration task. This is reflected, as well, in the design of the MWI and SWI implementations of each kernel. Broadly, SWI kernels mostly resemble their sequential CPU implementation, since this OpenCL execution model relies on a single thread. MWI kernels differ slightly in that the for loop is removed in favor of multiple threads execute the original loop body and which sequential iteration it corresponds to is determined by the thread's local ID and the work-group to which it belongs.

When implementing OpenCL kernels for each of these applications, we needed to isolate which component of a given application was to be accelerated by the HARPv2 system.

The original ebcdic_txt, idx_tiff, and fix_float applications, by design, consisted of few tasks and so it was easy to isolate which component of those applications to accelerate. The applications edgelist_csr, fa_2bit, and 2bit_fa, were all applications taken from the literature for which we needed to isolate the data integration tasks from their respective original applications. However, the number of data integration tasks was more than the first three applications. In order to isolate which components to accelerate, we use the Linux perf utility to generate a function call graph with the percentage of all counted CPU cycles spent in a given function. The command to generate this call graph, in general, takes the following form:

```
perf record -F 1024 --call-graph dwarf -- <app binary> <app arg0> <...>
```

where the -F option specifies the frequency (in HZ), to record profiling information (number of CPU cycles by default), --call-graph dwarf indicates a call graph is to be generated by employing the DWARF debugging information format, <app binary> is the application binary, and <app arg0> <...> are the associated application arguments. For the latter three applications, we elaborate on which task is selected to get an OpenCL FPGA implementation in Sections 6.2.4, 6.2.5, and 6.2.6.

In the following subsections, we will describe the design coarse-grained knobs for each application for which we implemented FPGA designs. We describe the subset of all kernels architected, but use the ebcdic_txt application as an in-depth example.

```
 1  __attribute__((num_compute_units(NUMCOMPUNITS)))
 2  __attribute__((reqd_work_group_size(WGSIZE,1,1)))
 3  __attribute__((num_simd_work_items(NUMSIMD)))
 4  __kernel void
 5  k_e2a(     __global const uchar* restrict src,
 6             __global uchar* restrict dst) {
 7    unsigned char e2a_lut[256] =
 8    {
 9      0,  1,  2,  3,156,  9,134,127, /* e2a chars 0-7
         */
10      151,141,142, 11, 12, 13, 14, 15, /* 8-15 */
11      ...
12      48, 49, 50, 51, 52, 53, 54, 55, /* 240-247 */
13      56, 57,250,251,252,253,254,255 /* 248-255 */
14    };
15
16    unsigned int i = get_global_id(0);
17    uchar orig_char = src[i];
18    uchar xformd_char;
19
20    xformd_char = e2a_lut[orig_char];
21
22    dst[i] = xformd_char;
23  }
```

Listing 6.1: Baseline Implementation of the MWI `ebcdic_txt` kernel using the OpenCL API and syntax.

6.2.1 ebcdic_txt

The pseudocode for the MWI implementation of `ebcdic_txt` is shown in Listing 6.1, and largely follows the sequential implementation found in the original application.

The conversion is performed by using the EBCDIC character as an index (line 20, Listing 6.1) into a 256 character look up table (line 7-14, Listing 6.1) that maps the input EBCDIC character to the appropriate ASCII character.

Memory Access Hardware Compiler Hints

The `const` keyword is applied to the global input buffer `src` (line 5, Listing 6.1) to tell the hardware compiler that this buffer is read-only. The hardware compiler, in turn, will be given permission to perform more aggressive optimizations regarding loads from this buffer [60]. Both the `src` and `dst` (lines 5 and 6, Listing 6.1) global memory buffers are both preceded by the `restrict` keyword. This hints to the hardware compiler to "trust" the programmer's global memory accesses–this is a guarantee that there will be no pointer aliasing among these global buffers, and that there is no need to account for load and/or store dependencies between the buffers.

MWI Implementation

For the MWI model, there are three knobs: number of compute unit replicates ($NUMCOMPUNITS$), the required work-group size ($WGSIZE$, i.e., the number of local work items that will belong to a work-group), and the SIMD factor ($NUMSIMD$), i.e., how many times to replicate the data path). These knobs are set in lines 1-3 of Listing 6.1.

The design space for this kernel is shown in Equation 6.1.

$$\begin{gathered} WG = \{128, 256, 512, 1024\} \\ NCU = \{1, 2, 4, 8\} \\ NS = \{1, 2, 4, 8, 16\} \end{gathered} \tag{6.1}$$

We find that, generally, MWI kernels benefit mostly from increasing the knobs to their highest assumable values, which we will use as a design heuristic in Section 6.3.3. In particular, larger work-group sizes allow for work to be chunked in a spatially local way. Increasing $NUMCOMPUNITS$ and $NUMSIMD$ increases throughput by inferring multiple I/O interfaces and widening those interfaces, respectively. Additionally for the latter case, these wider interfaces allow for more data to be statically coalesced for access, which makes better use of the available bandwidth.

SWI Implementation

The SWI kernel code, shown in Listing 6.2, is similar to the MWI kernel, even though their execution models are orthogonal.

```
 1  __attribute__((max_global_work_dim(0)))
 2  __kernel void
 3  k_e2a(      __global const uchar* restrict src,
 4              __global uchar* restrict dst),
 5              unsigned int total_work_items) {
 6    unsigned char e2a_lut[256] = { ... }
 7    uchar orig_char, xformd_char;
 8    unsigned int i;
 9
10    #pragma unroll UNROLL
11    for (i = 0; i < total_work_items; ++i)
12    {
13      orig_char = src[i];
14      xformd_char = e2a_lut[orig_char];
15      dst[i] = xformd_char;
16    }
```

Listing 6.2: Implementation of SWI `ebcdic_txt` kernel.

One difference is the extra argument that tells the kernel how many times to perform the data transformation (`total_work_items` in line 6, Listing 6.2). All of the work to be executed is wrapped in a for loop whose exit is conditioned on `total_work_items`. Another difference is that there is only one coarse-grained knob associated with this execution model: the loop unroll factor for the for loop in line 11 of Listing 6.2. This is supplied as a compiler hint set by the tunable parameter `UNROLL` in line 10 of Listing 6.2.

The design space for this kernel is shown in Equation 6.2.

$$UNROLL = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\} \tag{6.2}$$

6.2.2 idx_tiff

The pseudocode for the MWI implementation of `idx_tiff` is shown in Algorithm 2. The MWI implementation of this kernel follows the same structure as the original sequential CPU implementation. The main difference is that we exploit the task parallelism inherent in the creation of each `TIFF` image; since the creation of one `TIFF` image does not depend on the creation of other TIFF images, we assign the creation of each image its own work-group so that it may be scheduled concurrently. The design space for this kernel is shown in Equation 6.3.

Algorithm 2: OpenCL kernel pseudocode for MWI `idx_tiff` implementation.

```
work-group threads read TIFF header data from input buffer and write to output buffer
work-group threads read IDX3 pixel data and write to output buffer
work-group threads read TIFF header data from input buffer and write to output buffer
```

$$\begin{aligned} WG &= \{64, 128, 256, 512\} \\ NCU &= \{1, 2, 4, 8\} \\ NS &= \{1, 2, 4, 8, 16\} \end{aligned} \tag{6.3}$$

The pseudocode for the SWI implementation is shown in Algorithm 3. The implementation is largely the same as the MWI implementation, except that the transformation of each image is handled by a single thread as opposed to multiple threads and work-groups, and the boundary condition `num_images` is passed as a kernel argument. The design space for this kernel is shown in Equation 6.4.

Algorithm 3: OpenCL kernel pseudocode for SWI `idx_tiff` implementation.

```
for $i \leftarrow 0$ to $num\_images$ do
    read TIFF header data from input buffer and write to output buffer
    read IDX3 pixel data and write to output buffer
    read TIFF header data from input buffer and write to output buffer
end for
```

$$UNROLL = \{1, 2, 4, 8, 16, 32, 64, 128, 256\} \tag{6.4}$$

6.2.3 fix_float

The pseudocode for the MWI implementation of `idx_tiff` is shown in Algorithm 4. The MWI implementation of this kernel follows the sequential CPU implementation, but takes advantage of the fact that the conversion of each fixed-point value does not depend on any other conversion. Thus, threads in any work-group can be scheduled to execute concurrently. The design space for this kernel is shown in Equation 6.5.

Algorithm 4: OpenCL kernel pseudocode for MWI `fix_float` implementation.

```
work-group threads **read** fixed point values from input
work-group threads **cast** fixed point values to float type
work-group threads **calculate** $1 << Qvalue$ to determine binary point and then cast to float type
work-group threads **divide** by fixed point value by binary point to get float type representation point and then cast to float type
work-group threads **write** converted data to output buffer
```

$$\begin{aligned} WG &= \{128, 256, 512, 1024\} \\ NCU &= \{1, 2, 4, 8\} \\ NS &= \{1, 2, 4, 8, 16\} \end{aligned} \tag{6.5}$$

The pseudocode for the SWI implementation is shown in Algorithm 5. The SWI implementation is the same as the MWI implementation, except that only one thread handles the entirety of the conversions, and the boundary condition `num_fix_vals` is passed as a kernel argument. The design space for this kernel is shown in Equation 6.6.

$$UNROLL = \{1, 2, 4, 8, 16, 32, 64, 128\}. \tag{6.6}$$

Algorithm 5: OpenCL kernel pseudocode for SWI `fix_float` implementation.

```
for $i \leftarrow 0$ to $num\_fix\_vals$ do
    read fixed point value from input
    cast fixed point value to float type
    calculate $1 << Q - value$ to determine binary point and then cast to float type
    divide by fixed point value by binary point to get float type representation point
    and then cast to float type
    write converted data to output buffer
end for
```

6.2.4 edgelist_csr

Upon profiling `edgelist_csr` with the Linux `perf` utility, we found that over 50% of the time spent generating the CSR representation from the input edgelist is spent in sorting the function sorting the adjacency lists for each vertex. The original downstream application for this conversion was breadth first search, and sorting each adjacency list is a precursor to deduplicating vertices since multiple edges to the same vertex is redundant for this application. Because the majority of time spent generating the CSR representation is spent sorting, we decide to focus on building sorting kernel. Gautier et al., as part of an OpenCL FPGA benchmarking suite called Spector [47], provide the OpenCL kernel source code for a merge sort implementation that we adapt for our data integration application. The MWI OpenCL kernel pseudocode for merge sort is shown in Algorithm 6. The design space for this kernel is shown in Equation 6.7.

Though we use the knob nomenclature from Spector, the knobs still largely match our methodology of defining a design space with coarse-grained knobs for a given execution model. The subset of the design space that we use is shown below.

Algorithm 6: OpenCL kernel pseudocode for MWI merge sort in `edgelist_csr` application.

```
work-group threads read chunks of local_sort_size into local memory
work-group threads sort chunks using mergesort
work-group threads write sorted chunks back to global memory
```

$$\begin{aligned} NUMWORKITEMS &= \{1, 2, 4, 8, 16\} \\ NUMWORKGROUPS &= \{1\} \\ NUMCOMPUTEUNITS &= \{1, 2\} \\ UNROLLLOCAL &= \{1, 2, 4, 8, 16\}. \end{aligned} \tag{6.7}$$

For this kernel, the total number of global work items is set to

$$n_{num\_work\_items} \times n_{num\_work\_groups}$$

where

$$n_{num\_work\_items} \in NUMWORKITEMS$$

and

$$n_{num\_work\_groups} \in NUMWORKGROUPS$$

.

Before this kernel, each of the prior kernels' total global work item size was a function of the input data size. In this case, it is solely a function of the knob configuration. The amount of work that each work item completes, however, is still a function of the input data size and the knob configuration. The knob $UNROLLLOCAL$ determines the loop unroll factor for how many concurrent reads of input data and writes of sorted data occur. $NUMWORKITEMS$ is the same as work-group size.

Algorithm 7: OpenCL kernel pseudocode for SWI merge sort in `edgelist_csr` application.

```
for $i \leftarrow 0$ to $num\_chunks$ do
    read chunk of local_sort_size into local memory
    sort chunk using mergesort
    write sorted chunk back to global memory
end for
```

For the SWI implementation, we restrict the number of work items, work groups, and compute units to 1 in order to force the synthesis of a single work item kernel. The pseudocode for this kernel is shown in Algorithm 7, and the boundary condition `num_chunks` is passed as a kernel argument. The resulting design space subset is shown in Equation 6.8.

$$UNROLLLOCAL = \{1, 2, 4, 8, 16\}. \tag{6.8}$$

6.2.5 2bit_fa

The Linux `perf` profiling results for this application revealed that 25.64% of the counted CPU cycles were spent in the function `toUpperN`, which takes a pointer to a `char` buffer of FASTA bases characters and the size of that buffer. The function iterates over the entire buffer and transforms each base to its upper-case representation if it is not already upper case. The actual upper-case transformation is provided by the `C` standard library. Examining this source code shows that the conversion is made by referencing a look-up table.

