Copyright
by
Rashid Kaleem
2017

The Dissertation Committee for Rashid Kaleem
certifies that this is the approved version of the following dissertation:

Efficient execution of irregular programs on heterogeneous systems

Committee:

Keshav Pingali, Supervisor

Donald Fussell

Vijay Janapa Reddi

Tatiana Shpeisman

Emmett Witchel

Efficient execution of irregular programs on heterogeneous systems

by

Rashid Kaleem, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2017

In the name of Allah[1], the Most Gracious, the Most Merciful.

Dedicated to

AK & *SK*

HR

HR & *MHR*

[1] *Allah* is the Arabic word for God in Islam

Acknowledgments

All parise is due to Allah, the Cherisher and Sustainer of all the worlds. I would like to thank my parents who have always encouraged the pursuit of knowledge. I would also like to thank my spouse, whose patience and support was critical for me to focus on my research.

I am indebted to my advisor, Keshav Pingali, for his mentorship and support. His passion, patience, and persistence for research has been a great source of inspiration. His insistence on clarity and excellence has transformed my approach towards research.

While the list of all those who have supported my research directly or indirectly is too long to mention by name, I would like to thank each and everyone who I have interacted with during my graduate studies.

I am thankful to all the wonderfull collaborators and mentors I have had over my tenure as a graduate student. My colleagues, who made our window-less lab a second home, have undoubtedly contributed to my research. I have had the privilege to spent the time with such good colleagues and friends – Donald Nguyen, Dimitris Prountzos, Xin Sui, M. Amber Hassaan, and Gurbinder Gill.

I would also like to thank Tatiana Shpeisman and Raj Barik at Intel Labs, and Gabriel Tanase at IBM Research for hosting me as an intern. I am very fortunate to have enrolled in Professor Fussell's graduate course during my first semester -

not only has his interactions proved invaluable for my research, his course undoubtedly helped my transition to a foreign country tremendously. I am also thankful to my committee members Vijay Reddi and Emmett Witchel for their constructive feedback on my research.

I would like to mention some of my close friends, who have been beside me through thick and thin - Ahmed Abrar, Shahzad Afzal, Fahad Khalil, Jawad Zafar, and Muhammad Shoaib Sehgal (late). I am also thankful for having wonderful mentors over the years – Shaiq A. Haq, Mian M. Saleem, Muhammad Afzal, Syed M. Ahsan, and Margaret Kilvington. Last but not least, I would like to mention Talha Waheed - who encouraged me to attend graduate school in the US.

Efficient execution of irregular programs on heterogeneous systems

Publication No. ______________

Rashid Kaleem, Ph.D.
The University of Texas at Austin, 2017

Supervisor: Keshav Pingali

Programmable accelerators such as GPUs, FPGAs, and DSPs enable modern systems to provide higher performance for many workloads than is possible by using conventional processors alone. Traditionally, portability of applications to these accelerators and between accelerators was a major hurdle in utilizing accelerators in a heterogeneous system. With the emergence of standardized programming APIs such as OpenCL, this problem is being ameliorated and many accelerators can now be programmed using a single API.

In this work, we address the efficient execution of *irregular* programs on heterogeneous systems. Irregular programs are used extensively in problem domains like graph analytics and finite-element methods, and they are characterized by data-dependent control flow and memory accesses that cannot be predicted at compile time. We focus on heterogeneous systems that provide a coherent memory to all devices.

First, we describe a set of compiler and runtime techniques to support efficient execution of irregular programs on heterogeneous systems composed of a CPU and an integrated GPU. The compiler allows applications written in C++ to be executed on the GPU without any programmer effort. The runtime system solves the load imbalance arising from irregularity in the applications by dynamically assigning work to each device.

Next, we present an alternative implementation strategy for irregular applications on a system with more heterogeneity. Specifically, graph applications can be expressed as *producer-consumer* computations on FPGA+CPU heterogeneous systems. This approach allows for better utilization of the capabilities of each device and suggests a programming model for accelerators that goes beyond the *offload* model.

Finally, we explore efficient execution of irregular applications on accelerators that do not share a coherent memory with the master processor. For discrete GPUs, we explore implementation strategies of graph application, focusing on synchronization tradeoffs and present optimizations that address the synchronization overheads both within and across devices.

Table of Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xiii**

List of Figures **xiv**

Chapter 1. Introduction **1**

1.1 Contributions 2

1.2 Outline 3

Chapter 2. Background **5**

2.1 Heterogeneous Systems 5

2.1.1 big.LITTLE 6

2.1.2 Intel Haswell 7

2.1.3 Terasic DE1-SoC 8

2.2 Irregular Applications 10

2.2.1 Parallelism 10

2.3 Graph Applications 14

2.3.1 Operator 16

2.3.2 Schedules 17

2.4 Implementation Issues 18

2.4.1 Graph representation 19

2.4.2 Pull Implementations 21

2.4.3 Push Implementations 24

2.5 Abstractions 27

2.5.1 CUDA 27

2.5.2 OpenCL 28

Chapter 3. Data-parallel execution with Integrated GPUs **29**
3.1 Introduction 29
3.2 Compiler 31
3.2.1 Programming constructs 32
3.2.2 Shared Virtual Memory (SVM) support 34
3.2.3 Virtual Functions 37
3.2.4 Reduction 38
3.2.5 Compiler Optimizations 39
3.2.5.1 Reduce SVM implementation overhead 40
3.2.5.2 Reduce GPU cache-line contention 41
3.3 Heterogeneous execution 43
3.4 Naïve profiling 48
3.4.1 Analysis 50
3.4.2 Determining profiling size 53
3.5 Asymmetric profiling 55
3.5.1 Analysis 57
3.5.2 Addressing load imbalance 59
3.5.3 Multiple invocations per kernel 60
3.5.3.1 Update functions: 60
3.6 Evaluation 61
3.6.1 Environment 61
3.6.2 Benchmarks 62
3.6.3 Comparison schemes 65
3.6.4 Results 67
3.7 Summary 70

Chapter 4. Data-parallel execution with discrete GPUs **72**
4.1 Introduction 72
4.2 Graph algorithms on discrete GPUs 73
4.2.1 Stochastic Gradient Descent (SGD) 75
4.3 Offline Schedules 78
4.3.1 Maximal matchings schedules 79

4.3.1.1 All-Graph Matching-Edge schedule (AGM-E) 79
4.3.1.2 All-Graph Matching-Node schedule (AGM-N) 81
4.3.1.3 Sub-Graph Matching (SGM) 82
4.3.2 Diagonal matchings schedules 83
4.3.2.1 Diagonal (Diag) schedule 84
4.3.2.2 Block-Diagonal (BlkDiag) schedule 84
4.4 Online Schedules 85
4.4.1 Edge-locked (EL) 86
4.4.2 Node-locked (NL) 88
4.5 Heterogeneous schedules 89
4.5.1 Graph partitioning 90
4.6 Evaluation 93
4.6.1 Overall performance 95
4.6.2 Hybrid schedules 98
4.6.3 Offline schedules 99
4.6.4 Heterogeneous schedules 100
4.7 Data driven algorithms 102
4.7.1 Base implementation 105
4.7.2 Combiners 106
4.7.3 Synchronization 108
4.7.4 Partitioning 110
4.7.5 Results 111
4.8 Conclusion 115

Chapter 5. Pipeline-parallel execution with FPGAs **117**
5.1 Introduction 117
5.2 Bottleneck Analysis 120
5.3 Execution Models 121
5.3.1 Data-parallel Execution 122
5.3.2 Gather-Apply Execution 124
5.3.2.1 Implementation choices 126
5.3.3 Apply-Scatter implementation 128

5.4 Evaluation . . . 131
5.4.1 Applications . . . 132
5.4.2 Overall performance . . . 136
5.4.3 Data-parallel implementations . . . 137
5.4.4 Gather-Apply implementations . . . 139
5.4.5 Apply-Scatter implementations . . . 142
5.4.6 Work-list driven implementations . . . 143
5.5 Conclusion . . . 146

Chapter 6. Related work **148**
6.1 Graph programming models . . . 148
6.2 GPU programming . . . 148
6.3 FPGA programming . . . 150
6.4 Heterogeneous execution . . . 151
6.5 Data partitioning and layout . . . 154

Chapter 7. Future work **157**

Chapter 8. Conclusion **160**

Bibliography **162**

List of Tables

3.1 Key statistics for the kernels of benchmarks used in the evaluation. *Oracle-time* is the time taken by the best offline distribution, and is also used as the normalizing factor in Fig. 3.14 and Fig. 3.15. 62

3.2 GPU work percentages computed by different schemes for single-kernel applications. 68

4.1 Specifications of the platforms used for evaluation. 93

4.2 Characteristics of the scale-free and uniform inputs. $|V|$ is the total number of vertices in the graph, $|E|$ is the number of edges in the graph, and D_{max} represents the maximum degree of any node in the graph. $EL(s)$ is the running time of the EL versions in seconds. . . 94

5.1 Application node and edge data. The initial work-list sizes for the work-list driven implementations are also shown. 131

5.2 Application node and edge computations. The push implementations require atomic updates on the edges. 131

5.3 Inputs and their key properties. 132

5.4 Differences in the number of regular and random reads and writes for the node labels of each variant. 136

List of Figures

2.1 Intel Haswell (4th generation) processor architecture [Junkins, 2014]. 8

2.2 Organization of a Cyclone-V SoC from Altera. The DE1-SoC from Terasic is an instance of this SoC. 9

2.3 Adding two array elements. The first snippet add two arrays sequentially, the second one performs an indirect read, while the third one performs an indirect write. 11

2.4 Control flow irregularity. 12

2.5 A graph colored to represent the different dynamic components during execution. *Red* nodes indicate *active elements* – nodes where a computation needs to be performed, and the shaded region around each active element denote the *neighborhood* of each activity – sets of nodes and edges that are accessed during the execution of the activity. 14

2.6 A directed graph and its CSR representation. Node e does not have any outgoing edges, so the indices entries for its edges point to the end of neighbors array. 21

2.7 Single Source Shortest Path solved through Bellman-Ford algorithm. 22

2.8 A directed graph and its CSR representation. Node f does not have any outgoing edges, so the indices entries for its edges point to the end of neighbors array. 22