When designing the OpenCL kernel for this transformation, we instead take advantage of the fact that the difference between the upper and lower case values is equal to 32, which, in binary, can be added or removed by setting the 5^{th} bit in an n-bit binary value. To take advantage of this, we apply a mask of `0xDF` to each character to isolate the 5^{th} bit and remove it. If the character is lower case, the mask will un-set the 5^{th} bit. If the character is already upper-case, then the mask preserves the character's original value. The MWI OpenCL kernel pseudocode for merge sort is shown in Algorithm 8. The design space for this kernel is shown in Equation 6.9.

Algorithm 8: OpenCL kernel pseudocode for MWI upper case conversion in `2bit_fa` application.

```
initialize $mask \leftarrow$ 0xDF for each thread
work-group threads **read** FASTA characters
work-group threads **compute** FASTA $characters \land mask$
work-group threads **write** converted characters back to global memory
```

$$
\begin{aligned}
WG &= \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768\} \\
NCU &= \{1, 2, 4, 8\} \\
NS &= \{1, 2, 4, 8, 16\}
\end{aligned}
\tag{6.9}
$$

The SWI implementation is largely the same as the MWI implementation, except there is only one thread coordinating all of the upper-case transformations, and the boundary condition `seq_size` is passed as a kernel argument. The pseudocode for this kernel is shown in Algorithm 9 and the design space is shown in Equation 6.10.

$$
UNROLL = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048\}. \tag{6.10}
$$

Algorithm 9: OpenCL kernel pseudocode for SWI upper case conversion in `2bit_fa` application.

```
initialize $mask \leftarrow$ 0xDF for each thread
for $i \leftarrow 0$ to $seq\_size$ do
    read FASTA characters
    compute FASTA $character \wedge mask$
    write converted characters back to global memory
end for
```

6.2.6 fa_2bit

The Linux `perf` report for this application showed that approximately 44% the counted CPU cycles are split evenly between four, similar functions: The first two functions count the number of blocks of the character {`n,N`} and the number of blocks of lower case bases, respectively. The other two functions store the indices of of {`n,N`} and lower case blocks. We opt to accelerate one of the latter two functions: specifically, the function that stores the indices of lower case blocks. Though we present the implementation for only this function, similar kernels could be constructed for each of the other applications. We leave this to future work.

The pseudocode is for this implementation is shown in Equation 10. In the original, sequential implementation of the lower case block counting function, a variable is used to track whether or not the base from the previous loop iteration was lower case or not. If the base of the current loop iteration is also lower case, the current lower case block has not yet ended. If this is not the case, the current character is upper case, which marks the end of the lower case block, and its size and initial position is recorded. Because of the loop dependency between iterations, we use a map-reduce approach to parallelize the function. The total number of bases is divided by the work-group size of the kernel being run, e.g., a kernel built with a work-group size of 2048 means that there are $\frac{\texttt{total\_bases}}{2048}$ chunks of the original size, and thus $\frac{\texttt{total\_bases}}{2048}$ work-groups. Each work-group will locally track its positions of lower-case blocks. Once all work-groups have completed, a single-thread will combine all of the local information tracking the indices of the local lower case blocks with their respective sizes and write them into a global buffer.

Algorithm 10: OpenCL kernel pseudocode for MWI lower case block tracker in `fa_2bit` application.

```
work-group threads **track** lower case blocks in blk_size chunks by keeping track of the
  start and stop indices of consecutive lower case bases
one thread **reduces** chunks to track lower case blocks across entire input and writes
  results to output buffers
```

The design space for this kernel is shown in 6.11.

$$\begin{gathered} WG = \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144\} \\ NCU = \{1\} \\ NS = \{1\} \end{gathered} \tag{6.11}$$

The SWI implementation for this application is shown in 11. This implementation follows the implementation of the original, sequential implementation and is orchestrated by one thread. The boundary condition `seq_size` is passed as a kernel argument.

Algorithm 11: OpenCL kernel pseudocode for SWI lower case block tracker in `fa_2bit` application.

```
**for** $i \leftarrow 0$ to $seq\_size$ **do**
    **track** lower case blocks in entire input by recording the start and stop indices of
    consecutive lower case bases
**end for**
```

The design space is shown in Equation 6.12.

$$UNROLL = \{1, 2, 4, 8, 16\} \tag{6.12}$$

6.3 Other Design Considerations

In this section, we confirm the benefit of using the SVM abstraction used in Chapter 5 and how we can visualize OpenCL-level design choices, despite the available tools for OpenCL kernel development on the platform, in order to make smarter design choices. We also discuss the implications of further adding optimizations to a kernel for which the most performant execution has been found. Specifically, we evaluate the impact of vectorizing the `uchar` datatype in the MWI version of the `ebcdic_txt` application.

6.3.1 Overlapping Data Transfer and Execution

A key feature of the OpenCL environment specific to the Intel HARPv2 platform is that external memory is shared between the CPU and FPGA. This removes the problem of having to transfer data from host memory to FPGA memory and vice versa. This also allows for data transfer to directly overlap execution instead of waiting for explicit reads and writes between host and device memories. The interface to this memory is made available as an extension to the OpenCL 1.0 specification. Here, we allocate the `src` and `dst` buffers on the host side using the extension. Figure 6.3 shows the benefit of this method, which is congruent with related work [17].

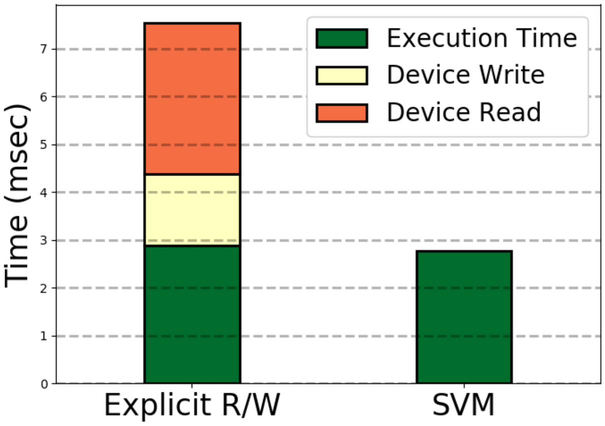


Figure 6.3: Execution times of explicit reads and writes to memory and using the OpenCL 1.0 SVM extension.

6.3.2 Visualizing the Hardware

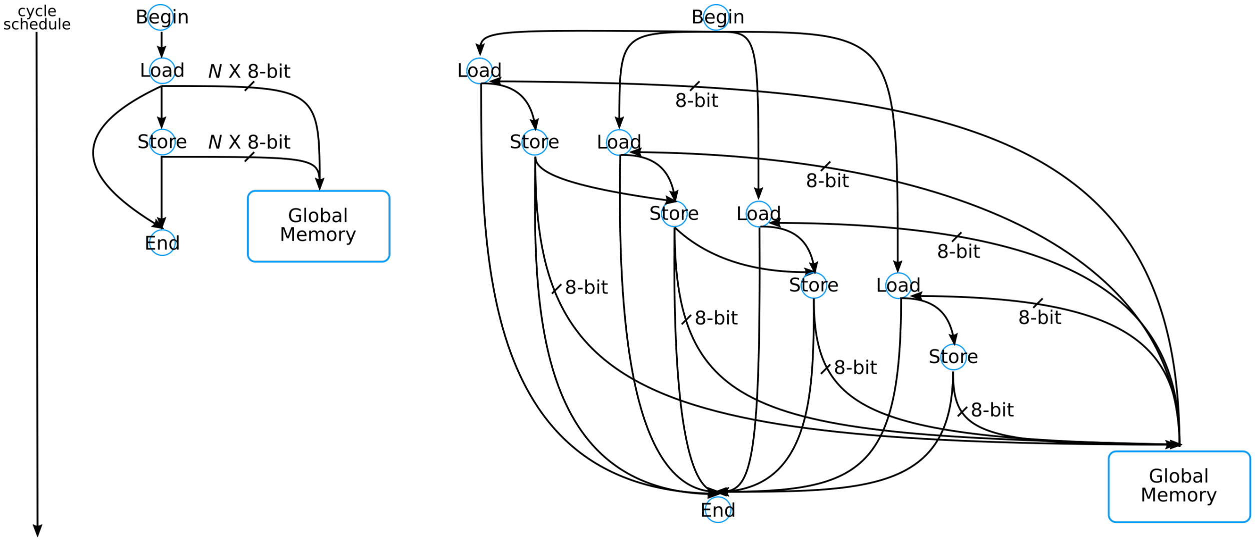


Figure 6.4: Approximate cycle schedule of the control flow data graphs (CDFGs) that represent unbounded (left) and bounded (right) versions of the `ebcdic_txt` kernel.

A challenge of using HLS to design hardware is the lack of ability to visualize what the hardware compiler will synthesize based on the OpenCL kernel that is authored. To this end, more recent versions of the Intel tools allow for an abstracted system level view of the hardware to be synthesized, by representing the operations to be executed as a control data flow diagram (CDFG) without having to fully synthesize a kernel. Historically in high level synthesis, viewing the abstracted hardware in this form is used to help reason about data dependencies and what cycle(s) to schedule operations on [93]. In this work, we will use this visualization as an aid to understand how a design choice made during the OpenCL kernel design process will impact the hardware that results. While this newer version of the tools is not supported by our target platform, we can still use them to effectively visualize design choices. This is valuable as the issue of tool versions is a general problem. We now show a use of this technique that allowed us to prune the design space and make an informed design decision by allowing us visualize a poor design choice made at the OpenCL kernel level and subsequently ignore it.

A requirement of MWI OpenCL kernels is that the work-group size evenly divides the number of global work-items (the total amount of work to be done). This is often not a naturally occurring feature when trying to accelerate applications. Consider, for example, that the optimal work-group size of the `ebcdic_txt` kernel was found to be 1024. Since the global work-item size is not a multiple of 1024, this requirement is not met. In order to address the "loose ends," a common solution is to inflate the global work-item size to satisfy the requirement. In our case, we could pad the input file sizes with `NULL` characters until the input size is a multiple of 1024 and modify the `ebcdic_txt` kernel to implement bounds checking to make sure that the kernel only processes meaningful input items. This is done by wrapping lines 16-22 of Listing 6.1 in an `if` statement conditioned on the true global work item size, as shown in Listing 6.3.

While this is a seemingly innocuous design choice for kernels targeting CPUs or GPUs with hardware support for conditional code, this is a costly operation when synthesizing hardware for the FPGA. Every operation (excluding dead code) specified in the kernel results in logic that gets synthesized into real hardware. The negative impact of this conditioned execution

```
...
  if (i < total_work_items)
  {
    unsigned int i = get_global_id(0);
    uchar orig_char = src[i];
    uchar xformd_char;

    xformd_char = e2a_lut[orig_char];

    dst[i] = xformd_char;
  }
```

Listing 6.3: Using bounds checking to avoid the “loose ends.”

on the hardware may not be immediately obvious, so we leverage the system level viewer of a more recent version of the Intel tools to help better understand the impact of this choice.

Figure 6.4 shows the system level view of two versions of the MWI `ebcdic_txt` kernel and the approximate cycle schedules of the CDFGs for an FPGA in the same product line (Arria 10) as our target platform. We will refer to the kernel version with no bounds checking as the unbounded case, and kernels with bounds checking as the bounded case. The left figure represents the CDFG and schedule for the unbounded case and the data path is replicated by a factor of `N` (i.e., $NUMSIMD = \texttt{N}$). It is also the same schedule for the bounded case, but only when $NUMSIMD = 1$. The right figure represents the bounded case where the data path is replicated 4 times.

The intuition behind this juxtaposition is that the schedule of the unbounded case does not change as the data path is replicated while the bounded case serializes accesses to global memory to maintain correctness. The total number of cycles for multiple replicas in the unbounded case scales well as $NUMSIMD$ increases because the hardware compiler is able to infer a wider I/O interface to global memory. The bounded case shows that each replica of the data path requires an additional serial global memory access, thereby increasing the cycle count when a work-item is scheduled. Thus, by performing this visualization, we opted to design the MWI kernel using the unbounded approach and take care of the remaining global work-items on the host side.

Going a step further, we observe the estimated resource utilization and work-item latencies for the unbounded and bounded cases when $WGSIZE = 512$, $NUMCOMPUNITS = 8$, and $NUMSIMD = 16$. We observe that the bounded case takes up at least 4× more resources than the unbounded case with respect to available look-up tables, flip-flops, and RAMs. In fact, the bounded case uses up 108% of the FPGA's RAMs, which would not be synthesizable. Additionally, we can see the effect of serialization in that the latency of the bounded case is $> 16\times$ larger than the unbounded case. Thus, by performing this visualization using a newer version of the tools, it allows us to ignore a design choice that would be blatantly detrimental to overall performance.

6.3.3 Widening the Data Type

We now build upon the baseline `ebcdic_txt` MWI kernel configuration established in Section 6.2.1. In this section, we detail an OpenCL design optimization to aid the hardware compiler in inferring even wider I/O interfaces and further statically coalescing memory accesses. This is accomplished by increasing the width of the data types in the `ebcdic_txt` kernel, we we do by leveraging the OpenCL specification for vectorized data types. Specifically, we can modify the `uchar` type to `uchar{2,4,8,16}`. The modified kernel version using `uchar4` is shown in Listing 6.4, where the `src`, `dst`, `orig_char`, and `xformd_char` variables all reflect the new data type. While the kernel description is unaffected by the data type vectorization, this optimization implicitly modifies the global work item size by a factor of the new data type width and effectively creates additional "loose ends." We must account for this in the host side code.

```
...
__kernel void
k_e2a(     __global const uchar4* restrict src,
           __global uchar4* restrict dst)
{
  unsigned char e2a_lut[256] = { ... };

  unsigned int i = get_global_id(0);
  uchar4 orig_char = src[i];
  uchar4 xformd_char;

  xformd_char.s0 = e2a_lut[orig_char.s0];
  xformd_char.s1 = e2a_lut[orig_char.s1];
  xformd_char.s2 = e2a_lut[orig_char.s2];
  xformd_char.s3 = e2a_lut[orig_char.s3];

  dst[i] = xformd_char;
}
```

Listing 6.4: Kernel with vectorized `uchar` types.