2.9 Implementation of Bellman-Ford for Single Source Shortest Path. . . 23

2.10 Single Source Shortest Path topology-driven push algorithm. 25

2.11 Single Source Shortest Path work-list driven push algorithm. 26

3.1 Concord programming constructs. 32

3.2 Example demonstrating use of Concord $parallel_for$ construct. . . 33

3.3 Concord compiler and runtime overview. 34

3.4 Address translation from CPU virtual address space to GPU virtual address space. 35

3.5 Example demonstrating Concord SVM implementation. Left side shows the C++ implementation provided by the programmer. Right side shows the transformed code including translation code to convert from CPU virtual addresses to GPU virtual addresses. 36

3.6 Example illustrating compiler transformation of shared pointers on GPU: lazy vs. eager. 40
3.7 Reducing cache-line contention among GPU cores. Left side shows the original code. 42
3.8 Relative execution time of **BH**-BarnesHut (left) and **FD**-Facedetect (right) as ratio of work offloaded to the GPU is varied from 0% to 100% in increments of 10% (*lower is better*). **BH** is optimal at 40% and **FD** is optimal at 0%. 44
3.9 Heterogeneous execution via Naïve profiling. 51
3.10 Plot showing the change in G_r with increasing number of work-items for a regular kernel (note the logarithmic scales) 54
3.11 Plot showing average C_r and G_r for different values of Nf_p for BarnesHut (**BH**) and BarnesHut-unoptimized (**BH-U**). 54
3.12 Heterogeneous execution through symmetric profiling. 57
3.13 Total number of dynamic instructions of benchmarks divided into three categories: *memory*, *control*, and *remaining*, obtained via a serial CPU execution. The number of dynamic instructions is given in Table 3.1(column 9). 63
3.14 Relative speedup for all benchmarks compared to Oracle. Oracle is at 100% (higher is better). 64
3.15 Comparison of different adaptive schemes. Vertical axis shows relative speedup vs. Oracle (higher is better). 66

4.1 Taxonomy of scheduling strategies. 74
4.2 Low-rank approximation of a sparse matrix R by low rank matrices $W : |U| \times t$ and $H : t \times |M|$, usually $t \approx 16$. 76
4.3 Sample bipartite graph between 6 users and 4 movies. Edge labels indicate ratings. 76
4.4 Maximal matchings schedules executed by different strategies for sample input. Tables with **M** in top-left cell indicate matching sets, whereas tables with **T** in top-left cell indicate schedules where each row indicates a time-step and each column lists the edges processed by a thread. 80
4.5 Diagonal matchings schedules for the sample input. 83
4.6 Schedules observed for sample input under EL(without and with shuffle) and NL on the hypothetical GPU. Each row indicates edges scheduled at that time slot, and each column indicates the item processed, if any, by each thread. 86

4.7 Partitioning the graph along movies and users for multi-device execution. Arrows indicate the search pattern for more work for a device which has just completed work on partition $(1, 1)$. First it searches horizontally to maximize user-locality, and if it fails, it retries vertically. The numbers indicate the search order. 91
4.8 Geomean normalized runtime of scheduling schemes over the two input classes evaluated over two platforms. The runtimes are normalized to EL's runtime. Lower is better. 95
4.9 Speedup of hybrid schedule over EL on the different GPUs across the two input classes. Higher is better. 98
4.10 Atomic throughput of the NVIDIA K40 and AMD R9-290X. X-axis shows the number of threads launched, and Y-axis shows the throughput of atomics operations *achieved*. The K40 peaks at about 600M atomic operations per second, while the R9-290X saturates at about 45M. 100
4.11 Absolute runtime of multi-device SGD on different inputs. Each plot represents an input with varying over-decomposition along the x-axis. Different colors represent different device configurations. Lower is better. 101
4.12 Baseline vertex-program implementations for 2 devices. Gray boxes indicate cross-device communication. (a) shows the baseline version, whereas (b) shows the *combiner* version. 103
4.13 Different optimization for vertex-program implementations on 2 devices. Gray boxes indicate cross-device communication. (a) shows the delayed synchronization strategy and (b) shows the vertex-cut implementation. 104
4.14 A *Pregel* program for single source shortest path. 104
4.15 A *combiner* implementation for single source shortest path. 106
4.16 Vertex cut implementations. (a) shows a high degree node x, also known as a hub, whereas (b) shows the hub split into two low degree nodes xa and xb. 110
4.17 (a) Relative runtime for different schemes compared to a base-version. Lower is better. (b)Characteristics of the input and impact of degree threshold vertex-cut performance. 112
4.18 Performance metrics for SSSP on two K40s for FLA road input. The time-step is along the x-axis, and the size of the data structure (work queue - left, local messages - center, and non-local messages - right) are shown along the y-axis. 113
4.19 Performance metrics for SSSP on two K40s for RMAT20 input. The time-step is along the x-axis, and the size of the data structure (work queue - left, local messages - center, and non-local messages - right) are shown along the y-axis. 114

5.1 SSSP front-end and back-end stalls for pull and push versions on scale-free and road network inputs. The bars for GA and AS shows the breakdown of metric for the two phases using different colors. . 121
5.2 A simple gather-apply implementation for Single Source Shortest Path. 122
5.3 An optimized implementation of Single Source Shortest Path where some computation is performed in the gather implementation reducing the overall traffic to the apply. 123
5.4 Single Source Shortest Path topology-driven apply-scatter algorithm. 129
5.5 Single Source Shortest Path work-list driven apply-scatter algorithm. 130
5.6 Geometric means of relative execution time for all variants of different applications on RMAT networks. The fastest variant is at 1. The logic synthesized for Hetero-AS-WL variant for PR did not fit the area, hence was not executed. Lower is better. 134
5.7 Geometric means of relative execution time for all variants of different applications on ROAD networks. The fastest variant is at 1. The logic synthesized for Hetero-AS-WL variant for PR did not fit the area, hence was not executed. Lower is better. 135
5.8 Absolute runtime for different edge-distribution of the graph on a heterogeneous system. x-axis shows the percentage of edges processed on the FPGA and y-axis shows the absolute execution time. . 138
5.9 Relative execution time for different configurations of gather and apply relative to the best implementation, lower is better. 139
5.10 Relative execution time for different configurations of apply and scatter relative to the best implementation, lower is better. 141
5.11 Relative execution time for different configurations of apply and scatter relative to the best implementation on RMAT18, lower is better. The *red* horizontal line indicates 1. 144
5.12 Relative execution time for different configurations of apply and scatter relative to the best implementation on LKS, lower is better. The *red* horizontal line indicates 1. 145

Chapter 1

Introduction

Heterogeneous hardware is emerging as a key direction taken by industry to satisfy expectations of continuous improvement in performance, especially in the datacenter. Traditionally, large-scale deployments of computers were used to execute scientific and financial applications, since these applications required a lot of computational power. In the last decade, applications from new problem domains like machine learning and graph analytics have also become major consumers of cycles in datacenters.

Scaling these diverse applications requires developing new hardware platforms that go beyond traditional latency-optimized cores, which have been shown to be limited by physical design constraints [Beamer et al., 2015, Ozdal et al., 2016, Ham et al., 2016]. Programming these next-generation platforms will require new abstractions as well as revamped tool-chains to support these abstractions efficiently on hardware platforms. While traditionally, a single ISA and architecture design made the transition between platforms very smooth, the transition from homogeneous parallelism to heterogeneous parallelism presents a set of interesting challenges. The trade-off between software and hardware complexity is one of the key challenges - should programmers have to learn new abstractions to express their

algorithms efficiently, or should hardware architects develop hardware that implements existing abstractions more efficiently?

In this dissertation, we explore the execution of irregular programs on heterogeneous systems. Irregular programs capture a large class of problems that are not amenable to compiler optimizations since in these applications, most of the optimization opportunities are available only during execution. Our work shows that while existing programming constructs can be used to efficiently utilize heterogeneous hardware without having programmer effort spent in porting the applications across different hardware platforms, opportunities for further improvement are available if these abstractions are expanded to benefit from the diversity in a heterogeneous system.

1.1 Contributions

This dissertation explores efficient implementation of irregular programs on heterogeneous systems and proposes techniques across the software stack necessary to achieve that goal. Specifically, we highlight contributions in three areas.

1. *Compilers:* These assist in porting applications across devices. While vendors do provide APIs for programming accelerators, programmers have to learn a new API to port applications for accelerators. We show that a compiler can transform existing applications to target these accelerators efficiently.

2. *Runtime systems:* These are necessary to exploit information available only during execution. In a heterogeneous system, the runtime is responsible for

load balancing, communication management and synchronization across the different devices.

3. *Programming abstractions:* The current range of programming models designed for accelerators rely on the *offload* model in which a master processor off-loads computations to the accelerator and obtains the result. We show that this model is inadequate, especially for more diverse accelerators such as the FPGA, and propose new abstractions.

1.2 Outline

The outline of the rest of this dissertation is as follows. We first present some background for the work, covering the hardware and software aspects needed to understand the dissertation. In Chapter 3, we describe a set of compiler and runtime techniques that assist in executing irregular applications on a heterogeneous system composed of a commodity CPU and an integrated GPU. The key challenge addressed here is to reduce the overhead of dynamic load balancing between the heterogeneous processors in such a system. In Chapter 4, we focus on discrete GPUs that provide more computational capabilities at the expense of increased communication latencies and complexity. We explore and present guidelines for implementing efficient synchronization strategies necessary to execute graph applications on discrete GPUs. We also explore heterogeneous execution of graph applications for multi-GPU systems in which optimizing for communication becomes a key challenge in the face of heterogeneity.

In Chapter 5, we explore a system composed of an FPGA and a CPU, which offers increased heterogeneity, and hence presents a different set of opportunities to optimize graph applications. We present a novel strategy of executing graph applications on such heterogeneous systems which addresses CPU performance limitations by offloading some work to the FPGA.

We discuss related work in Chapter 6, present some directions for future work in Chapter 7, and conclude in Chapter 8.

Chapter 2

Background

In this chapter, we present some background to put the dissertation into context. We first briefly describe the motivation for heterogeneous hardware, and describe some heterogeneous platforms available today. Next, we describe irregular applications which is the domain of applications addressed in this dissertation.