In order to understand the effects of this optimization as it interacts with the existing knobs of the baseline MWI kernel, we create versions of the kernel with each available widened data type and use a reduced design space as guided by the heuristic outline in Section 6.2.1. The new design space becomes

$$\begin{aligned} WG &= \{512, 1024\} \\ NCU &= \{1, 2, 4, 8\} \\ NS &= \{1, 2, 4, 16\} \end{aligned} \tag{6.13}$$

In this case, there are 32 unique configurations that we consider in order to evaluate this optimization.

Once the most performant kernel is found, we measure the impact of input scaling on this kernel. This is done by using the original file to create differently-sized versions (roughly powers of 2 in file size) up to 1 GB.

6.4 Results

The results of the design space search using all possible knob configurations outlined in Section 6.2 are presented in Section 6.4.1. Each application was run with 1GB of input data except for the `edgelist_csr` and `fa_2bit` applications; those applications used input sizes of 2MB and 256MB respectively. The former is due to limitations in the hardware compiler caused by the complexity of the merge sort kernel. While the applications produced functionally correct outputs using 1GB of input data during the hardware emulation phase, the actual hardware for that application was never able to finish. The latter is due to the MWI implementation requiring multiple, large memory allocations from the available 4GB of memory shared between the CPU and FPGA. An input size of 256MB was the largest possible input to stay within the shared memory budget.

The input data for each application remained the same for each application, i.e., each application used the same dataset and the size never varied. For 5 out of the 6 applications, this would cause minimal or no performance variation. However, the sort implemented in the `edgelist_csr` application is susceptible to the contents of the input edge list even if the size remains the same. Consider an edge list with n edges. If the origin for each edge in the edge list is the same for all n edges, only one sort needs to execute. However, if there are 2 origins, at least two instances of the sort need to execute. As the number of origins increases, so does the number of sorts that need to occur. Issuing multiple instances of the sort kernel to the command queue may incur non-trivial overheads as the number of items to be sorted in each kernel instance decreases. This, in turn, may result in performance degradation.

An overview of the most performant knob configuration for each execution model is shown in Table 6.2. When performance is mentioned, it is always assumed that the performance metric is data rate. Claims of "optimal" designs and performance are optimal for the design space that is defined in this work for a given kernel. In Section 6.4.2, we present the results of our performance predictions from Section 6.1.1. Finally, we present the results of widening the datatype in the MWI implementation of the `ebcdic_txt` application in Section 6.4.3.

6.4.1 Design Space Search Sweeps

SWI Design Space Search Results

In 4 of the 6 applications–`idx_tiff`, `fix_float`, `edgelist_csr`, and `2bit_fa`–display the same trend of a monotonically increasing and then decreasing data rate when increasing the amount that the computation loop is unrolled. The SWI sweep results for these applications are featured in Figures 6.5, 6.6, 6.7, and 6.8, respectively.

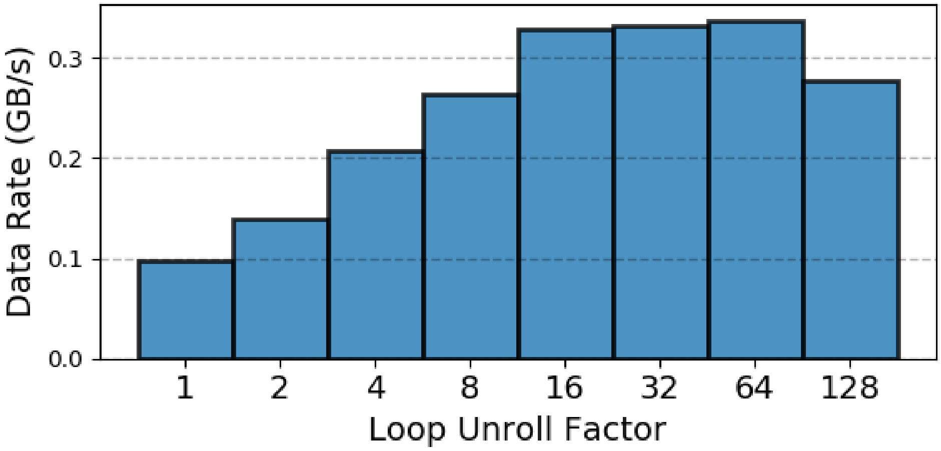


Figure 6.5: Result of sweeping across loop unrolling factor for `idx_tiff` application.

The cause of this result is similar to the effects of scaling knobs as described in Section 5.2.1. As the knobs are set to higher values, the complexity, area, and FPGA resource utilization of the hardware to be synthesized also increases. This, in turn, increases the difficulty in routing all of the logic elements and memories (and hardened floating-point units when applicable),

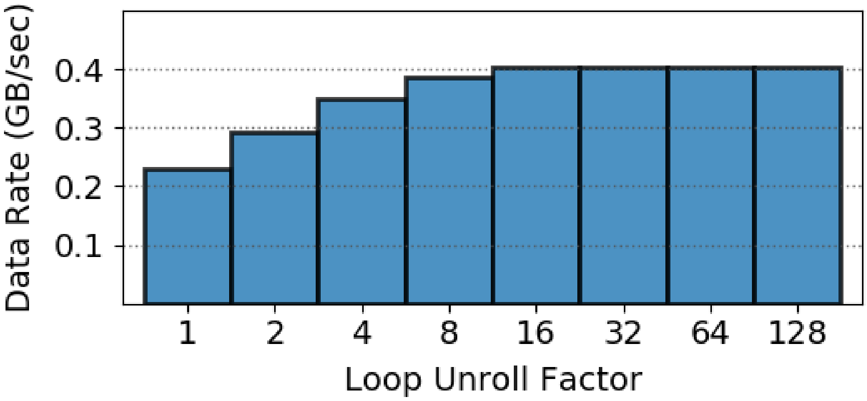


Figure 6.6: Result of sweeping across loop unrolling factor for `fix_float` application.

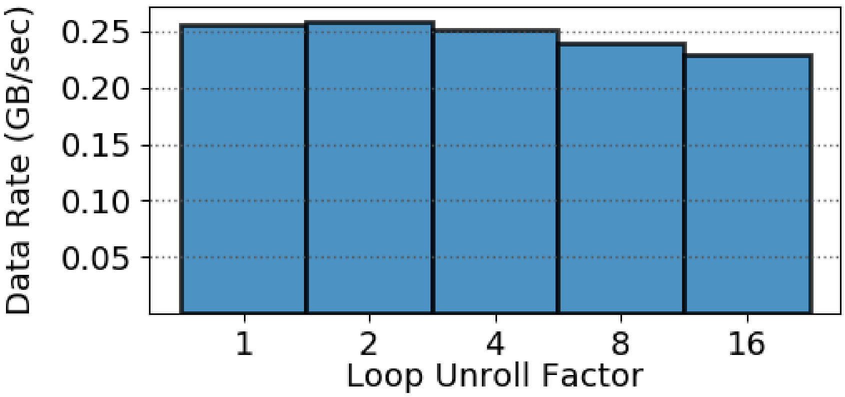


Figure 6.7: Result of sweeping across loop unrolling factor for `mergesort` implementation of the `edgelist_csr` application.

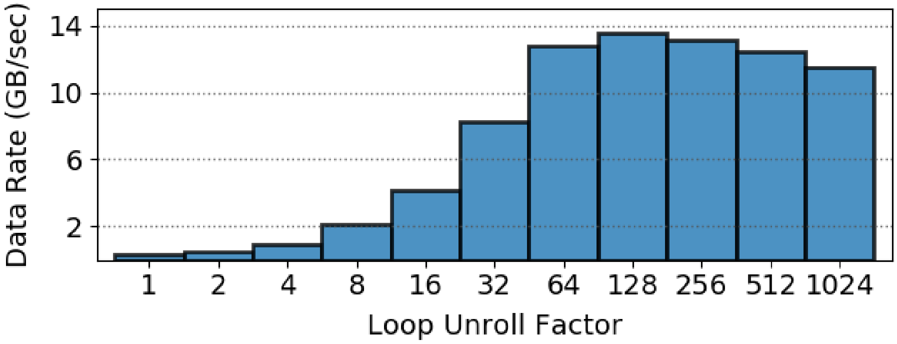


Figure 6.8: Result of sweeping across loop unrolling factor for `toUpper` implementation of the `2bit_fa` application.

and causes the hardware compiler to reduce the maximum clock frequency in order to meet timing constraints.

The hardware implementations for `ebcdic_txt` and `fa_2bit` each have different trends. The SWI design space sweep for these applications is shown in Figures 6.9 and 6.10, respectively.

In the case of `ebcdic_txt`, a loop unrolling factor of 1, i.e. no loop unrolling, is the second best performing SWI kernel for this application. From loop unrolling factors 2 through 1024, the performance is monotonically increasing through the rest of the design space. Intuitively, the monotonically increasing performance as the loop is unrolled is to be expected because unrolling the loop further exposes more parallelism, and there are no dependencies between loop iterations. This means the kernel iterations should be launched successively with no stalling. Additionally, the benefit of loop unrolling outweighs the effects of higher FPGA resource utilization and lower clock speeds, since the performance monotonically increases. However, the high performance with no loop unrolling is uninuitive. We attribute this to effects introduced by the hardware compiler. When the loop unrolling pragma is set to 1, it is effectively ignored by the hardware compiler. By not setting the loop unrolling pragma,

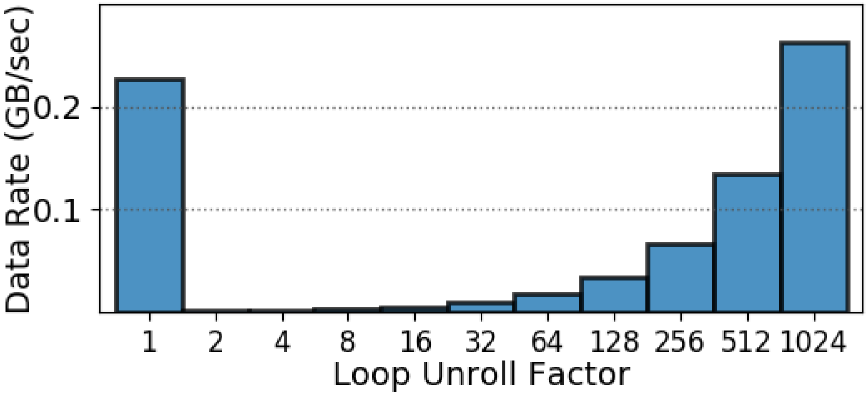


Figure 6.9: Result of sweeping across loop unrolling factor for `ebcdic_txt` application.

this, in turn, may allow the hardware compiler to try more aggressive optimizations that it would otherwise be limited when unrolling the loop.

For the `fa_2bit` implementation, the performance of each design as the loop unrolling factor is scaled up is monotonically decreasing. The performance of the implementation with no loop unrolling is 45× faster than the unrolling the loop by a factor 2. The partially unrolled loop has a data dependency between iterations; a write to memory in one iteration directly depends on a state variable written in the previous iteration. In the version with no loop unrolling, the hardware compiler is able to pipeline the original data path and still execute an iteration of the loop on every clock cycle. Once the loop becomes partially unrolled, however, the hardware compiler stalls the pipeline until the previous writes to the state variable have been resolved. Thus, successive iterations cannot be launched every cycle.

In general, then, we observe that the presence of data dependencies in a SWI kernel loop suggests that the loop will not benefit from being unrolled because the hardware compiler will introduce stalls between iterations in order to resolve the dependency; the performance without loop unrolling will be superior. However, when there is no data dependency present,

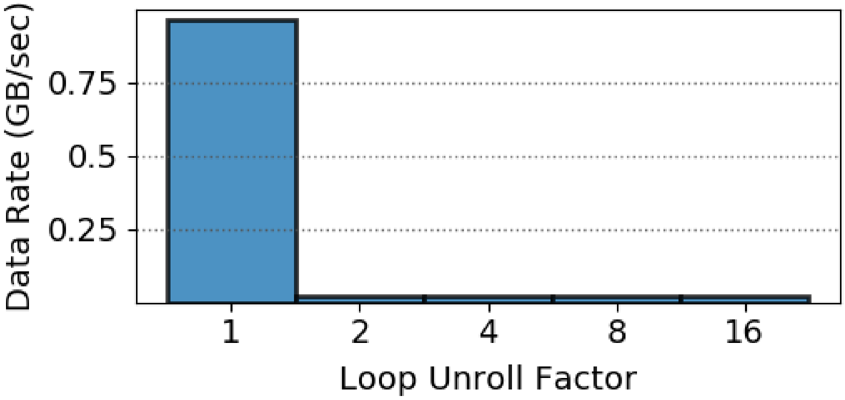


Figure 6.10: Result of sweeping across loop unrolling factor for `storeBlocksOfLower` implementation of the `fa_2bit` application.

there is a global maximum that exists in the search space, such that past a certain level of partial loop unrolling, the effects of exposing that iteration-level parallelism is outweighed by the amount of resources and routing complexity that arises when more operations required to realize that parallelism.

MWI Design Space Search Results

Figures 6.11, 6.12, 6.13, and 6.14, represent the MWI results for the MWI implementations of the `edgelist_csr`, `ebcdic_txt`, `fix_float`, and `2bit_fa` applications, respectively.

From these results, we observe that, in general, increasing the work group size of a kernel will increase the performance. Taking the average of the performance for the subset of the design space at each work-group size (e.g., the Cartesian product of all possible values of $NUMCOMPUNITS$ and $NUMSIMD$ when holding $WGSIZE$ fixed) for the aforementioned kernels results in better performance as $WGSIZE$ increases. The average data rates of these kernels for fixed work group sizes are shown in Table 6.1.

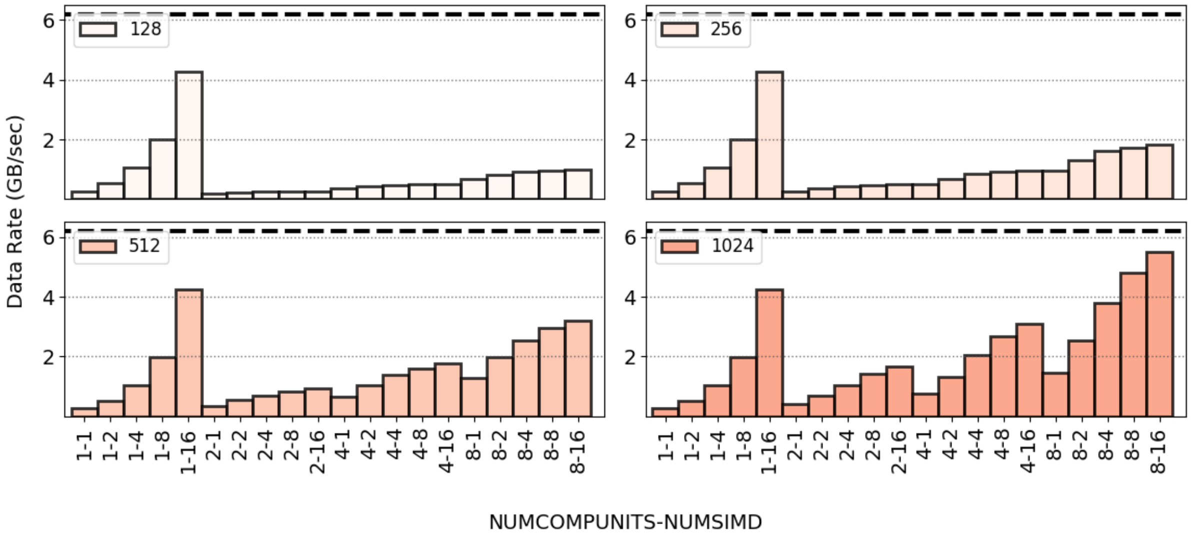


Figure 6.11: Result of sweeping across all possible of configurations within the `ebcdic_txt` MWI hardware design space. The set of possible WG, NCU and NS combinations are evaluated.

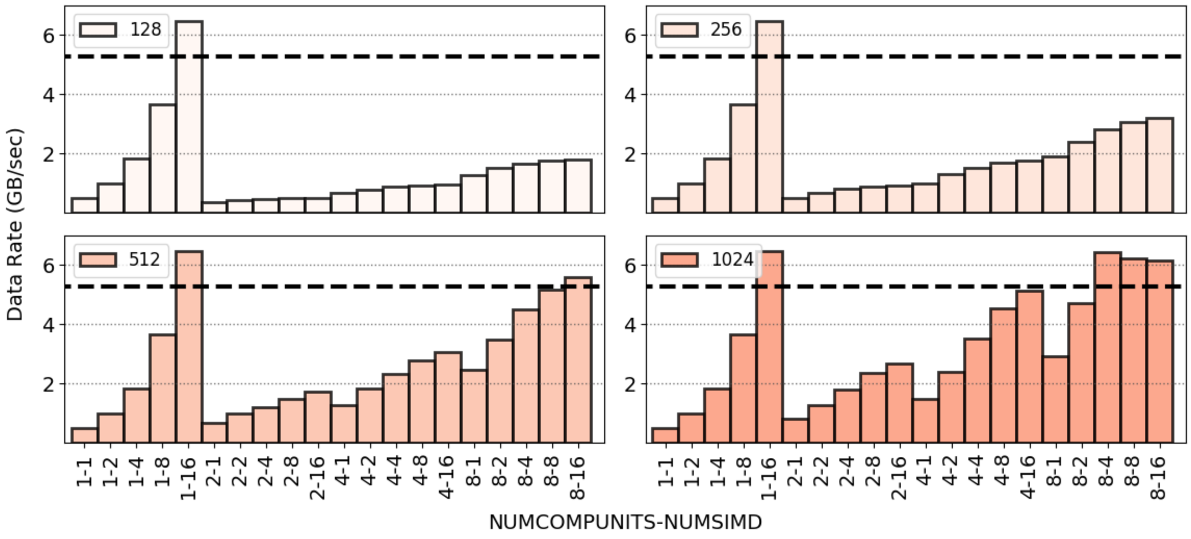


Figure 6.12: Result of sweeping across all possible of configurations within the `fix_float` MWI hardware design space.

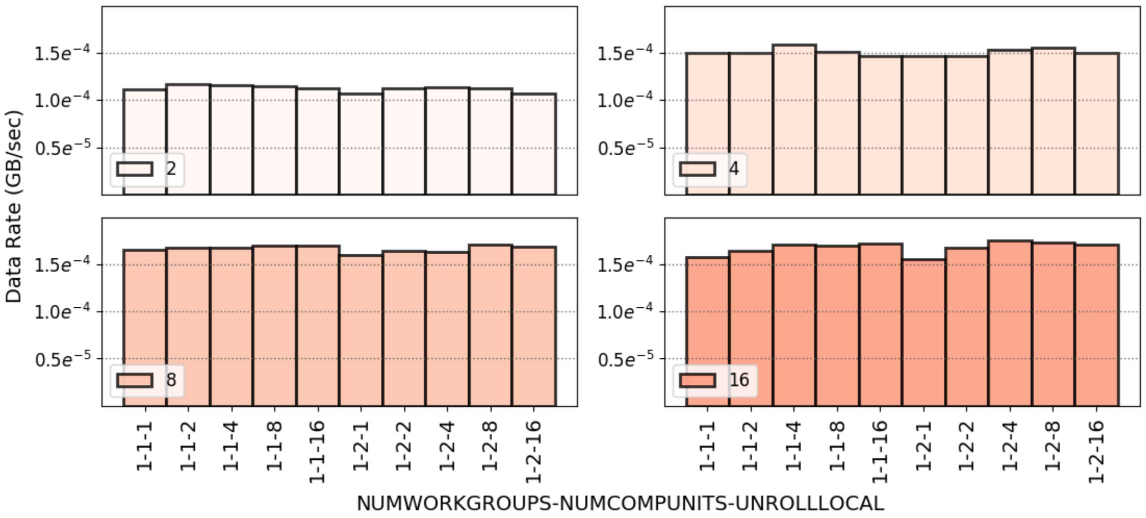


Figure 6.13: Result of sweeping across all possible of configurations within the `mergesort` implementation of the `edgelist_csr` application.

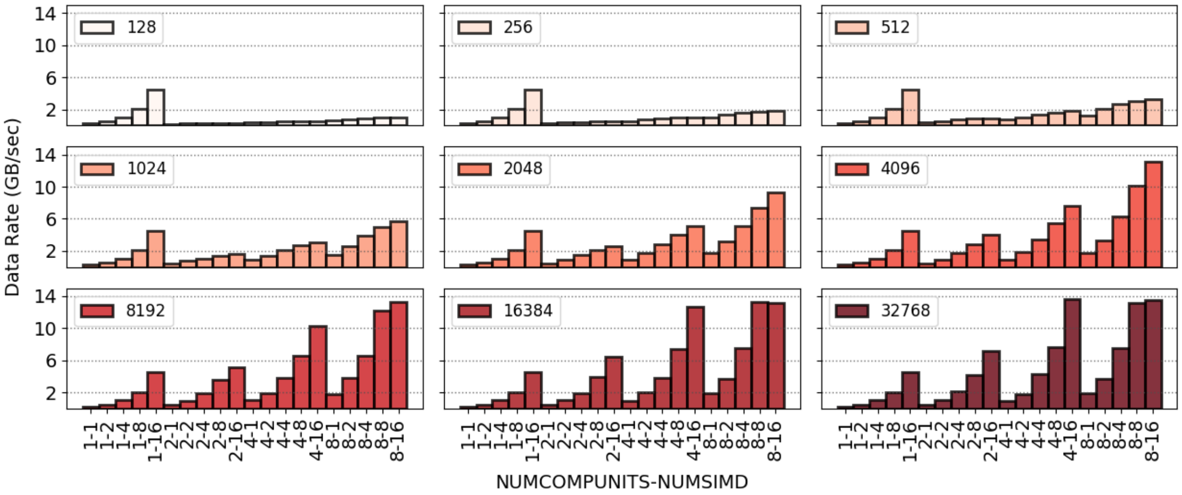


Figure 6.14: Result of sweeping across all possible of configurations within the `toUpper` implementation of the `2bit_fa` application.

Application	Work-Group Size	Data Rate (GB/s)
`ebcdic_txt`	128	0.430
	256	0.651
	512	0.886
	1024	1.076
`fix_float`	128	0.799
	256	1.205
	512	1.610
	512	1.943
`edgelist_csr`	128	$1.123e^{-4}$
	256	$1.511e^{-4}$
	512	$1.669e^{-4}$
	512	$1.677e^{-4}$
`2bit_fa`	128	0.434
	256	0.664
	512	0.898
	1024	1.114
	2048	1.242
	4096	1.333
	8192	1.373
	16384	1.406
	32768	1.418

Table 6.1: Average data rates of the MWI kernel for which there is a positive relationship between work-group size and performance.

While the performance is not monotonically increasing in `idx_tiff` and `fig:fa_2bit`, the results for which are shown in Figures 6.15 and 6.16, respectively, the most performant versions of these kernels occur when the work-group size is of the respective kernels are set to their maximum values.

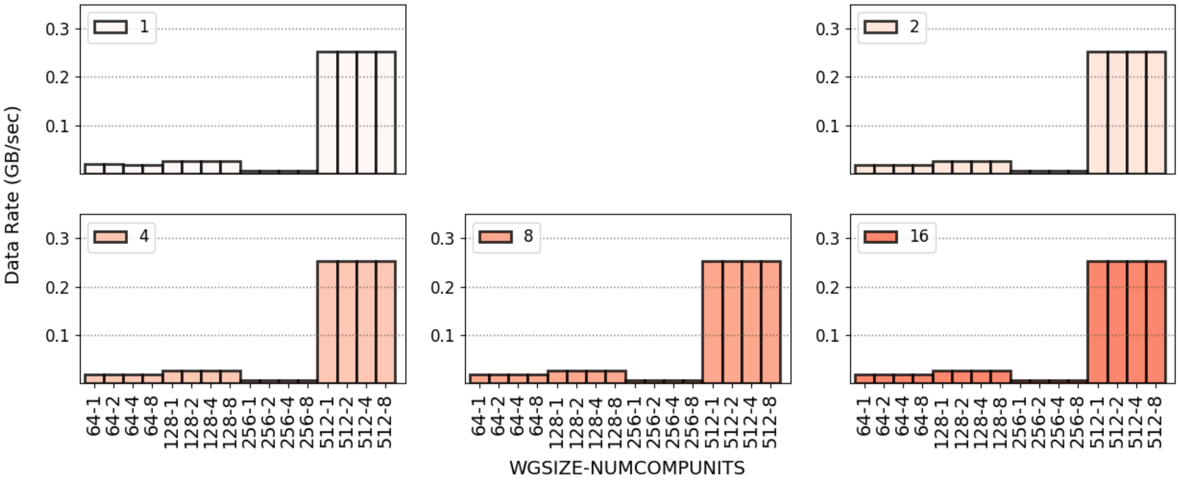


Figure 6.15: Result of sweeping across all possible of configurations within the `idx_tiff` MWI hardware design space. The set of possible NS, WG and NCU combinations are evaluated.

And while there are exceptions to this in our empirical results, another general trend observable in our results is that a kernel's performance at the smallest assumable value for the SIMD datapath replication factor, i.e., 1, the performance monotonically increases as the factor is increased. A similar trend exists for the number of compute units.