2.1 Heterogeneous Systems

There are two factors that have led to a continuous improvement in processor performance over the past two decades - *Moore's Law* and *Dennard's scaling*. Moore's law [Moore et al., 1998] is the observation that the number of of transistors per unit area doubles every 2 years. Equivalently, this can be phrased as shrinking of the transistor sizes - every 2 years, the size of a transistor shrinks to half. Dennard's scaling [Dennard et al., 1974], describes the power requirements of transistors decreases as the size decreases. This roughly translates to a constant power per unit area. However, Dennard's scaling does not take into account power leakage which becomes significant when transistor sizes become small enough. This new scenario, where Dennard's scaling does not apply, but Moore's law does, has led to a different set of challenges for hardware designers. Fundamentally, this means that although

more transistors can be placed into the same area, not all of these can be powered simultaneously due to power and thermal constraints.

Traditionally, these transistors were used to speed up serial processing of a program by different techniques such as pipelining, out-of-order execution, and vectorization. However, the limits of Dennard's scaling prevent future progress along those lines. The alternate direction adopted by the industry has been to turn to parallelism. Instead of accelerating the execution of a single stream of instructions, the hardware provides more throughput by enabling the execution of multiple streams of instructions, possibly at a lower rate.

The simplest strategy to utilize parallelism in a multi-processor is to replicate cores, and add a coordination network. While this approach is simple, an alternative is to use the extra transistors to provide heterogeneous capabilities. We present several different designs next, some of which are used to evaluate software implementations of graph applications in the remainder of the document. The three devices describe different levels of heterogeneity - from using the same ISA in the case of Sec. 2.1.1, to having different design points as demonstrated by Sec. 2.1.3.

2.1.1 big.LITTLE

To satisfy the wide range of requirements presented by applications on a mobile platform, especially power constraints, ARM has developed heterogeneous systems composed of cores that execute the same ISA but differ in power/performance tradeoffs. The central idea is to provide small *LITTLE* cores that can provide power efficiency, and large *big* cores that run at higher clock rates and have greater

instruction throughput while consuming more power than the *LITTLE* cores. The operating system task scheduler [big.LITTLE, 2013] can move tasks from a *LITTLE* core to a *big* core or vice versa according to different dynamic schemes. As the cores share the same ISA, process migration is relatively simple. This design offers the simplest form of heterogeneity, which stems from architecture, contrasted with homogeneous cores behaving differently under dynamic voltage-frequency scaling (DVFS) [Macken et al., 1990].

2.1.2 Intel Haswell

The Intel Haswell, also known as 4th generation Intel Processor, is a commodity x86 processor introduced in 2013. The high level organization of the processor is shown in Fig. 2.1. The processor also contains a generation 7.5 graphics processor. The $HD-4600$ integrated GPU consists of a number of slices, each containing two sub-slices. Each sub-slice consists of 10 execution units (EU), and each EU can execute 7 threads. Each level of the organization has a different set of shared components, allowing for a scalable design for different market segments. The GPU also shares the physical memory with the CPU cores.

The GPUs and the CPUs share a last-level cache (LLC), with a cache line size of 64-bytes. Cache coherency is provided through the shared distributed last-level cache where each "core" (CPU core or a slice) gets a slice of the cache. This is done through a hash of the physical address. Both the integrated GPUs and CPU cores are connected by a bi-directional 32-byte wide ring interconnect. Some models are equipped with an on-die EDRAM which acts as a victim cache. A system

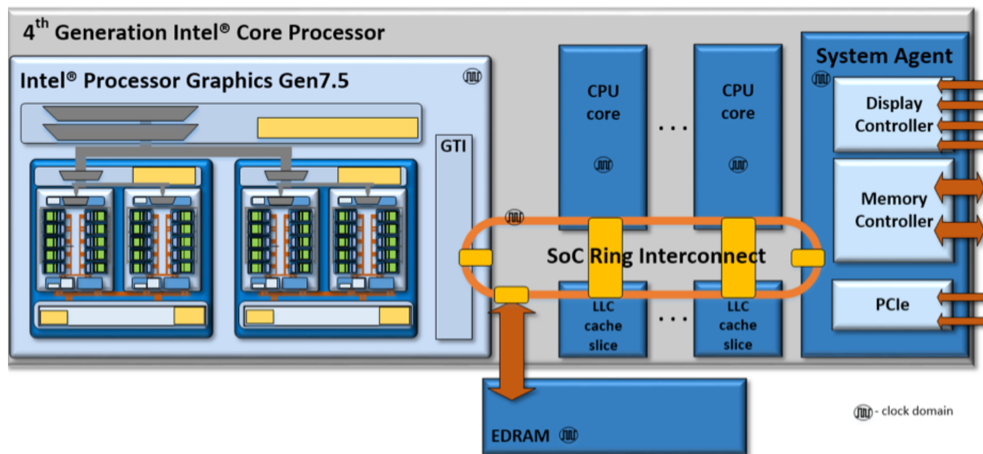


Figure 2.1: Intel Haswell (4th generation) processor architecture [Junkins, 2014].

agent, connected through the ring-interconnect, manages the communication between the CPU/GPU with the memory or other devices.

This system provides a middle ground for heterogeneity, providing a CPU and an integrated GPU which can be used to accelerate graphics workloads at a modest energy budget.

2.1.3 Terasic DE1-SoC

The Terasic DE1-SoC is a system-on-chip with a dual core ARM Cortex-A9 and a Altera Cyclone-V FPGA. The high-level organization of the architecture is given in Fig. 2.2. The MPU subsystem consists of the Cortex-A9 CPU as well as a L2 cache and an accelerator coherency port (ACP) IP mapper which is responsible for coherency with accelerators. The Cortex-A9 has a snoop-control unit (SCU), which snoops for data-accesses to physical addresses shared with the FPGA. Co-

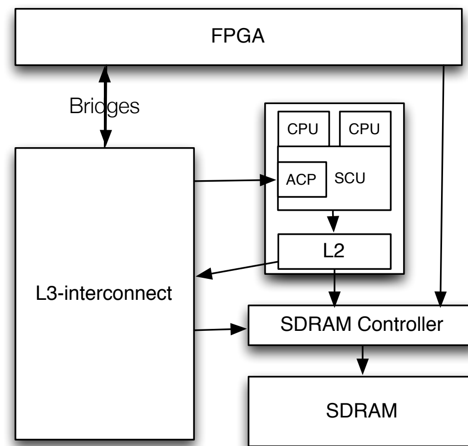


Figure 2.2: Organization of a Cyclone-V SoC from Altera. The DE1-SoC from Terasic is an instance of this SoC.

herency is managed by mapping a window of physical memory to be shared between the CPU and the FPGA. The FPGA fabric communicates with the MPU through a number of bridges which enable bi-directional transfers. The FPGA also has a direct path to the memory subsystems through the SDRAM controller subsystem that bypasses the caches and the L3-controller.

This system provides another extreme design point on the heterogeneous spectrum. The accelerator has a fundamentally different design compared to the CPU, while still providing the ability of general workloads to be executed on the accelerator.

2.2 Irregular Applications

To understand the sources of irregularity and what features distinguish an *irregular* application from a regular one, we start with a simple example of adding two arrays. In a imperative language such as C++, the procedure involves going over the two arrays, and adding the two corresponding elements while storing the result at the current location indexed by a loop counter. This is a fairly simple application, and modern processors excel at this template as it has a very predictable behavior, both in terms of what instructions to execute as well as what data to access.

2.2.1 Parallelism

We first consider a simple program that adds two arrays and stores the result in a new array. The first snippet in Fig. 2.3 illustrates the example where two arrays, *arr_a* and *arr_b*, each containing *num_elements* items, are added and the result is stored in the *result* array. A processor will go over all the entries in order, reading them from both arrays, computing and writing the result into the destination array's corresponding index. Since the addition of individual entries is independent of one another, they can be performed out of order. A modern processor is equipped with a hardware prefetcher, which can detect simple memory access patterns and prefetch addresses before they are accessed by an instruction. In this case, it will detect that consecutive locations of arrays *arr_a* and *arr_b* are being read and *result* being written.

On a multi-core system, which is equipped with multiple processors, the uti-

```
1  ///Sample 1
2  for(int i=0; i< num_elements; ++i){
3     result[i] = arr_a[i] + arr_b[i];
4  }
5  ///Sample 2
6  for(int i=0; i< num_elements; ++i){
7     result[i] = arr_a[i] + arr_b[index[i]];
8  }
9
10 ///Sample 3
11 for(int i=0; i<num_elements;++i){
12    result[index[i]] = arr_a[i] + arr_b[i];
13 }
```

Figure 2.3: Adding two array elements. The first snippet add two arrays sequentially, the second one performs an indirect read, while the third one performs an indirect write.

lization of all the processor requires more coordination. One of the simplest way of utilizing all the cores is to partition the workload between the cores. This is possible for simple computations such as our running example. If *num_elements* is large enough, all the participating cores can work on their individual partitions. Here, too, the prefetchers on each core will be able to detect that each core is working on a sequence of memory locations and prefetch values from each array.

While the *Sample 1* code can be parallelized in a simple and efficient way, most applications do not have such simple patterns. As described later in more detail, most graph applications require accessing memory locations through an indirection array. This is shown in the code annotated *Sample 2* in Fig. 2.3. Here, the application is going over an array of elements, and each iteration adds two values, one from a sequential access of *arr_a*, and the other an indirect access of *arr_b* through *index*. In this case, a serial as well as a parallel execution will show poor locality in referencing *arr_2* as the accesses cannot be predicted by the prefetcher

```
///Sample 1
for(int i=0; i< num_elements; ++i){
   for(int j=0; j < WORK_SIZE; ++j){
   //do work
   }
}
///Sample 2
for(int i=0; i< num_elements; ++i){
   for(int j=0; j<work_size[i]; ++j){
      //do work
   }
}
```

Figure 2.4: Control flow irregularity.

and performance suffers. Another source of access irregularity is having irregular writes as shown in *Sample_3* in Fig. 2.3. Here, *index* array is used to access the location where the result of the addition will be placed.

Just as we have described memory access irregularity above, the control flow of an application can also exhibit irregularity. To observe this, we consider a derivative of the sample codes in the Fig. 2.3. The first code sample in Fig. 2.4 shows a nested loop where the outer loop iterates over *num_elements* items, and for each iteration, performs *WORK_SIZE* inner loop iterations. Assuming that the body of the inner loop has a fixed amount of work, if the value of *WORK_SIZE* is known at compile time, the compiler can perform certain optimizations such as unrolling to avoid the inner loop from requiring a conditional statement. If *WORK_SIZE* is not known at compile time, these optimizations cannot be performed.