In Section 5.2.2, we noted how long the kernel synthesis process was for the Needleman Wunsch kernels, and performing the same kind of design space search here is no exception, which necessitates strategies for efficiently searching and intelligently pruning the design space. Towards the latter, it may be beneficial to restrict the kernels built to a subset with the larger of the assumable knob values. Using our empirical results as a test set, we would

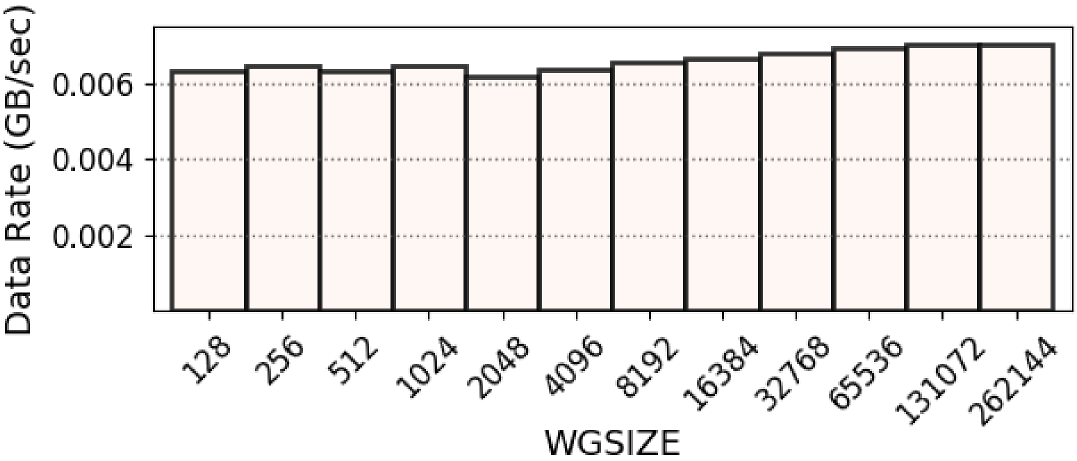


Figure 6.16: Result of sweeping across all possible of configurations within the `storeBlocksOfLower` implementation of the `fa_2bit` application.

find the optimal configurations for 3 of the 6 applications, and never be more than 5.54% away from the optimal solution (assuming the configurations where all of the knobs take their maximum values. As mentioned previously in 5.2.2, pruning the design space in this way is a trade-off–finding the true optimal configuration at the expense of time or finding a near-optimal kernel that sacrifices optimality–that must be made by the kernel designer.

The `idx_tiff` application is unaffected by the number of SIMD datapath replication and compute units. Upon further inspection of logs generated by the hardware compiler, we found that the hardware compiler could not replicate the data path and, at most, only two work-groups would ever run concurrently. Intuitively, this application should be able to take advantage of multiple compute units and replicated data paths. Future work would involve redesigning this kernel in a way that allows the computation to take advantage of multiple processing elements.

Interestingly, the optimal configuration for the `fix_float` replicates the data path 16 times, but has a compute unit replication factor of 1, i.e., no replication. This is attributed to the non-spatially local accesses that may arise depending on which work-groups are concurrently executing. Specifically, one work-group may be accessing one region of memory while the other $N-1$ work-groups currently executing are all reading/writing memory in very different regions of memory. It should be noted, though, that the replicated data path does aid in spatially local memory access because the compiler can coalesce reads and writes for a given compute unit. Additionally, we observe from the kernel build logs that the hardened DSP blocks are being utilized in each of the kernels (because the percent DSP block utilization is non-zero). We assume that each compute unit gets its own 32-bit floating point divide unit (because the percent DSP block utilization increases as the number of compute units increases), i.e., not shared among compute units, and that those units are also replicated in the data path (because of increased DSP block utilization when the SIMD factor increases) such that operations can happen concurrently among and within compute units. Still, one compute unit is more performant for this particular kernel. Another benefit to using one compute unit is that it is much easier for the hardware scheduler to schedule a work-group for execution if there is only compute unit for which to execute work-groups.

In Figures 6.6 and 6.9, the black dotted lines represent the comparison to the most performant multithreaded CPU implementation for that kernel using OpenCL. For `fix_float`, there are 8 designs that achieve better performance than its multi-threaded CPU counterpart, but no `ebcdic_txt` designs achieve better performance. We explore this further in Section 6.4.3, as we build upon the result of finding the most performant execution model. Future work would include finding the knob configurations of the CPU OpenCL kernels and comparing their performance to their FPGA-accelerated versions. However, the main thrust of this thesis is to understand how to make the right execution model choice.

6.4.2 MWI versus SWI Implementations

In ebcdic_txt, idx_tiff, fix_float, edgelist_csr, and 2bit_fa, all of the computation that was handled by multiple work-groups and threads in the MWI implementation is handled by a single thread in the SWI implementation. Effectively, a single *for* loop in the SWI version of a given kernel replaces the global/local/work-group indexing required to correctly execute computation in the MWI model. In this case, when a SWI kernel performs better than an MWI kernel, that means that the kernel benefits more from unrolling the loop of computation and pipelining that unrolled loop as opposed to using multiple processing elements to complete that same computation. The fact that this is not immediately obvious just by looking at the OpenCL kernels themselves accentuates the necessity for this approach, and more generally, the quantification of application domains to enable this kind of experimentation.

In fa_2bit, the MWI and SWI OpenCL implementations do not share the relationship outlined above. Recalling from Section 6.2.6, the SWI implementation follows the original sequential CPU implementation, while the MWI version uses a map reduce approach by dividing all of the bases into sub-problems, and finding lower case blocks in those sub-problems and then using a single thread to combine all of the sub-problems. Because of this difference, the CPU kernel that was profiled using multi-spectral reuse distance may not accurately reflect the new implementation implemented in the MWI case. Additionally, the reduce step of the map-reduce step is usually handled by a single processing element; it might have been beneficial to split the map portion of the computation into a MWI kernel and the reduce portion into a SWI kernel. Furthermore, the reduce step could be more performant on the host side instead of on the accelerator side. Future work, then, would examine what to do in the instances where the MWI implementation differs from the SWI implementation more than outlined in the previous paragraph. Additionally, quantitaively determining what parts of a kernel or kernels would be beneficial on a particular platform, e.g., a CPU, GPU, or FPGA. This could be done similarly to the k-means analysis and using more features to better answer those questions.

The results of the most performant execution model for each application are shown in Table 6.2, and a comparison of the most performant data rate for the different execution models compared to the sequential CPU data rates are shown in Figure 6.17.

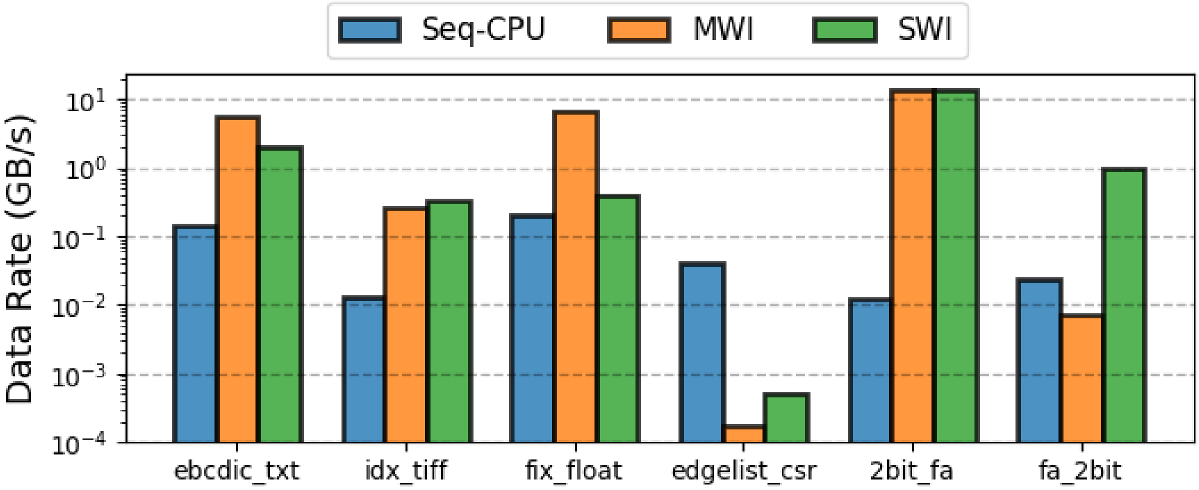


Figure 6.17: Comparison between the most performant configurations of each execution model and their sequential CPU data rate.

Our prediction of the most performant execution model using the k-means clustering of the DIBS applications proved to be correct in each application for which we implemented FPGA hardware. The correctness is, in part, attributed to how prevalent data movement, and therefore locality, is in the DIBS applications. Recall that in 3.5.3, the dynamic instruction mix in 10 of the 12 DIBS applications were comprised of 50% data movement instructions. Additionally, the MWI and SWI execution models are inherently affected by data movement. Applications that are more performant as MWI implementations benefit from contiguous and spatially local accesses in order to fully utilize all of the available compute units that become

Application	Execution Model	# of Designs	Knob Configuration	Logic	M20K Bits	M20K Blocks	Data Rate (GB/s)
`ebcdic_txt`	MWI	80	$WGSIZE = 1024$ $NUMCOMPUNITS = 8$ $NUMSIMD = 16$	28%	8%	23%	**5.49**
	SWI	11	$UNROLL = 1024$	43%	35%	78%	0.26
`idx_tiff`	MWI	80	$WGSIZE = 512$ $NUMCOMPUNITS = 8$ $NUMSIMD = 16$	26%	8%	17%	0.25
	SWI	9	$UNROLL = 64$	32%	12%	23%	**0.34**
`fix_float`	MWI	80	$WGSIZE = 128$ $NUMCOMPUNITS = 1$ $NUMSIMD = 16$	25%	6%	15%	**6.48**
	SWI	8	$UNROLL = 64$	31%	10%	18%	0.40
`edgelist_csr`	MWI	50	$NUMWORKITEMS = 16$ $NUMWORKGROUPS = 1$ $NUMCOMPUNITS = 2$ $UNROLLLOCAL = 4$	39%	22%	37%	$1.75e^{-3}$
	SWI	5	$UNROLLLOCAL = 2$	29%	11%	21%	$\mathbf{5.06e^{-3}}$
`2bit_fa`	MWI	180	$WGSIZE = 32768$ $NUMCOMPUNITS = 4$ $NUMSIMD = 16$	26%	7%	16%	**13.61**
	SWI	12	$UNROLL = 128$	25%	9%	17%	13.58
`fa_2bit`	MWI	12	$WGSIZE = 262144$ $NUMCOMPUNITS = 1$ $NUMSIMD = 1$	32%	49%	68%	0.007
	SWI	5	$UNROLL = 1$	25%	7%	15%	**0.97**

Table 6.2: Results for the most performant configuration of each execution model for each application.

available through compute unit and data path replication. Memory accesses that are not spatially local, then, adversely affect performance because the compute pipeline must stall in order to read/write the necessary data from/to global memory. Orthogonally, one of the benefits of the SWI model is that the hardware compiler can deepen the pipeline and account for irregular memory accesses while still launching successive iterations of the loop on every clock cycle. In the MWI model, the hardware compiler does not allow compute pipelines to launch successive iterations in the presence of loops.

When looking at the predictions made in Figure 6.1, we see that clusters are effectively dividing the DIBS application into two different regions of spatial locality, recalling that applications that are closer to the origin point are more spatially local than those that aren't. The applications that were predicted to be more performant as widely vectorized (MWI) kernels are in the region with more spatially local accesses. The applications predicted to be more performant as deeply pipelined applications are less spatially local. Computationally, these kernels are not compute intensive. Thus, using metrics that measure data movement to predict the most performant execution model was a successful method for making this design choice, for this particular domain. However, when generalizing the use of multi-spectral reuse distance and EMD, it should be noted that if the spatial locality is large enough, a spatially un-local application may "alias" as a spatially local one. This would be the case if the reuse distance granularities of a given application contain significant mass at bin sizes away from the origin and the EMD scores are still low. This was not a problem in our domain of choice, but is worth noting when extending this approach to other application domains.

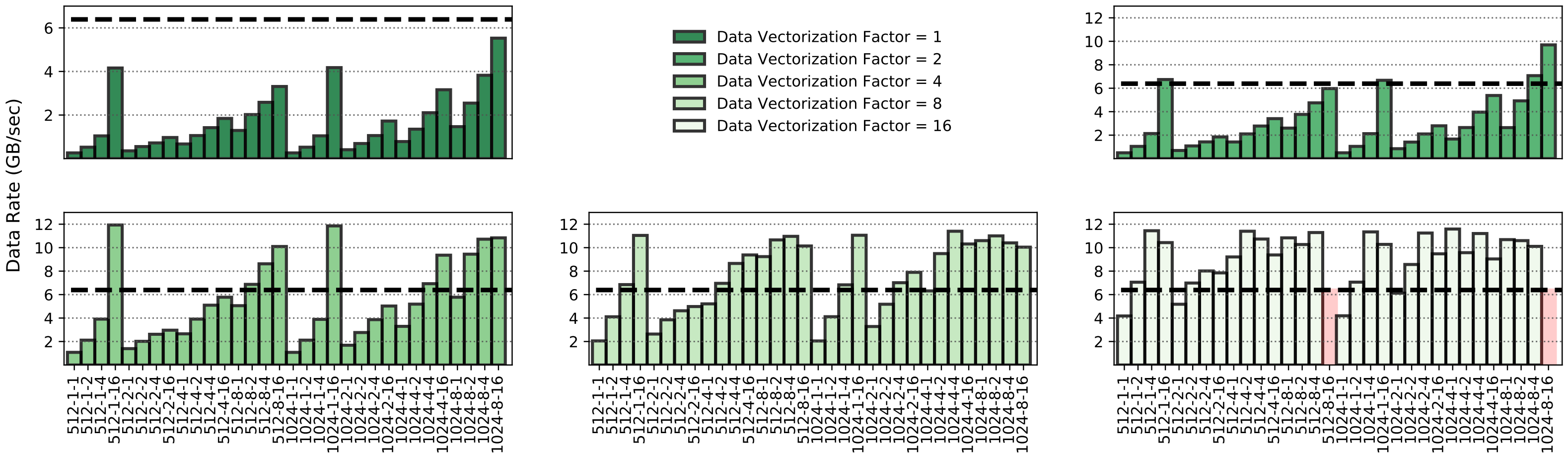


Figure 6.18: Design space search for data vectorization factors {1, 2, 4, 8, 16}. The x-axes represent the coarse-grained configuration `WGSIZE-NUMCOMPUNITS-NUMSIMD` for the given data vectorization factor. The y-axes show the resulting data rate.