When executing the code in *sample_1* of Fig. 2.4 on a parallel system, the scheduler can divide the *num_elements* evenly among the different processors. This balances the load across the different processors since the work performed by each inner loop iteration is the same, even if it is not known at compile time. This can

be shown by the second code sample in Fig. 2.4. Here, each inner loop performs a different amount of iterations, which is known only at runtime through a *work_size* array. This poses two challenges. *First*, the inner loop cannot be optimized by the compiler as described above. *Second*, load balancing across multiple processors becomes difficult since the different inner loop iterations have different amount of work.

For the discussion, we have ignored data-races between the concurrent threads executing the kernel. If the array *index* is not a permutation or contains duplicates, concurrent threads may access the same data. In the case of *Sample_2*, multiple threads maybe reading the same element in *arr_b*, which constitutes a benign race if the element accessed is less than the machine word. However, the code in *Sample_3* results in concurrent threads writing to the same locations if they update the same entry in *result* array. A data race is used to refer to situations where multiple threads concurrently access the same location and at least one of the accesses is a write. This require some form of *synchronization* between the concurrent updates.

There are many solutions to synchronize concurrent access to shared data such as transactional memory (hardware [Herlihy and Moss, 1993] and software [Shavit and Touitou, 1995]), mutual exclusion [Dijkstra, 1965] through locks, and inspector-executor [Saltz et al., 1997].

We have described two of the characteristics of an *irregular* applications - memory access irregularity and control-flow irregularity. Addressing these two behaviors efficiently on modern systems is the key to irregular applications.

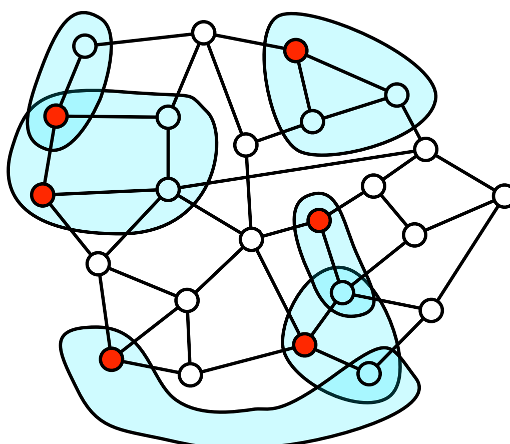


Figure 2.5: A graph colored to represent the different dynamic components during execution. *Red* nodes indicate *active elements* – nodes where a computation needs to be performed, and the shaded region around each active element denote the *neighborhood* of each activity – sets of nodes and edges that are accessed during the execution of the activity.

2.3 Graph Applications

Graphs are a convenient way of representing irregular computations since they allow the representation of arbitrary data structures as well as capture arbitrary relationships through edges. Many irregular applications are expressed using maps and the emergence of large scale social data has helped highlight the need for schema-less representations - which graphs excel at. This prevalence of graphs in modern applications makes it worthwhile to spend some time on characterizing key components of graphs and potentially categorizing the graph instances into sub-group which can then be targeted individually for optimization.

A graph can be described as a set of nodes V and a set of edges E, where each end-point of an edge is a node in the graph. The set of nodes and edges

collectively encapsulate the *topology* or structure of the graph - what the graph looks like. There may be application specific data labels associated with the nodes and the edges. For instance, the nodes may represent individuals on a social network and the edges may represent interactions between the individuals.

Given a graph $G(V, E)$ and the data labels D_V, D_E, a graph application is a program that performs some computation over the graph and generates new label(s) for the nodes or the edges or summarizes the graph. For instance, in a *Single Source Shortest Path*(SSSP) application, the nodes may represent cities and the edges represent the length of the roads between them. The goal of the computation is to find the length of the shortest path from a specific node or city (*source*) to every other node. The data labels on the node are all initialized to *infinity* except for the source which is set to 0. While there are many different algorithms for computing SSSP, there is one key operation common to all – *if a shorter path is available for a city, the distance label of that city should be updated.* If no such update exists, the computation has terminated, and each node (city) has its shortest distance as its label.

A graph application can be specified through two key components - *operator* and *schedule*. The *operator* defines what computation needs to be performed, whereas the *schedule* defines when and where the computation needs to be performed.

2.3.1 Operator

We refer to this computation as the *operator*. An operator can be applied to different components of a graph - nodes, edges or sub-graphs. These are the *active elements* of the application and define what component of the graph the operator is applied to. For SSSP, the active elements are nodes. An operator applied to an active element may access elements in the graph besides the active element itself. For instance, in SSSP, an operator will scan the adjacent nodes of a node to check if a shorter path is available. The region of the graph accessed by an activity constitutes the neighborhood of the activity and may span beyond the immediate neighbors.

One dimension along which an operator can be classified is how the operator updates the graph. A *morph* computation changes the structure of the graph by adding or removing nodes and edges. Most graph analytics applications only update the labels on the nodes and edges of the graph, and do not modify the structure of the graph by adding/removing nodes and edges. Furthermore, the operator for these applications only accesses directly adjacent to the active elements. These operators, which access direct neighbors and do not modify the structure of the graph are also called *local computations*.

A local computation operator can be further classified according to the accesses it performs on the graph. If the operator reads from multiple nodes and only updates the label of the active node, it is known as a *pull* operator as it pulls updates from nodes adjacent to the active element. Conversely, a *push* operator writes to the nodes adjacent to the active element. Note that some algorithms may fall in both categories as they have a mixture of the characteristics of both operator styles.

Fig. 2.5 shows the key concepts in a graph application. The graph shown has circular nodes connected by edges. At any point during the execution, the operator may be applied to many of the active elements. In the figure, nodes circles colored *red* are candidates for execution. For each candidate, the shaded regions represents the neighborhood of the activity. As described earlier, the neighborhood of an activity is not necessarily adjacent to the active element.

2.3.2 Schedules

Given an operator, we can repeatedly apply it to all graph nodes until no nodes changes its label. For instance, in SSSP, we can repeatedly scan the nodes to see if a shorter path is available. This algorithm, also known as *Bellman-Ford*, computes the shortest path from a specified source in $O(|V||E|)$ steps. These algorithms are called *topology-driven* algorithms, as the topology of the graph drives the computations. While these are simple to implement, for many algorithms a topology-driven algorithm is very inefficient. Consider SSSP where initially all the nodes are at a distance *infinity*, and the source node is initialized to be at distance *zero*. For a topology-driven schedule, all the nodes execute the operator, but only a small subset of the nodes adjacent to the source node are updated. In the next round, nodes adjacent to those updated in the first step will be updated. This process will repeat until all the nodes have settled on their final version or some other termination criteria is reached (such as the maximum number of operator executions). For many graph applications, the fraction of nodes updated in each round is very small - leading to a large fraction of wasted work.

An alternate is to track the active elements and execute operators only at sites that potentially contain updates to be propagated. As described above, for SSSP, if the distance of a node is updated as the result of the application of an operator, it needs to propagate those updates to nodes adjacent to it - hence it becomes *active*. Initially, only the *source* is active, and the result of applying the operator to the source will activate adjacent nodes that update their distances. This reduces the wasted work significantly at the expense of tracking the active elements.

Another key property described by the schedule is whether the operator is *ordered* or not. Given a set of active elements, an ordered operator defines a strict order in which the active elements are to be executed. *Dijkstra's* algorithm is an ordered algorithm for SSSP where the active element with the least distance is to be executed first. This *ordered, data-driven* algorithm is the most work-efficient algorithm for SSSP. The Bellman-Ford algorithm for SSSP, as described above, is an example of an *unordered* algorithm as the nodes will be updated to their correct distance even if they are not performed in a specific order. However, it is a very inefficient algorithm.

2.4 Implementation Issues

In this section we briefly describe the different issues relevant to implementing graph algorithms. First, we describe a data structure to represent graphs. Next we describe different implementation strategies for implementing graph algorithms using this data structure.

We use *Single Source Shortest Path* (SSSP) to illustrate the issues that arise

in implementing graph analytics algorithms. In the SSSP application, a directed graph represents a collection of nodes which are connected through some weighted edges. Each node maintains a label $dist$ representing its distance from a start node, initialized to *infinity*. The goal of the algorithm is to compute the shortest distance of every node from a given start node. This can be achieved through different algorithms as described in the Sec. 2.4.2 and Sec. 2.4.3.

2.4.1 Graph representation

One of the challenges of implementing a graph application is the choice of representation. While the node or edge labels can be represented by arrays indexed by the node or edge ID, the topology of the graph can be specialized. We can start with a naive adjacency matrix representation where we store the topology of the graph in a boolean matrix M of dimension $|V| \times |V|$. A node a is adjacent to b iff the entry $M[a][b]$ is *true*. While this representation provides efficient access to the adjacency information, it can waste memory for sparse graph. Most real world graphs are sparse, meaning $|E| << |V|^2$. This means that a large fraction of the adjacency matrix will be *false*. If the operator has to go over the adjacent nodes of an active node, it has to scan the entire row of $|V|$ entries for any *true* entries. This results in not only wasted memory but also wasted computation.

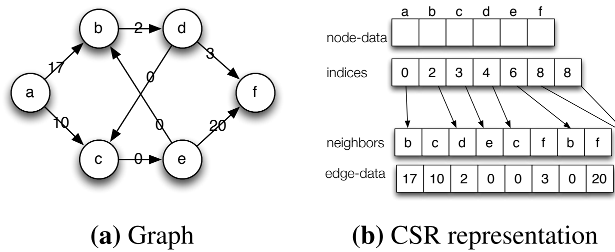
An alternate representation trades the simplicity of an adjacency matrix for efficiency in both memory and computation. The *compressed sparse row*(CSR) representation packs all the adjacency information of each node contiguously. This compression not only reduces the memory required to store the topology of the

graph but also makes it more efficient to traverse the adjacent nodes of an active element. A graph and its CSR representation are shown in Fig. 2.6. In the CSR representation, for a graph $G(V, E)$, where V is the set of nodes, and E is the set of edges, each node is given a unique number between 1 and $|V|$. The representation uses four arrays, whose role is described below.

- *node-data*: an array of size $|V|$, indexed by node number, that contains the label of each node.
- *indices*: an array of size $|V| + 1$ used to access the edges connected to a node. The elements between $indices[n]$ and $indices[n + 1] - 1$ in the *neighbors* and *edge-data* arrays below contain information about the edges connected to node n.
- *neighbors*: an array of size $|E|$ that stores the IDs of the out-neighbors of each node.
- *edge-data*: an array of size $|E|$ that stores the data on edges.

Edge or node labels can be omitted as necessary for applications; for example, Page-Rank does not require labels associated with edges. An undirected graph can be represented by storing edges in both directions. The pull-style SSSP algorithm needs the transpose of the hyper-link graph since each node needs to access fields in nodes that point to it.

While many other graph representations are good candidates for different algorithms and devices, a single representation is necessary, especially if the graph



(a) Graph **(b)** CSR representation

Figure 2.6: A directed graph and its CSR representation. Node e does not have any outgoing edges, so the indices entries for its edges point to the end of neighbors array.

is to be loaded from disk. The work in this dissertation assumes that the graph is represented in a CSR representation, unless otherwise specified.

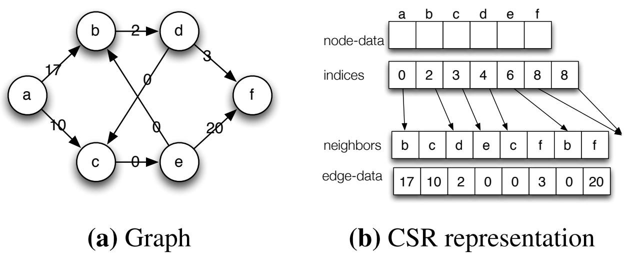
2.4.2 Pull Implementations

We first describe a pull algorithm for computing the single source shortest path for a graph. Fig. 2.9 shows a sequential implementation of SSSP using the CSR representation. Each node has the fields `dist` that store the distance of the node from the start node.

The algorithm operates in rounds. In each round, every node n is visited, its incoming neighbors are scanned to determine if a shorter distance is available by summing the distance of the incoming neighbor and the weight of the edge connecting n to it. If a shorter distance is available, the label on n is updated to the shorter distance. This is known as the *Bellman-Ford* algorithm for Single Source Shortest Path, and shown in Fig. 2.7. This algorithm is a *topology-driven algorithm* because each round visits all the nodes, and the computation at each node is said to

```
1 void sssp_pull(Graph g){
2    for(Node n : g.nodes()){
3       int min_dist=INT_MAX;
4       for(Edge e : n.in_edges()){
5          min_dist= min(e.source.dist + e.weight, min_dist);
6       }
7       n.dist = min(n.dist, min_dist);
8    }
9 }
```

Figure 2.7: Single Source Shortest Path solved through Bellman-Ford algorithm.



(a) Graph **(b)** CSR representation

Figure 2.8: A directed graph and its CSR representation. Node f does not have any outgoing edges, so the indices entries for its edges point to the end of neighbors array.

be a *pull-style operator* because the label at that node is updated by reading labels from the immediate neighbors.

There are four key memory accesses in this implementation.

- Line 3 – Accesses the data of the node currently being processed. This is a sequential access since nodes are accessed in sequence by the outer loop.
- Line 5 – Accesses the starting and ending locations of edges for the current node. This is also a sequential access since the indices array stores end-points sequentially.

```
1     void sssp_pull(Graph &g){
2     for(int i=0; i<g.n_nodes; ++i){
3        NodeData & src = g.node_data[i];
4        int min_dist=INT_MAX;
5        for(int e=g.indices[i];e!=g.indices[i+1]; ++e){
6           int dst_id = g.neighbors[e];
7           int weight = g.edge_data[e];
8           NodeData & dst = g.node_data[dst_id];
9           min_dist = min(min_dist, dst.dist + weight );
10       }//end for-edges
11       src.dist = min(src.dist, min_dist);
12    }//end for-nodes
13 }//end sssp
```

Figure 2.9: Implementation of Bellman-Ford for Single Source Shortest Path.

- Line 6 – Accesses the destination node of the edge currently being processed. This too is a sequential access since the neighbor array is accessed sequentially via e.

- Line 7 – Accesses the data associated with the destination of the edge currently being processed. This is an irregular access since the destination of the edge can be any node.

To get good performance for graph applications such as Single Source Shortest Path, each of these accesses must be performed efficiently. Implementations of the other graph algorithms are similar, except that the computation performed at nodes is different. For BFS for example, the labels $v_1, .., v_i$ of the neighbors are read, and the label of the node is set to the minimum of its current label and the values of v_1+1, ..., v_i+1.

2.4.3 Push Implementations

The algorithm described in Fig. 2.7 can be classified as a *pull*-algorithm. In such a graph algorithm, each node goes over its neighbors and pulls updates from them. These are relatively simpler to implement as there is only one writer per node. This obviates the need for synchronizations.Furthermore, these algorithms can be implemented efficiently in a bulk synchronous parallel manner on modern architectures.

The drawback of pull-style algorithms is that updates in the graph are not tracked - every node has to check its neighbors for any updates that might lead to an update in the node itself. Most graph algorithms such as *Breadth First Search* (BFS) and *Connected Components* converge when no node modifies its label in a give round. This can lead to a large waste in computation as each node scans its neighbors, and does not find any updates, even if there is only one node in the graph that was updated in the round. The extreme example is performing BFS on a linked list. Only one node is updated in each round, but all the node have to be scanned in every round.

An alternate implementation style is the *push*-style algorithm. Here, each node pushes updates to its neighbors. In the basic topology-driven implementation, all the node are repeatedly processed, and each node will send its update to its outgoing neighbors. Fig. 2.10 shows an implementation of SSSP using a push algorithm. The graph is loaded with the source node initialized to zero. Next, *sssp_push* is repeatedly applied to the graph. If any node is updated in a call to *sssp_push*, the node is updated, and a boolean flag *updated* is set to true, to indicate

```
1 bool updated = false;
2 void sssp_push(Graph g){
3    updated = false;
4    for(Node n : g.nodes()){
5       for(Edge e : n.out_edges()){
6          new_dist = n.dist+e.wt;
7          if(e.destination.dist>new_dist){
8             e.destination.dist= new_dist;
9             updated=true;
10         }//end if
11      }//end for-e
12   }//end for-n
13 }//end sssp
14 ...
15 Graph g(...);//load graph
16 do{
17    sssp_push(g);
18 }while(updated);
```

Figure 2.10: Single Source Shortest Path topology-driven push algorithm.

that at least one node was updated in the graph, and the changes will need to be updated further.

At first, this seems like a very inefficient implementation. There can be concurrent writes which requires synchronization between multiple writes to the same memory location (destination distance). For Fig. 2.10, the write in line 8 has to be execute atomically. However, we can observe that any update that needs to be propagated in the next rounds are those that set the *updated* flag to true. We can use this to track active node and only go over active node in every pass. Any node that had not been updated in the previous round has no new information that needs to be propagated to its neighbors. For instance, in SSSP, if the distance of a node has been lowered, it will be scheduled to push its distance to its outgoing neighbors in the next round. This is also known as a work-list driven or data-driven implementation since the data-values on the nodes and edges dictates which nodes are processed.

```
1 WorkList sssp_wl(Graph g, WorkList wl){
2    WorkList next_wl;
3    for(Node n : wl){
4       for(Edge e : n.out_edges()){
5          new_dist = n.dist+e.wt;
6          if(e.destination.dist>new_dist){
7             e.destination.dist= new_dist;
8             next_wl.push_back(e.destination);
9          }//end if
10       }//end for-e
11    }//end for-n
12 }//end sssp
13 Graph g(...);//load graph
14 WorkList wl(...);//initialize
15 while(!wl.empty()){
16    wl = sssp_wl(g,wl);
17 }
```

Figure 2.11: Single Source Shortest Path work-list driven push algorithm.

In contrast, a topology-driven algorithm has all the nodes nodes executed in every pass.

A data-driven implementation, as shown in Fig. 2.11 for SSSP, uses a work-list to track the active items. The graph is first loaded, and the distance of source node initialized to zero. The source node is also added to the work-list since it has updates that need to be propagated. Next, we call the *sssp_wl* procedure which goes over all the nodes in the work-list, and adds nodes to the next work-list if their distances have been updated. This is repeated until no node is updated. These algorithms are more work-efficient since they do not touch the whole graph in every pass - unless it is required. The drawback of this approach is the extra synchronization required, not only from having multiple writes being performed to each node from its neighbors, but also maintaining the work-list where active nodes for the next round are maintained.

2.5 Abstractions

Given a graph algorithm specification as described above, we would like to program it to execute on an accelerator. A simple approach is to understand the native instruction set of the accelerator and to translate the algorithm specification to it. Clearly, this approach is not productive for application developers looking to port applications to these accelerators. Alternatively, high level abstractions can be used to express the graph algorithms at a higher level. This requires a compiler and runtime to support the translation and execution of the application on the accelerator.

2.5.1 CUDA

Compute Unified Device Architecture(CUDA) is a proprietary application programmer interface developed by Nvidia. The language is designed to make general purpose GPU (GPGPU) programming more accessible by extending C++ with certain constructs. CUDA provides support for executing kernels on the GPU which are specified in an augmented subset of C. This makes it very attractive for application developers experienced with structured programming as kernels require minor modifications. CUDA also provides a comprehensive set of functions for management, memory, and event management. A large number of libraries have been developed to support CUDA which has added to its popularity in many domains. One key drawback to using CUDA is vendor lock-in – only devices from Nvidia currently support it.

2.5.2 OpenCL

In response to the growing popularity of CUDA, *Open Compute Language(OpenCL)* was developed by *Khronos group*, a consortium of companies. The OpenCL specification describes an API that different accelerators must support to obtain compliance. The original 1.0 of the standard was released in 2009. Similar to CUDA, OpenCL provides API for device, memory and event management. Programming in OpenCL is slightly more difficult compared to CUDA as the API is designed to support a larger range of devices. Currently, OpenCL is supported by a range of devices such as CPUs, GPUs, FPGAs, and DSPs.

Chapter 3

Data-parallel execution with Integrated GPUs [1]

Heterogeneous systems composed of integrated GPUs and multi-core CPUs are ubiquitous in commodity systems. In this chapter, we explore the efficient execution of irregular programs on such heterogeneous systems by first addressing the portability of existing applications to these platforms, and then describe runtime techniques to address load imbalance between the CPU and the GPU in a heterogeneous execution.

3.1 Introduction

Graphics processing units (GPUs) have become increasingly popular for accelerating general purpose computations. GPUs provide massive parallelism on a small energy budget and offer opportunities for significant energy savings and performance improvements compared to multi-core CPUs.