Data Vectorization	Work-group Size	# Compute Units	SIMD Factor	f_{max} (MHz)	Logic	M20K Bits	M20K Blocks	Data Rate (GB/s)	Speedup
1	1024	8	16	238.77	28%	8%	23%	5.529	0.865
	1024	1	16	280.19	24%	6%	16%	4.176	0.654
2	1024	8	16	237.19	29%	9%	28%	9.694	1.517
	1024	8	4	254.71	28%	7%	21%	7.071	1.107
4	512	1	16	268.81	24%	7%	17%	**11.933**	**1.868**
	1024	1	16	268.81	24%	7%	17%	11.856	1.856
8	1024	4	4	258.26	26%	8%	21%	11.400	1.784
	1024	1	16	247.77	24%	8%	20%	11.058	1.731
16	1024	4	1	282.56	26%	7%	19%	11.587	1.814
	512	1	4	255.75	24%	7%	17%	11.443	1.791

Table 6.3: Resource utilization and results for the two best coarse-grained configurations for each level of data type vectorization.

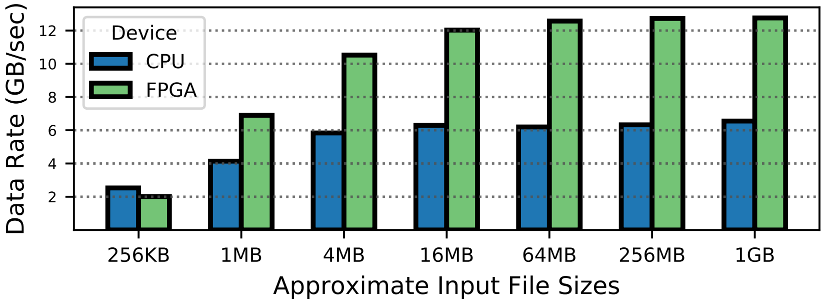


Figure 6.19: Input file size sweep for best vectorized configuration.

6.4.3 Results of Widening The Datatype

Figure 6.18 shows the result of the design space search detailed in Section 6.3.3. Each sub-graph represents a different data vectorization factor.The x-axes show every kernel configuration for its respective vectorization factor, where each label represents:

`WGSIZE-NUMCOMPUNITS-NUMSIMD`.

On the y-axes are the observed data rates for each configuration. The two differently colored bars in Figure 6.18(e) represent configurations that could not be physically realized by the hardware compiler. The dotted black line represents the best data rate for a OpenCL MWI kernel targeting a Intel Core i7 Kaby Lake processor. The line is situated at 6.39 GB/s. Thus, any configuration whose respective bar is below the dotted line has a lower data rate than the multi-core CPU version and a higher data rate if a bar is above the line.

When the data vectorization is set to 1, all configurations in this group perform worse than the best CPU implementation. As shown in Table 6.3, we see that the most performant version without data type vectorization is 0.865× the CPU data rate. However after the first level of type vectorization, i.e., the vectorization factor is set to 2, we observe four configurations that perform better than the CPU. This indicates that while the kernel configuration with just the coarse-grained knobs has been tuned, there is still further room for improvement. Specifically, as the data types become wider 4, 8, and 16, we observe that additional configurations become more performant than the CPU case. This further validates our heuristic from Section 6.2.1 for pruning the design space for MWI kernels.

The main performance benefit of vectorized types comes from aiding the hardware compiler to statically coalesce memory accesses. Without this optimization, the widest interface that can be created for a single compute unit instance of a kernel is by replicating the data path up to 16 times. With the wider data type, a single data path can read and write `N uchars`, where `N` is the data vectorization factor. Additionally, this aids the burst-coalesced load-store unit (LSU) generated by the hardware compiler. Because of the vectorized types, a single address points to multiple data items, as opposed to just one data item. These addresses are queued up and coalesced in the LSU for a burst access. Thus, a burst access can grab up to `N`× more data in the best case when compared to the coarse-grained configuration without data type vectorization.

We also observe evidence supporting the heuristic in Table 6.3, which shows the Intel HARPv2 FPGA resource utilization, data rate, and speedup relative to the CPU for the two best configurations for each level of data type vectorization. We observe that the speedup for `uchar{8,16}` are only 4.7% and 3.0% slower, respectively, than the optimal configuration. This is a reasonable heuristic to follow when one is willing to make the tradeoff of the locally optimal configuration for one that is relatively close to optimal found in less time.

Finding the optimum requires more experimentation, as shown in Figure 6.18. When holding $WGSIZE$ and $NUMCOMPUNITS$ constant, we observe in Figure 6.18 for data vectorization values of 1, 2, and 4 that increasing `NUMSIMD` results in a monotonically increasing data rate. However, this monotonic behavior ends when the data vectorization factor is set to 8 and 16. In this case, replicating the data path with wider types creates enough contention for the global memory resources such that the performance degrades by having to orchestrate these accesses.

From the table, we observe that the overall best performing configuration is (4, 512, 1, and 16) for the data vectorization factor, `WGSIZE`, `NUMCOMPUNITS`, and `NUMSIMD`, respectively. Its data rate is 11.933 GB/s–over one-third of the theoretical read/write bandwidth [42]–and a speedup of 1.868× over the CPU implementation. We observe that the best result does not have the widest data vector type or any replicated compute units. In this case, there is less contention for global memory among compute unit replicates. Additionally, it is easier for the OpenCL runtime to schedule work-groups for execution because there is only one compute unit for which to issue commands. These system-level observations can be aided by observing the reported resource utilization numbers and maximum clock speeds in Table 6.3. (Historically, related work [149] as well as our results from Chapter 5 have been able to account for differences in performance, in part, by using such results.) Future work could include being able to incorporate this data to model the performance impact of interactions like these between design choices in order to more efficiently search the design space.

Figure 6.19 compares the performance of the best kernel configuration to the best CPU version when scaling the input size from approximately 256 KB to 1 GB. The CPU data rate performance starts to plateau at when the input file size is 16 MB, and the best achievable data rate is 6.55 GB/s. The Intel HARPv2 platform data rate begins to plateau when the input size is greater than 16 MB, and the best achievable data rate is 12.76 GB/s. The speedup factor of the Intel HARPv2 performance over the CPU is 1.95×. Although the kernel is relatively simple, input sizes of 16 MB and up are sufficient to stress the system into the asymptotic limit for data rate.

6.5 Conclusion

We have illustrated the use of our multi-spectral reuse distance tool to measure locality on the DIBS application, and the use of those results as inputs to a clustering algorithm to to classify applications within the data integration domain. The two resulting clusters formed two distinct sub-domains. Our hypothesis was that applications within each sub-domain will benefit from different hardware implementations. Because these clusters effectively divided these applications into two different classes of locality performance, and because these applications are comprised mostly of data movement, the multi-spectral reuse data proved to be an accurate predictor of which hardware execution model to choose. Our evaluation of this hypothesis was comprised of selecting two representative applications from each sub-domain, performing a comprehensive search of the design space for each of these representative applications, and using those results to assess our hypothesis.

Additionally, we presented the use of CDFGs to visualize the pre-synthesized hardware in order to make more informed design decisions. We also develop a design heuristic for MWI kernels to prune the space design space, trading the optimal configuration for a near-optimal one using less development time. By sweeping the the target kernel's hardware knobs, we show that the interactions between knobs are non-trivial. Specifically, we show that there is a benefit to vectorizing data types for buffers that will be accessed contiguously. However, global memory contention induced when knob settings were near their maximum values necessitates a finer tuning of the configuration to achieve optimal performance. Finally, we show that scaling the input size in our case study stressed our platform enough to reach the asymptotic data rate.

Chapter 7

Conclusion and Future Work

In this dissertation, we presented our work towards domain specific computing. We addressed two fundamental research questions regarding domain identification and domain specific hardware design. Towards the end of domain specific computing, we outlined our methodology for specifying a domain. We created a definition for the domain of data integration by crafting a definition for this domain and then creating a benchmark suite comprised of applications that aligned with our definition. From there, we used metrics from the literature to characterize these applications and to provide insights to what features might be beneficial to hardware designed specifically for this domain. Specifically, we gleaned the importance of data movement and locality based on our initial characterization. This approach of crafting a domain definition, creating a suite of applications to reflect that definition, and then characterizing them using a battery of metrics is generalizable to the identification of most if not all domains of interest.

We evaluated the Intel HARPv2 system using OpenCL as our domain specific hardware design platform and design framework. First, we evaluated the portability and performance of the HARPv2 system. Before our work, the literature regarding targeting the HARPv2 system using OpenCL was sparse. We contributed to this area by using OpenCL kernels originally designed for FPGAs attached via PCIe card, synthesizing them for the HARPv2 system, and comparing the performance between the two. From this evaluation, we showed that OpenCL FPGA design techniques intended for PCIe card FPGAs were also beneficial on the HARPv2. The measured results from the PCIe card FPGA were better than those of the HARPv2, but when accounting for the benefit of shared memory between HARPv2 CPU and FPGA, we showed that the HARPv2 system was much more efficient at moving data between host and device. Going forward, any hardware developers targeting the HARPv2 platform should utilize the shared memory region of the HARPv2 system in order to create the most performant HARPv2 designs.

Guided by the identification and characterization of our domain, as well as the lessons learned in our evaluation of the HARPv2 system, we demonstrated our work towards architecting domain specific hardware. Using our novel multi-spectral reuse distance tool, we quantified the spatial and temporal locality of our benchmarking suite, and used the outputs generated by this tool as features into an unsupervised clustering technique. We posited that the resulting clusters represented a hardware design choice regarding "width" versus "depth". Specifically, is a given kernel better designed as a widely vectorized or deeply pipelined compute unit. To evaluate our predictions, we used a subset of the DIBS applications and architected both wide and deep versions of these applications, and provided new methods and insights to designing HARPv2 OpenCL kernels and additional optimizations to add once the most performant execution model was found. In the end, the predictions made by our method correctly chose the most performant execution model for each application we tested. We showed that the prevalence of data movement in these applications validated our choice of using locality measures to cluster the applications into sub-domains. This technique, then, is promising, and should be applicable to the data-driven design of domain specific hardware.

7.1 Future Work

There are many possibilities for future work. Here, we describe several such possibilities.

Intelligent Design Space Search

We observed in Chapters 5 and 6 that the amount of time spent synthesizing kernels OpenCL kernels may be prohibitive to applying our design space search methods to kernels for which the design space is considerably larger. Related work shows that, with appropriate models of the hardware and application, the time spent finding the most performant knob configuration could be drastically shortened by not synthesizing every possible configuration [134]. An avenue of future work, then would be to build and evaluate models of the HARPv2 system and target applications to save weeks of compute time spent synthesizing the entire design space.

What/Where to Accelerate

As evidenced by our acceleration of the `fa_2bit` application, we observed that even within a kernel of computation, the correct choice of execution model may not be uniform across the whole kernel. Additionally, we relied on the Linux `perf` utility to tell us where to focus our hardware design efforts. Because the next wave of computing is becoming increasingly heterogeneous (e.g., high-bandwidth memories, processing near memories, TPUs, GPUs, FPGAs), tooling to determine what parts of applications would be beneficial on which hardware platform would be a fruitful future endeavor. Additionally, this heterogeneity begs the question of how to orchestrate all of this heterogeneity. Recently, there has been work towards single-source programming models like SYCL [139] that aim to coordinate the power of heterogeneous systems using a single source programming model.

A New Domain

We claim that the methods presented in this dissertation are general. To test this claim, another direction of future work would be to find another domain for which to perform this analysis in order to craft hardware specific to that domain. In this work, we were able to exploit the prevalence of data movement in our target domain and use features that describe data movement as effective predictors of OpenCL hardware design choices. However, there are other possible features that could be used in addition to the outputs generated by our multi-spectral reuse distance tool.