Integrated GPUs are manufactured onto the same die as the CPU, where they share resources like physical memory (and on Intel's integrated processors, the last-

[1] Portions from this chapter have been published in peer-reviewed conferences. The Concord compiler was presented in [Barik et al., 2014]. The scheduling strategies described have been published in [Kaleem et al., 2014]. The first author was responsible for implementation and evaluation of graph applications as well as conception, design and evaluation of heterogeneous schedule.

level cache). The ubiquity of integrated GPUs from major hardware vendors such as AMD and Intel has made these accelerators very accessible. The advantage of integrated GPUs is that they benefit from low-latency communication and eliminate most data copying, which significantly lowers the cost of offloading work to the GPU. However, integrated GPUs are limited by the power and size budget allocated for the die which is shared with the multi-core CPU.

This interest in GPU acceleration of applications has led to the development of specialized programming languages such as CUDA [NVIDIA, 2010], OpenCL [OpenCL, 2009], and, more recently, OpenACC [OpenACC, 2011] and Microsoft C++ AMP [C++AMP, 2015]. These specialized languages expose details of the GPU architecture and CPU/GPU communication model to the programmer. While they enable expert programmers to achieve high performance, their complexity and the architectural understanding required limits widespread use of GPU programming. One way to reduce the complexity of GPU programming is to use the same data-parallel programming models that are already used for programming multi-core CPUs. Recent work [Baskaran et al., 2010, O'Boyle et al., 2013] demonstrated the practicality of this approach for regular applications operating on array-based data structures.

The question, though, remains whether benefits of GPU execution can be extended to irregular applications written in an object-oriented programming style that features object references, virtual functions, and functor-based parallel constructs. To address the irregularity in data-accesses, especially through pointer-based codes, the programmer can used the shared physical memory between the

CPU and the integrated GPU. This however, requires virtual addresses in both device-spaces to map to the same physical address.

This chapter describes a compiler and runtime combination to efficiently utilize a heterogeneous systems composed of a multi-core CPU and an integrated GPU for irregular application. First in Sec. 3.2, we describe Concord, a compiler that can support a large set of C++ programming constructs on the GPU. We then describe strategies to dynamically balance the workload between the CPU and the integrated GPU to speed-up execution of applications in Sec. 3.3, Sec. 3.4, and Sec. 3.5. We evaluate the different schemes for heterogeneous execution in Sec. 3.6, and conclude in Sec. 3.7.

3.2 Compiler

We first describe Concord, a compiler for translating C++ code to execute on the integrated GPU. As Concord is targeted towards irregular applications, it supports most C++ features with some exceptions. The features supported include classes, virtual functions, multiple inheritance, operator and function overloading, templates, and namespaces. However, due to compiler and GPU hardware limitations, there are restrictions to its C++ support, violations of which result in compile-time warnings and parallel code being executed only on the CPU. In particular, Concord does not support recursion (except for tail-recursion that can be eliminated at the compile time), function calls via a function pointer, taking the address of a local variable, memory allocation on GPU, and exceptions. It provides two API functions for data-parallel iteration and reduction, and has shared virtual memory

(SVM) support that enables programs to transparently share pointer-containing data structures.

3.2.1 Programming constructs

Concord's two template API functions for data-parallel computation are modeled after the corresponding ones in Intel Threading Building Blocks (TBB)[TBB (Intel Threading Building Blocks), 2011], OpenMP[Dagum and Menon, 1998], and Cilk[Leiserson, 2009].

```
1 template <class Body>
2 void parallel_for_hetero(int n, const Body &b, bool on CPU);
3
4 template <class Body>
5 void parallel_reduce_hetero(int n, const Body &b, bool on CPU);
```

Figure 3.1: Concord programming constructs.

Both template functions take a parameter n that specifies the iteration space, $[0..n-1]$ to be executed in parallel. For both functions, the second parameter b must be an instance of a class $Body$ that defines a function call $operator()$ specifying the body of the parallel loop or reduction. The third parameter controls whether execution should be on the CPU or GPU. For parallel_reduce_hetero, the $Body$ class must define an additional method join to combine the results for two Body objects. Concord does not guarantee that the different loop iterations will be executed in parallel. Also, as in TBB, programmers should make no assumption about the order in which different iterations are done. Similarly, floating point determinism in reductions is not guaranteed.

Fig. 3.2 shows an example Concord C++ program demonstrating the use

```
// Functor class implementing one loop body
class Foo {
  int *a;
  void operator(int i) { a[i] += f(i); }
};
// Functor class implementing another loop body
class Bar {
  int *a;
  void operator(int i) { a[i] -= g(i);}
};
...
Foo *f1 = new Foo();
// Single invocation of a data-parallel loop
parallel_for(N, *f1);
...
Bar *f2 = new Bar();
// Multiple invocations of a data-parallel loop
for (int i=M; i<P; i++)
   parallel_for(i, *f2);
```

Figure 3.2: Example demonstrating use of Concord $parallel_for$ construct.

of the *parallel_for* construct. The first *parallel_for* increments every element of an N-element array a using the data-parallel kernel in *Foo::operator* once (single invocation kernel), while the later loop decrements the elements of a using the *Bar::operator* kernel multiple times (multiple invocation kernel). The classes *Foo* and *Bar* contain the environment (e.g., a) for the parallel code specified in the *operator* functions.

Fig. 3.3 depicts the components of the Concord framework along with their interaction with other components. Concord builds on the CLANG and LLVM infrastructure to compile Concord C++ programs. A compiler pass identifies the loop body functions (i.e., the $operator()$ and $join$ methods of a body class) and generates CPU code as well as GPU OpenCL kernel code for them. Concord generates a host-side executable that embeds the generated OpenCL. To execute the parallel loop, the runtime extracts its OpenCL code, compiles it just-in-time to the GPU

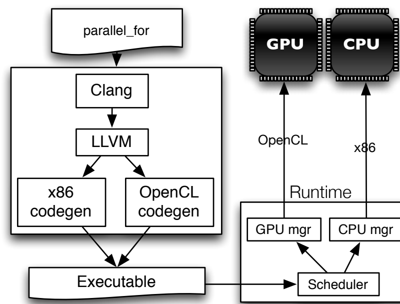


Figure 3.3: Concord compiler and runtime overview.

ISA if necessary via the vendor-specific OpenCL compiler, and then, based on the on_CPU flag, decides whether to execute it on the CPU or GPU.

3.2.2 Shared Virtual Memory (SVM) support

To make it easier to run existing C++ programs on an integrated processor, Concord provides software based shared virtual memory (SVM). This allows programs running on the CPU and GPU to directly share complex, pointer-containing data structures such as trees and linked lists. SVM also eliminates the need to marshal data between the CPU and GPU. The challenge of implementing this translation is that the CPU and GPU may have separate virtual-to-physical mappings and different pointer representations. These details differ greatly from one processor architecture to the next. The remainder of this chapter assumes that the Intel's 4th Generation Core (Haswell) processor is used. On this processor, the GPU and CPU use separate page tables. The GPU's virtual address space is segmented into surfaces and each surface is referenced by a binding table entry. A GPU pointer is represented as a binding table index plus an offset. To access memory, the offset is

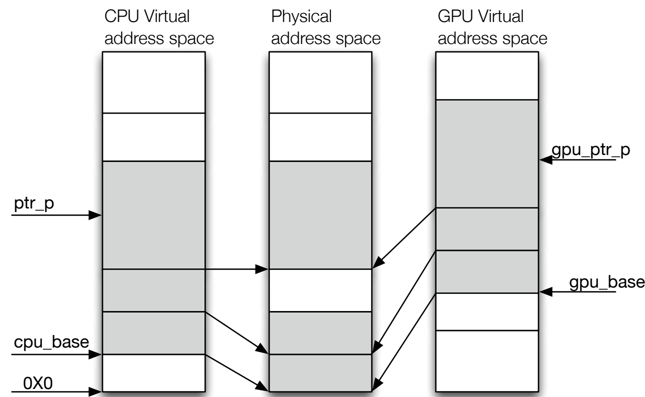


Figure 3.4: Address translation from CPU virtual address space to GPU virtual address space.

added to the surface's base address obtained by looking up that surface's binding table entry. Thus, when a shared pointer is dereferenced on the GPU, the CPU virtual address must be translated so that it refers to the same physical memory location on both GPU and CPU.

To perform the translation, a virtual memory region is created at program startup that is shared between the CPU and GPU. Any shared pointer that the GPU needs to dereference must be allocated in this shared memory region. This is achieved by redirecting $malloc$ and $free$ to specialized routines that allocate and free memory in the shared memory region. The shared memory region is pinned during GPU kernel execution and has a backing GPU surface with a binding table entry that is constant during runtime. This approach substantially reduces the cost of Concord's shared pointer translation.

Fig. 3.4 depicts the compiler transformation necessary to synchronize the virtual addresses of shared pointers between CPU and GPU. Given the base ad-

```
1 class Arr2List{
2   Node * array;
3   public:
4     Arr2List(Node * a):array(a){}
5     void operator()(int i){
6       array[i].next = &(array[i+1]);
7     }
8   };
9   void convert(Node * a, int N){
10    Arr2List * func = new Arr2List(a);
11    parallel_for_hetero(N,func,false);
12  }
```

```
1 typedef unsigned long CPUPtr;
2 #define AS_GPU_PTR(T,p) \\
3           (__global T*)(&svm_const[(p)])
4 __kernel void operator_1(
5             __global char *gpu_base,
6             __global char *cpu_base,
7             CPUPtr cpu_ptr){
8   uint i = get_global_id(0);
9   __global char * svm_const
10                = (gpu_base-cpu_base);
11  __global Node * gpu_ptr
12                = AS_GPU_PTR(Node,cpu_ptr);
13  *(AS_GPU_PTR(Node,gpu_ptr[i].next))
14                =&gpu_ptr[i+1];
15 }
```

Figure 3.5: Example demonstrating Concord SVM implementation. Left side shows the C++ implementation provided by the programmer. Right side shows the transformed code including translation code to convert from CPU virtual addresses to GPU virtual addresses.

dresses of CPU and GPU for the shared region as cpu_base and gpu_base respectively, a pointer ptr_p in the CPU virtual address space has a corresponding GPU virtual address gpu_ptr_p where gpu_ptr_p = $gpu_base+(ptr_p-cpu_base)$. This address translation can be optimized by using the runtime constant svm_const = $gpu_base-cpu_base$ that is computed only once. Then, before dereferencing ptr_p on the GPU, it can be translated to gpu_ptr_p by simply adding the runtime constant svm_const. Sec. 3.2.5.1 describes how we further optimize away part of this translation. The right hand side of Fig. 3.5 presents the compiler generated OpenCL code for the $operator()$ function using the pointer transformation described in this section. The OpenCL kernel $operator_1$ takes additional arguments for gpu_base, cpu_base, and the pointer cpu_ptr to the $Body$ object (which is same as b in the source program). The shared pointers, cpu_ptr and gpu_ptr $p[i]$.next are translated from the CPU address space to the GPU address space using the AS_GPU_PTR

macro. Our pointer translation technique can be generalized to scenarios where CPU and GPU use different encoding schemes and lengths. For example, if CPU memory is addressed using $64-$bits and GPU memory uses $32-$bits, we can apply the same pointer arithmetic as long as the shared region does not exceed 4GB.

3.2.3 Virtual Functions

One of the most widely used dynamic features of C++ is its virtual function support. Although there are a variety of different ways to implement virtual functions, the vtable (virtual table) approach is common in modern C++ compilers. In this approach, a compiler creates a separate vtable for each class and, when creating an instance of that class (an object), adds to that object a pointer to the class's vtable. A call to a virtual function is then handled by dereferencing the underlying runtime object's vtable pointer, locating the corresponding virtual function entry and finally dereferencing that pointer to call the function. To implement virtual functions on the GPU, vtables need to be allocated in the shared region and, more importantly, function pointers are required on the GPU. Current integrated GPU hardware designs are not yet capable of supporting function pointers, so we use a compiler-based solution. To support virtual functions on the GPU, the Concord compiler implements three key operations: a) move necessary vtables and runtime-type information to the shared region; b) share the global symbols of relevant virtual functions between the CPU and GPU using shared memory; c) translate a virtual function call into an inline sequence of tests of the call target against the possible target function pointer values for that call. The compiler implements global symbol

sharing between CPU and GPU by allocating a new structure in the shared memory region that encapsulates all global symbols needed for the virtual function calls executed by a GPU function. It also determines the set of call targets for a given virtual function using class hierarchy analysis and alias analysis.

3.2.4 Reduction

When using parallel_reduce_hetero, the $Body$ object's $join$ method contains reduction code that combines two $Body$ objects. Concord performs hierarchical reduction of the body objects on the GPU using local memory, the high-speed on-GPU memory that is shared among all work-items of a work-group in OpenCL. Concord generates OpenCL code for the join method similar to the code generation technique described in Fig. 3.5. We generate additional wrapper OpenCL code that makes multiple copies of the shared $Body$ object in each thread's private memory, invokes the $operator()$ function to compute the thread's value that participates in reduction, moves the private objects to local memory, and finally, iteratively performs reduction using local memory until a single value is left. The local memory copies hold intermediate reduction results. The final reduced value is copied back to the original shared Body object. The original sequential join function pointer is also passed to the runtime to perform sequential reduction if local memory is insufficient or if the GPU is busy.

3.2.5 Compiler Optimizations

The large register files offered by GPU architectures must be utilized by the compiler to improve performance. Classical compiler optimizations such as sub-expression elimination, aggressive register promotion, and loop-unrolling must be applied in order to exploit the register files. Given that shared pointers incur some overhead during translation, register promotion should be applied aggressively to eliminate memory loads of the same location, in particular, across loop iterations. Since loop-unrolling eliminates the overhead of address calculation and control instructions, we perform unrolling and control the unroll-factor by restricting max live to the available physical registers. Currently, our compiler promotes stack-allocated objects and reduction-based private copies of the body objects to private memory. These objects are not shared across threads since each thread creates its own instance. We also use local memory for performing reductions. Although it may make sense to also use local memory for C++ applications that reuse data among several GPU threads, the irregularity in the applications makes it harder to perform it automatically in the compiler. Additionally, a language-based approach to support local memory in C++ has the disadvantage that the same C++ function cannot execute seamlessly between the CPU and GPU which is one of the key objectives of Concord. Apart from the above standard compiler optimization techniques, we devise two new optimizations in Concord. The first optimization reduces the S/W-based SVM implementation overheads and the second optimization reduces cache contention among multiple cores of the GPU. These two compiler optimizations are described in details below.

3.2.5.1 Reduce SVM implementation overhead

The pointer arithmetic operations inserted as described in Sec. 3.2.2 must be minimized by the compiler whenever possible. Depending on how shared pointers are used on the GPU, it may be beneficial to retain the CPU virtual address representation for a shared pointer instead of eagerly translating it to GPU address space. For example, if the GPU code loads a shared pointer and stores it into a memory location without dereferencing it, then it is better to never convert the CPU virtual address. Similarly, there are some situations where it is better to eagerly translate CPU to GPU addresses, and others where lazy translation is better. For example, consider the code sample shown in Fig. 3.6.

```
1 int ** a = data->a, **b=data->b;
2 for ( int i=0; i<N; ++i)
3    b[i]=a[i];
4 //a is not used on GPU after this
```

Figure 3.6: Example illustrating compiler transformation of shared pointers on GPU: lazy vs. eager.

In this code fragment, pointer $a[i]$ is loaded from memory and written into $b[i]$ at each iteration of the loop. With eager translation (i.e., convert to GPU virtual memory representation as soon as the pointer is loaded), we need pointer arithmetic operations to translate the array addresses a and b only immediately after their definitions, which are outside the for-loop. Using lazy translation (i.e., keep the CPU virtual memory representation as is and translate to GPU representation just before dereferencing it), we must add pointer arithmetic to translate a and b from the CPU to the GPU representation on every loop iteration. The eager approach is clearly beneficial in this case. On the other hand, eagerly converting the address of an array

element $a[i]$ to a GPU virtual address results in wasted work because $a[i]$ is never dereferenced on the GPU. It would convert all $a[i]$ pointers to GPU addresses only to immediately convert them back to CPU addresses in order to store them in array b. The lazy approach is preferable in this case. Both eager and lazy approaches have their advantages and disadvantages and can perform better or worse depending on the code patterns in a program. We devise a strategy where we keep both the CPU representation and GPU representation for every pointer. The GPU representation is obtained by converting the pointer eagerly when it is loaded from memory. If at a later use the pointer is stored into a memory location (as $a[i]$ in Fig. 3.6), we replace the use by the CPU representation. Otherwise, we use GPU representation. If a pointer is never dereferenced on the GPU, a standard dead code elimination pass eliminates the redundant conversion to GPU address space. We optimize the placement of GPU pointer conversion operations using standard live-range shrinking techniques used in optimal code motion[Knoop et al., 1994].

3.2.5.2 Reduce GPU cache-line contention

GPUs include a large number of hardware threads that execute concurrently and may access data from the global memory. GPU hardware typically coalesces global memory accesses from a workgroup to hide the latency of global memory access. Additionally, these accesses may be cached in the GPU cache hierarchy. The integrated GPUs use a unified L3 cache for all GPU cores to cache global memory accesses. This cache is not banked and thus suffers from contention among multiple GPU cores trying to access the same data in a cache line at the same time.

We devise a compilerbased transformation in which we minimize the number of simultaneous accesses to the same cache line from multiple GPU cores.

```
1 class Body{
2   float * d;
3   public:
4     Body(float * f):d(f){}
5     void operator()(int i){
6       ...
7       for(j=0;j<N;j++){
8         ... = d[j];
9       }
10    }
11 };
```

```
1 class Body{
2   float *d;
3   public:
4     Body(float *f):d(f){}
5     void operator()(int i){
6       ...
7       int start=i/N;
8       for(j=0; j<N; ++j){
9         j_tmp=(j+start)%N;
10        ...=a[j_tmp]
11      }
12    }
13
14 };
```

Figure 3.7: Reducing cache-line contention among GPU cores. Left side shows the original code.

Fig. 3.7 depicts our loop transformation to reduce GPU cache-line contention. The $operator()$ function on the left hand side has a loop that iterates over same array elements of a across multiple iterations of i. If iterations i and $i+1$ are executed on two separate GPU cores, then they will access the same array elements of a in the same order. This will result in increasing cache line contention. If the number of read and write ports to a cache line is not the same as the number of GPU cores (which is always the case), some cores will have to access the cache-line in a serialized fashion. On the other hand, the operator() function shown on the right hand side of Fig. 3.7 does not suffer from this problem. Note that, W represents the number of GPU cores. The key idea is to ensure that the j loop is accessed in a different order for each GPU core. We apply this transformation to innermost loops.

The Concord compiler translates parallel_for_hetero and parallel_reduce_-

hetero to the runtime API functions offload and offload_reduce respectively. These runtime functions take additional compiler-generated arguments: (1) a gpu_program_t structure for the entire program to hold the OpenCL code and its cached JIT-compiled GPU binary; (2) a gpu_function_t structure to cache per-function GPU binary code in order to reuse the JIT-compiled code. The gpu_function_t also carries the user specified device information per kernel as specified in the third argument of parallel_for_hetero and parallel_reduce_hetero.

3.3 Heterogeneous execution

We can extract more performance from this heterogeneous system comprising of a multi-core CPU and an integrated GPU by simultaneously executing the workload on the CPU and the GPU. This requires a systematic way of dividing the work between the multi-core CPU and the integrated GPU. However, the optimal division of work between the CPU and GPU is very application dependent. The CPU and GPU have different device characteristics. CPU cores are typically out-of-order, have sophisticated branch predictors, and use deep cache hierarchies to reduce memory access latency. GPU cores are typically in-order, spend their transistors on a large number of ALUs, and hide memory latency by switching between hardware threads. This dissimilarity leads to significant differences in execution performance. Certain applications may execute significantly faster on one device than the other. As a result, executing even a small amount of work on the slower device may hurt performance.

As an example, consider Fig. 3.8, which shows the performance of Barnes-

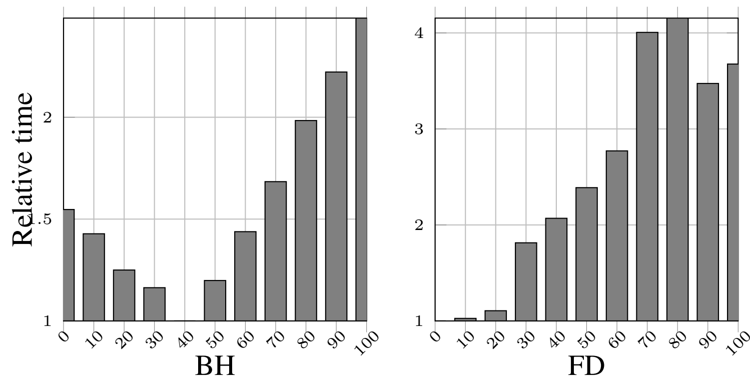


Figure 3.8: Relative execution time of **BH**-BarnesHut (left) and **FD**-Facedetect (right) as ratio of work offloaded to the GPU is varied from 0% to 100% in increments of 10% (*lower is better*). **BH** is optimal at 40% and **FD** is optimal at 0%.