Hardware Design Patterns

Design patterns are a concept utilized by software engineers in order to craft solutions to problems without having to reinvent the wheel. Similar work by DeHon et al. has proposed the same kind of approach using FPGA design [37], and leveraging a library of composable components to create hardware designs. This is especially important because FPGA design using traditional approaches like Verilog and VHDL are difficult. To extend this to OpenCL FPGA research, another research direction would be to compile a set of commonly-used components when targeting FPGAs through using OpenCL, e.g., sliding windows and stencil computations, and create a parameterizable framework and library in order to make OpenCL FPGA development more palatable.

The Right Programming Language Abstraction

While OpenCL makes the use of FPGAs easier than using traditional methods, an open question is whether or not OpenCL C is the best programming language to target FPGAs, in part, because it is unclear if it is the right level of abstraction with respect to the hardware-software stack [100]. For example, common, performant hardware constructs such as shift registers for latency hiding are often used in FPGAs. It is possible to author OpenCL code such that the hardware compiler will synthesize a shift register, but the notation is clunky and does not intuitively convey a parallel operation that happens on one clock cycle. This problem, then, becomes a fascinating intersection between hardware design and programming languages.

References

[1] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. GROMACS: High Performance Molecular Simulations Through Multi-level Parallelism from Laptops to Supercomputers. *SoftwareX*, 1:19–25, 2015.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An Extensible System for design and execution of scientific orkflows. In *Proc. of 16th Int'l Conf. on Scientific and Statistical Database Management*, pages 423–424, June 2004.

[3] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, and David J Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] Fredy Augusto M Alves, Peter Jamieson, Lucas B da Silva, Ricardo S Ferreira, and José Augusto M Nacif. Designing a Collision Detection Accelerator on a Heterogeneous CPU-FPGA Platform. In *Proc. of Int'l Conf. on ReConFigurable Computing and FPGAs*, 2017.

[5] Amazon EC2 Instance Types, 2019.

[6] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proc. of 49th Design Automation Conference*, pages 1212–1221, 2012.

[7] Jonathan C Beard. The Sparse Data Reduction Engine (spidre): Chopping Sparse Data One Byte at a Time. In *Proc. of 2nd International Symposium on Memory Systems*. ACM, October 2017.

[8] Jonathan C Beard and Joshua Randall. Eliminating Dark Bandwidth: A Data-centric View of Scalable, Efficient Performance, Post-Moore. In *Proc. of International Conference on High Performance Computing*, pages 106–114. Springer, 2017.

[9] Mirza Beg and Peter Van Beek. A Graph Theoretic Approach to Cache-conscious Placement of Data for Direct Mapped Caches. *ACM SIGPLAN Notices*, 45(8):113–120, 2010.

[10] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of the*

17th International Conference on Parallel Architectures and Compilation Techniques, pages 72–81, New York, NY, USA, 2008. ACM.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[12] Mark Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[13] Shawn Bowers and Bertram Ludäscher. An Ontology-Driven Framework for Data Transformation in Scientific Workflows. In Erhard Rahm, editor, *Data Integration in the Life Sciences*, volume 2994 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2004.

[14] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144, 2012.

[15] Randal E Bryant and David R O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall Upper Saddle River, 2003.

[16] P. Buneman, S.B. Davidson, K. Hart, C. Overton, and L. Wong. A Data Transformation System for Biological Data Sources. In *Proc. of 21st Int'l Conf. on Very Large Data Bases*, pages 158–169, September 1995.

[17] Anthony M Cabrera and Roger D Chamberlain. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2. In *Proc. of Int'l Workshop on OpenCL*. ACM, April 2019.

[18] Anthony M. Cabrera and Roger D. Chamberlain. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2: Research Artifacts. `https:/doi.org/10.7936/m2yq-a123`, April 2019.

[19] Anthony M Cabrera and Roger D Chamberlain. Designing Domain Specific Computing Systems. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 221–221. IEEE, 2020.

[20] Anthony M Cabrera, Roger D Chamberlain, and Jonathan C Beard. Multi-spectral Reuse Distance: Divining Spatial Information from Temporal Data. In *Proc. of High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2019.

[21] Anthony M Cabrera, Clayton Faber, Kyle Cepeda, Robert Deber, Cooper Epstein, Jason Zheng, Ron K Cytron, and Roger Chamberlain. Data Integration Benchmark Suite v1. 2018.

[22] Anthony M Cabrera, Clayton J Faber, Kyle Cepeda, Robert Derber, Cooper Epstein, Jason Zheng, Ron K Cytron, and Roger D Chamberlain. DIBS: A Data Integration Benchmark Suite. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 25–28, 2018.

[23] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. A Programmable Parallel Accelerator for Learning and Classification. In *Proc. of 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 273–284, 2010.

[24] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proc. of 19th ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, pages 33–36, 2011.

[25] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A Cloud-scale Acceleration Architecture. In *Proc. of 49th IEEE/ACM Int'l Symp. on Microarchitecture*, pages 7:1–7:13, 2016.

[26] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of IEEE International Symposium on Workload Characterization*, pages 44–54, October 2009.

[27] Razvan Cheveresan, Matt Ramsay, Chris Feucht, and Ilya Sharapov. Characteristics of Workloads Used in High Performance and Technical Computing. In *Proc. of ACM 21st Int'l Conf. on Supercomputing*, pages 73–82, 2007.

[28] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proc. of 53rd Design Automation Conference*, pages 109:1–109:6, 2016.

[29] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The Sanger FASTQ File Format for Sequences with Quality Scores, and the Solexa/Illumina FASTQ Variants. *Nucleic acids research*, 38(6):1767–1771, 2009.

[30] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable Domain-specific Computing. *IEEE Design & Test of Computers*, 28(2):6–15, 2010.

[31] Thomas M Conte and Wen-mei W Hwu. Benchmark Characterization for Experimental System Evaluation. In *Proc. of 23rd Hawaii Int'l Conf. on System Sciences*, volume 1, pages 6–18. IEEE, 1990.

[32] M. O. Cruz, H. Macedo, and A. Guimares. Grouping Similar Trajectories for Carpooling Purposes. In *Proc. of Brazilian Conference on Intelligent Systems*, pages 234–239, November 2015.

[33] William J Dally. GPU Computing: To Exascale and Beyond, 2010.

[34] Tamraparni Dasu and Theodore Johnson. *Exploratory Data Mining and Data Cleaning.* John Wiley & Sons, Inc., 2003.

[35] Tom Deakin, Wayne Gaudin, and Simon McIntosh-Smith. On the Mitigation of Cache Hostile Memory Access Patterns on Many-core CPU Architectures. In *Proc. of International Conference on High Performance Computing*, pages 348–362. Springer, 2017.

[36] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, 13(3):219–237, 2005.

[37] André DeHon, Joshua Adams, Michael DeLorimier, Nachiket Kapre, Yuki Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton. Design Patterns for Reconfigurable Computing. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–23. IEEE, 2004.

[38] Peter J Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[39] Zineb El Akkaoui and Esteban Zimanyi. Defining ETL Worfklows Using BPMN and BPEL. In *Proc. of ACM Int'l Workshop on Data Warehousing and OLAP*, pages 41–48, 2009.

[40] Clayton J Faber, Anthony M Cabrera, Orondé Booker, Gabe Maayan, and Roger D Chamberlain. Data Integration Tasks on Heterogeneous Systems Using OpenCL. In *Proceedings of the International Workshop on OpenCL*, pages 1–1, 2019.

[41] Thomas Faict. Exploring OpenCL on a CPU-FPGA Heterogeneous Architecture. Master's thesis, Ghent University, 2018.

[42] Thomas Faict, Erik D'Hollander, Dirk Stroobandt, and Bart Goossens. Exploring OpenCl on a CPU-FPGA Heterogeneous Architecture Research Platform. In *Proc. of Int'l Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC)*, 2019.

[43] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. CNP: An FPGA-based Processor for Convolutional Networks. In *Proc. of Int'l Conf. on Field Programmable Logic and Applications*, pages 32–37, 2009.

[44] James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes, and Richard L Phillips. *Introduction to Computer Graphics*, volume 55. Addison-Wesley Reading, 1994.

[45] Giancarlo Fortino and Paolo Trunfio, editors. *Internet of Things Based on Smart Objects*. Springer, 2014.

[46] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. AJAX: An Extensible Data Cleaning Tool. *SIGMOD Rec.*, 29(2):590, June 2000.

[47] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. Spector: An OpenCL FPGA Benchmark Suite. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 141–148. IEEE, 2016.

[48] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *Proc. of ACM SIGMOD Int'l Conf. on Management of Data*, pages 1197–1208, June 2013.

[49] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer*, 28(4):23–31, 1995.

[50] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented FPGA Computing in the Streams-C High Level Language. In *Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 49–56, 2000.

[51] Martin Gollery. Bioinformatics: Sequence and Genome Analysis. *Clinical Chemistry*, 51(11):2219–2219, 2005.

[52] Xiaoming Gu, Ian Christopher, Tongxin Bai, Chengliang Zhang, and Chen Ding. A Component Model of Spatial Locality. In *Proc. of ACM Int'l Symp. on Memory Management*, pages 99–108, 2009.

[53] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive Wrangling: Mixed-initiative End-user Programming of Data Transformation Scripts. In *Proc. of 24th ACM Symp. on User Interface Software and Technology*, pages 65–74, 2011.

[54] Philip Jia Guo. *Software Tools to Facilitate Research Programming*. PhD thesis, Stanford Univ., May 2012.

[55] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality Principle Revisited: A Probability-based Quantitative Approach. *Journal of Parallel and Distributed Computing*, 73(7):1011–1027, 2013.

[56] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of Fourth IEEE International Workshop on Workload Characterization*, pages 3–14, December 2001.

[57] John L Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[58] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis. In *Proc. of IEEE 26th International Conference on Data Engineering Workshops*, pages 41–51, March 2010.

[59] Sitao Huang, Li-Wen Chang, Izzat El Hajj, Simon Garcia De Gonzalo, Juan Gómez-Luna, Sai Rahul Chalamalasetti, Mohamed El-Hadedy, Dejan Milojicic, Onur Mutlu, Deming Chen, et al. Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures. April 2019.

[60] Intel. Intel® FPGA SDK for OpenCL™ Pro Edition Best Practices Guide, September 2019.

[61] Intel. Intel® FPGA SDK for OpenCL™ Pro Edition Programming Guide, December 2019.

[62] Arpith Jacob, Joseph Lancaster, Jeremy Buhler, Brandon Harris, and Roger D. Chamberlain. Mercury BLASTP: Accelerating Protein Sequence Alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2):9:1–9:44, June 2008.

[63] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[64] Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu. Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs. In *Proc. of ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays*, pages 299–309, 2020.

[65] Zheming Jin and Hal Finkel. Evaluating an OpenCL FPGA Platform for HPC: A Case Study with the HACCmk Kernel. In *Proc. of IEEE High Performance Extreme Computing Conference*, 2018.

[66] Zheming Jin and Hal Finkel. Evaluation of MD5Hash Kernel on OpenCL FPGA Platform. In *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1026–1032, 2018.

[67] Zheming Jin and Hal Finkel. Nuclear Reactor Simulation on OpenCL FPGA: A Case Study of RSBench. In *Proc. of International Workshop on OpenCL*, pages 2:1–2:9, 2018.

[68] Zheming Jin and Hal Finkel. Performance-oriented Optimizations for OpenCL Streaming Kernels on the FPGA. In *Proc. of International Workshop on OpenCL*, pages 1:1–1:8. ACM, 2018.

[69] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 2017.

[70] Sean Kandel, Jeffrey Heer, Catherine Plaisant, Jessie Kennedy, Frank van Ham, Nathalie Henry Riche, Chris Weaver, Bongshin Lee, Dominique Brodbeck, and Paolo Buono. Research Directions in Data Wrangling: Visualizations and Transformations for Usable and Credible Data. *Information Visualization*, 10(4):271–288, October 2011.

[71] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive Visual Specification of Data Transformation Scripts. In *Proc. of SIGCHI Conf. on Human Factors in Computing Systems*, pages 3363–3372, 2011.

[72] W James Kent. BLAT–the BLAST-like alignment tool. *genome research*, 12(4):656–664, 2002.

[73] G Kestor, R Gioiosa, D J Kerbyson, and A Hoisie. Quantifying the Energy Cost of Data Movement in Scientific Applications. In *Proc. of IEEE Int'l Symp. on Workload Characterization*, pages 56–65, September 2013.

[74] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[75] Sanjeev Kumar and Christopher Wilkerson. Exploiting Spatial Locality in Data Caches Using Spatial Footprints. In *Proc. of 25th Int'l Symp. on Computer Architecture*, pages 357–368, 1998.

[76] George Kurian, Omer Khan, and Srinivas Devadas. The Locality-aware Adaptive Cache Coherence Protocol. *ACM SIGARCH Computer Architecture News*, 41(3):523–534, 2013.

[77] Jakub Kurzak, Yaohung M Tsai, Mark Gates, Ahmad Abdelfattah, and Jack Dongarra. Massively Parallel Automated Software Tuning. In *Proc. of 48th Int'l Conf. on Parallel Processing*, 2019.

[78] Thierry Lafage and André Seznec. Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream. In L K John and A M G Maynard, editors, *Workload Characterization of Emerging Computer Applications*, volume 610 of *SECS*, pages 145–163. 2001.

[79] Christoph Lameter. NUMA (Non-uniform Memory Access): An Overview. *Queue*, 11(7):40, 2013.

[80] Edward D Lazowska, John Zahorjan, G Scott Graham, and Kenneth C Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

[81] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proc. of 30th ACM/IEEE International Symposium on Microarchitecture*, pages 330–335, 1997.

[82] Byron C. Lewis and Albert E. Crews. The Evolution of Benchmarking as a Computer Performance Evaluation Technique. *MIS Quarterly*, 9(1):7–16, March 1985.

[83] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, and Richard Durbin. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[84] M. Lichman. UCI Machine Learning Repository, 2013.

[85] David J Lipman and William R Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227(4693):1435–1441, 1985.

[86] Scott Lloyd and Maya Gokhale. In-memory Data Rearrangement for Irregular, Data-intensive Computing. *Computer*, (8):18–25, 2015.

[87] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[88] Ryouhei Maeda and Tsutomu Maruyama. An Implementation Method of Poisson Image Editing on FPGA. In *Proc. of 27th Int'l Conf. on Field Programmable Logic and Applications*, pages 1–6, 2017.

[89] Atabak Mahram and Martin C Herbordt. FMSA: FPGA-accelerated ClustalW-based Multiple Sequence Alignment Through Pipelined Prefiltering. In *Proc. of IEEE 20th Int'l Symp. on Field-Programmable Custom Computing Machines*, pages 177–183, 2012.

[90] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. Everything You Always Wanted to Know About Multicore Graph Processing but were Afraid to Ask. In *Proc. of USENIX Annual Technical Conference (USENIX ATC)*, pages 631–643, Santa Clara, CA, July 2017. USENIX Association.

[91] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, Andre Shumacher, Anthony D. Joseph, and David A. Patterson. ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing. Technical Report UCB/EECS-2013-207, UC Berkeley, December 2013.

[92] Richard L Mattson, Jan Gecsei, Donald R Slutz, and Irving L Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[93] Michael C McFarland, Alice C Parker, and Raul Camposano. The High-level Synthesis of Digital Systems. *Proceedings of the IEEE*, 78(2):301–318, 1990.

[94] Nathaniel McVicar, Chih-Ching Lin, and Scott Hauck. K-mer Counting Using Bloom Filters with an FPGA-Attached HMC. In *Proc. of IEEE 25th Int'l Symp. on Field-Programmable Custom Computing Machines*, pages 203–210, 2017.

[95] William K. Michener. Building SEEK: The Science Environment for Ecological Knowledge. In *DataBits: An Electronic Newsletter for Information Managers*, 2003.

[96] Sparsh Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys (CSUR)*, 49(2):35, 2016.

[97] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting Taxi–Passenger Demand Using Streaming Data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, September 2013.

[98] H DeGroot Morris. *Probability and Statistics: Classic Version.* Prentice Hall, Inc., 2018.

[99] Richard C Murphy and Peter M Kogge. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and its Implications. *IEEE Transactions on Computers*, 56(7):937–945, 2007.

[100] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable Accelerator Design with Time-Sensitive Affine Types. *arXiv preprint arXiv:2004.04852*, 2020.

[101] Tom Oinn, Mark Greenwood, Matthew Addis, M Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, et al. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[102] John Palmer. The Intel® 8087 Numeric Data Processor. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 174–181. ACM, 1980.

[103] Milan Pavlovic, Nikola Puzovic, and Alex Ramirez. Data Placement in HPC Architectures with Heterogeneous Off-chip Memory. In *Proc. of Int'l Conf. on Computer Design*, pages 193–200. IEEE, 2013.

[104] Georgios Petrousis. An Evaluation of Decoupled Access Execute on ARMv8. Master's thesis, Uppsala University, 2017.

[105] Abhinav Podili, Chi Zhang, and Viktor Prasanna. Fast and Efficient Implementation of Convolutional Neural Networks on FPGA. In *Proc. of 28th Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 11–18. IEEE, 2017.

[106] Meikel Poess, Tilmann Rabl, Hans-Arno Jacobsen, and Brian Caufield. TPC-DI: The First Industry Benchmark for Data Integration. *Proceedings of the VLDB Endowment*, 7(13):1367–1378, 2014.

[107] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proc. of 41st Int'l Symp. on Computer Architecuture (ISCA)*, pages 13–24, 2014.

[108] Vijayshankar Raman and Joseph M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of 27th Int'l Conf. on Very Large Data Bases*, pages 381–390, September 2001.

[109] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. In *Proc. of IEEE Int'l Symp. on Workload Characterization*, pages 110–119, 2014.

[110] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[111] Yousef Saad. *Iterative Methods For Sparse Linear Systems.* SIAM, 2003.

[112] Ahmed Sanaullah and Martin C Herbordt. FPGA HPC Using OpenCL: Case Study in 3D FFT. In *Proc. of 9th Int'l Symp. on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 7:1–7:6, 2018.

[113] Ahmed Sanaullah and Martin C Herbordt. Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. In *Proc. of IEEE High Performance Extreme Computing Conference*, 2018.

[114] Ahmed Sanaullah, Rushi Patel, and Martin Herbordt. An Empirically Guided Optimization Framework for FPGA OpenCL. In *Proc. of Int'l Conf. on Field-Programmable Technology*, pages 46–53. IEEE, 2018.

[115] Robert R Schaller. Moore's Law: Past, Present and Future. *IEEE spectrum*, 34(6):52–59, 1997.

[116] SG Shirinivas, S Vetrivel, and NM Elango. Applications of Graph Theory in Computer Science: An Overview. *International Journal of Engineering Science and Technology*, 2(9):4610–4621, 2010.

[117] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *Proc. of ACM International Conference on Management of Data*, pages 403–415, 2017.

[118] Harsha Vardhan Simhadri. *Program-Centric Cost Models for Locality and Parallelism.* PhD thesis, Carnegie Mellon University, 2013.

[119] Elizabeth S Sorenson and J Kelly Flanagan. Cache Characterization Surfaces and Predicting Workload Miss Rates. In *Proc. of IEEE Int'l Workshop on Workload Characterization*, pages 129–139, 2001.

[120] Elizabeth S Sorenson and J Kelly Flanagan. Evaluating Synthetic Trace Models Using Locality Surfaces. In *Proc. of IEEE Int'l Workshop on Workload Characterization*, pages 23–33, 2002.

[121] Jeffrey R Spirn and Peter J Denning. Experiments with Program Locality. In *Proc. of ACM Fall Joint Computer Conference, Part I*, AFIPS, pages 611–621, 1972.

[122] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable Window Generation for the Intel Broadwell+Arria 10 and High-bandwidth FPGA Systems. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 173–182, 2018.

[123] Yushan Su, Michael Anderson, Jonathan I Tamir, Michael Lustig, and Kai Li. Compressed Sensing MRI Reconstruction on Intel HARPv2. In *Proc. of 27th Int'l Symp. on Field-Programmable Custom Computing Machines*, pages 254–257. IEEE, 2019.

[124] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Forge: Generating a High Performance DSL Implementation from a Declarative Specification. In *Proc. of 12th Int'l Conf. on Generative Programming: Concepts & Experiences*, pages 145–154, 2013.

[125] Qing Y. Tang and Mohammed A. S. Khalid. Acceleration of k-means Algorithm Using Altera SDK for OpenCL. *ACM Trans. Reconfigurable Technol. Syst.*, 10(1):6:1–6:19, December 2016.

[126] Clark Tibbetts. Raw Data File Formats, and the Digital and Analog Raw Data Streams of the ABIPRISM™ 377 DNA Sequencer©. *Unpublished. Available online at: www.cs.cmu.edu/afs/cs/project/genome/ftp/other/377RawData.ps*, 1995.

[127] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations. In *Proc. of ACM/IEEE Conf. on Supercomputing*, 2007.

[128] Panos Vassiliadis, Zografoula Vagena, Spiros Skiadopoulos, Nikos Karayannidis, and Timos Sellis. ARKTOS: Towards the Modeling, Design, Control and Execution of ETL Processes. *Information Systems*, 26(8):537–561, December 2001.

[129] Fritz Venter and Andrew Stein. Images & Videos: Really Big Data. *Analytics Magazine*, pages 14–47, 2012.

[130] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. *ACM SIGPLAN Notices*, 31(9):279–289, 1996.

[131] Virtual Astronomical Observatory. VAO Annual Report, April 2013.

[132] Panagiotis D Vouzis and Nikolaos V Sahinidis. GPU-BLAST: Using Graphics Processors to Accelerate Protein Sequence Alignment. *Bioinformatics*, 27(2):182–188, 2010.

[133] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. Capturing the Spatio-Temporal Behavior of Real Traffic Data. *Performance Evaluation*, 49(1-4):147–163, 2002.

[134] Shuo Wang, Yun Liang, and Wei Zhang. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *Proc. of 54th Design Automation Conference*, 2017.

[135] Yu Wang. *Accelerating Graph Processing on a Shared-Memory FPGA System*. PhD thesis, Carnegie Mellon University, 2018.

[136] Jonathan Weinberg, Michael O McCracken, Erich Strohmaier, and Allan Snavely. Quantifying Locality in the Memory Access Patterns of HPC Applications. In *Proc. of ACM/IEEE Conference on Supercomputing*, 2005.

[137] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.

[138] T. Wolf and M. Franklin. CommBench–A Telecommunications Benchmark for Network Processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–162, 2000.

[139] Michael Wong, Andrew Richards, Maria Rovatsou, and Ruyman Reyes. Khronoss OpenCL SYCL to Support Heterogeneous Devices for C++, 2016.

[140] Andrew Yang. Deep Learning Training at Scale Spring Crest Deep Learning Accelerator (Intel® Nervana NNP-T). In *Proc. of Hot Chips 31 Symposium (HCS)*, pages 1–20. IEEE, 2019.

[141] Takashi Yokota, Kanemitsu Ootsu, and Takanobu Baba. Potentials of Branch Predictors: From Entropy Viewpoints. In *Proc. of International Conference on Architecture of Computing Systems*, pages 273–285. Springer, 2008.

[142] Masato Yoshimi, Yasin Oge, and Tsutomu Yoshinaga. Pipelined Parallel Join and its FPGA-based Acceleration. *ACM Trans. Reconfigurable Technol. Syst.*, 10(4):28:1–28:28, December 2017.

[143] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proc. of ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays*, pages 161–170, 2015.

[144] Chi Zhang and Viktor Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *Proc. of ACM/SIGDA Int'l Symp. on Field-Programmable Gate Arrays*, pages 35–44, 2017.

[145] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient Large-scale Approximate Nearest Neighbor Search on OpenCL FPGA. In *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition*, pages 4924–4932, 2018.

[146] Yu Zheng, Lizhu Zhang, Xing Xie, and Wei-Ying Ma. Mining Interesting Locations and Travel Sequences from GPS Trajectories. In *Proc. of 18th International Conference on World Wide Web*, pages 791–800, 2009.

[147] Yutao Zhong, Xipeng Shen, and Chen Ding. Program Locality Analysis Using Reuse Distance. *ACM Transactions on Programming Languages and Systems*, 31(6):20:1–20:39, 2009.

[148] Hamid Reza Zohouri. *High Performance Computing with FPGAs and OpenCL*. PhD thesis, Tokyo Institute of Technology, August 2018.

[149] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proc. of Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, pages 409–420, 2016.

[150] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In *Proc. of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 153–162, 2018.

Vita

Anthony Michael Cabrera

Degrees

Ph.D. Computer Engineering
Washington University in St. Louis
August 2020

M.S. Computer Science
Washington University in St. Louis
August 2018

B.S. Computer Engineering
Second Major, Computer Science
Washington University in St. Louis
May 2015

B.S.A.S. Electrical Engineering
Washington University in St. Louis
May 2015

B.A. Chemical Physics
Minor, Music
Hendrix College
May 2013

Publications

Anthony M. Cabrera and Roger D. Chamberlain, "Design and Performance Evaluation of Optimizations for OpenCL FPGA Kernels", in Proc. of IEEE High-Performance Computing Conference (HPEC), September 2020.

Anthony M. Cabrera and Roger D. Chamberlain, "Designing Domain Specific Compute Systems", in Proc. of 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), May 2020.

Anthony M. Cabrera, Roger D. Chamberlain, and Jonathan C. Beard, "Multi-spectral Reuse Distance: Divining Spatial Information from Temporal Data," in Proc. of IEEE High-Performance Computing Conference (HPEC), September 2019.

Anthony M. Cabrera and Roger D. Chamberlain, "Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2," in Proc. of 7th International Workshop on OpenCL (IWOCL), May 2019. **Received Best Presentation award.**

Clayton J. Faber, **Anthony M. Cabrera**, Oronde Booker, Gabe Maayan, and Roger D. Chamberlain, "Data Integration Tasks on Heterogeneous Systems Using OpenCL," in Proc. of 7th International Workshop on OpenCL (IWOCL), May 2019.

Anthony M. Cabrera, Clayton J. Faber, Kyle Cepeda, Robert Derber, Cooper Epstein, Jason Zheng, Ron K. Cytron, and Roger D. Chamberlain, "DIBS: A Data Integration Benchmark Suite," in Proc. of ACM/SPEC International Conference on Performance Engineering (ICPE) Companion, April 2018.

August 2020

Domain Specific Computing, Cabrera, Ph.D. 2020