Hut (**BH**) and Face detection (**FD**) on an Intel Haswell machine as the proportion of work assigned to the GPU is varied from 0% to 100% (the applications and the machine are described in more detail in Sec. 3.6). The y-axis in both graphs shows execution time normalized to the best running time for each application. For **BH**, the best running time is obtained when 40% of the work is done on the GPU. **FD** in contrast does not benefit at all from GPU execution and performs best when run on multicore CPU.

Furthermore, certain *parallel_for* iterations may take more time than others. Without a-priori information about the loop's behavior, it is hard to optimally divide work between the CPU and GPU automatically. Iterations may show execution irregularity due to data-dependent control flow operations for example. Without knowledge of the input data, it is hard to determine the optimal partitioning. There can be more than one *parallel_for* in an application and each may be invoked more than once, as in the second *parallel_for* in Fig. 3.2. Since one parallel loop can

have the side effect of warming the cache for a second loop, understanding the interactions among the different parallel loops is important to optimally partition work. Similarly, interactions between the multiple invocations of the same parallel loop can also be important for optimal work scheduling.

As a result, devising an automatic heterogeneous scheduling algorithm to handle all these situations is a daunting task.

Ideally, the scheduler should partition work between CPU and GPU automatically and efficiently without any input from the application developer. The division of work between the CPU and a GPU has been the subject of a number of prior studies [Luk et al., 2009, Augonnet et al., 2011, Lee et al., 2013, O'Boyle et al., 2013, Sbîrlea et al., 2012, Chatterjee et al., 2011]. Their techniques fall into three broad categories:

- *Off-line training* [Luk et al., 2009, Lee et al., 2013, O'Boyle et al., 2013]: The application is first run using a training data set and profiled. The profiling data is used to select the scheduling policy for subsequent application runs against real data. *Qilin* [Luk et al., 2009] performs an off-line analysis to measure the kernel's execution rate on each device (CPU and GPU). These rates are used to decide the distribution of work for each device. Qilin uses a linear performance model to chose the scheduling policy based on the size of the input data set. In general, there are two main drawbacks of off-line profiling. *First*, the scheduling policy chosen depends on the training data, i.e., if the training data differs significantly from the actual data used in subsequent runs,

the scheduling policy is likely to be suboptimal. *Second*, new execution rates and work distributions must be determined for each new target platform.

- Use a *performance model* [Augonnet et al., 2011, Hong and Kim, 2010]: Accurate performance models are notoriously difficult to construct, particularly for irregular workloads since runtime behavior is very dependent on characteristics of the input data.

- Extend standard *work-stealing*[Blumofe and Leiserson, 1999] with restrictions on stealing [Chatterjee et al., 2011, Sbîrlea et al., 2012]: Work-stealing is a popular way to address load imbalance in multi-core execution. However, a GPU differs from the CPU in one important aspect when addressing load imbalance. Current GPUs cannot initiate communication with the CPU or execute atomic operations visible to the CPU which means a GPU cannot request work from non-local work pools. The GPU cannot use persistent threads to steal work from a common pool because of the relaxed CPU-GPU memory consistency, which only guarantees that GPU memory updates will be visible after the GPU code terminates. Furthermore, the GPU is statically scheduled: once items are scheduled to be processed on the GPU, the order in which they are processed is undefined and hence we can make no assumptions about which items can be stolen from the GPU's local pool by other threads.

One simple approach to addressing load imbalance that avoids these limitations is to have a dedicated *GPU proxy thread* running on the CPU to per-

form work-stealing on behalf of the GPU. The GPU proxy thread can steal a number of loop iterations to execute on the GPU, then wait for the GPU to complete their execution or until all iterations have been processed by either the CPU or the GPU. The time to execute a number of loop iterations on the GPU is a key performance bottleneck. If we choose many iterations in order to reduce the overhead of launching work on the GPU, the GPU might stall loop termination while the CPU threads are out of work. On the other hand, if we choose too few iterations, this will increase the number of GPU offload requests as well as underutilize the GPU hardware, as we show later in Sec. 3.4.2.

- **Profiling-based** is an online profiling based approach. The application kernel is dynamically profiled and used to determine the distribution of workload between the CPU and the GPU. The key to this approach is that the profiling phase should not introduce any overhead that can not be compensated by the benefits obtained by doing it. We present two online-profiling based techniques in the next two sections and analyze their overheads.

Most of the prior work has been performed in the context of discrete GPUs, where data must be communicated between CPU memory and GPU memory over slow interconnect such as the PCIe bus. This high-latency communication limits the CPU to offload relatively coarse-grain tasks to the GPU. Integrated GPUs dramatically reduce the cost of data communication between CPU and GPU, so finer grain work sharing between the CPU and GPU than has been explored by prior

work becomes possible.

In this chapter, we present novel scheduling techniques for integrated CPU-GPU processors that leverage online profiling. Our techniques profile a fraction of the work-items on each device and decide how to distribute the workload between the CPU and GPU based on the measured device execution rates. Because our algorithm is fully online, it does not require any prior training and carries no additional overhead when applied to applications with new data sets or new platforms.

While seemingly simple, scheduling based on online profiling must avoid several pitfalls to produce good results. *First*, it should perform the profiling with near zero overhead, effectively utilizing all available resources, as otherwise, the profiling cost might significantly reduce the benefit of subsequent heterogeneous execution. *Second*, it should accurately measure the execution ratio of different devices, which might be non-trivial in the presence of load imbalance that often occurs in irregular applications. *Finally*, it should be able to effectively handle diverse realistic workloads including those with multiple kernels and multiple invocations of the same kernel, accounting for the fact that optimal execution might require different CPU/GPU partitioning of the different kernel invocations or different kernels.

3.4 Naïve profiling

In this section, we propose a heterogeneous scheduling algorithm based on a simple online profiling scheme and analyze the associated overheads.

It is well-known that regular applications with little or no load imbalance are well-suited for GPUs whereas multi-core CPUs equipped with accurate branch predictors are capable of performing well across regular or irregular, balanced or imbalanced workloads. As a result, we first determine the characteristics of a workload using an online profiling run in order to determine the rate of execution on each device. This profiling information is used to decide the percentage of remaining iterations to assign to each device.

Fig. 3.9 shows our naïve profiling scheduling algorithm. It executes in two phases – *profiling phase* and *execution phase*. When a kernel is executed for the first time over N items, the runtime forks a proxy thread that offloads Nf_p iterations to the GPU, where f_p is a ratio between 0 and 1[2], and measures the rate G_r at which the GPU processes the kernel. Concurrently, the runtime thread itself processes another Nf_p iterations on the multi-core CPU and computes the CPU rate C_r. Both threads merge on a barrier after completing the profiling phase where they compare the two rates G_r and C_r. Based on these rates, the runtime distributes the remaining iterations to the CPU and GPU in the execution phase. New rates are computed and cached for future invocations of the same kernel. For already-seen kernels, the runtime uses the old values of C_r and G_r to distribute iterations.

The naïve profiling based scheduling algorithm may introduce overheads in both the phases: (1) in the *profiling phase*, one of the devices completes execution before the other one; (2) if the iterations used in the profiling phase are not

[2]We discuss choosing f_p in Sec. 3.4.2.

representative of the entire iteration space (applications with imbalance and irregularity fall into this category), then the *execution phase* may introduce some more overheads. We address both these overheads with an asymmetric scheduling algorithm (Sec. 3.5). Below, we first provide theoretical analysis of our naïve profiling algorithm and then discuss how f_p is determined.

3.4.1 Analysis

Let α_g denote the ideal ratio for distribution to the GPU, and the remaining $1 - \alpha_g$ to the CPU. If we denote the rate with which CPU executes work items as C_r and GPU as G_r, we would like to find the ratio α_g of work items to be offloaded to the GPU such that:

$$\frac{N\alpha_g}{G_r} = \frac{N(1-\alpha_g)}{C_r} \tag{3.4.1}$$

Where N is the total number of work items to be processed due to a call to the *parallel_for*. Using this formula, we can derive the value of α_g as:

$$\alpha_g = \frac{G_r}{G_r + C_r}$$

Intuitively, α_g specifies how fast the GPU is compared to the CPU. So if the two devices are processing items at the same rate ($C_r = G_r$), we would have $\alpha_g = 0.5$.

The ideal time to execute the N work-items can be expressed as:

$$T_{ideal} = \frac{N}{C_r + G_r} \tag{3.4.2}$$

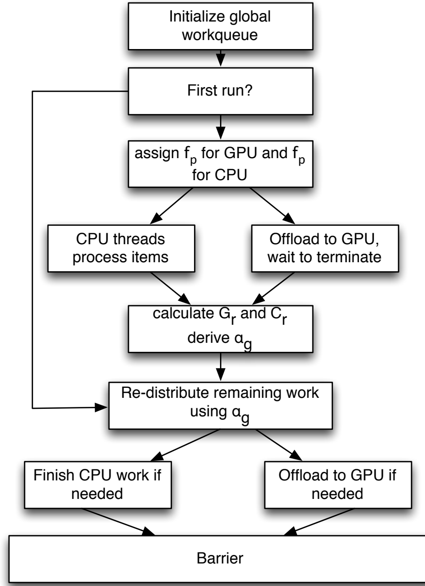


Figure 3.9: Heterogeneous execution via Naïve profiling.

Real execution time will consist of the T_{prof}, time to profile Nf_p work on CPU and GPU, and T_{exec}, time to execute the remaining items according to the measured distribution ratio β_g.

$$T_{real} = T_{prof} + T_{exec} \tag{3.4.3}$$

The distribution ratio β_g computed by the profiling may differ from the ideal distribution ratio α_g that allows both devices to terminate at the same time. This could happen, for example, if the workload is highly irregular or if the profiling stage is too short. How much the execution time T_{exec} differs from the ideal time to execute $(1 - 2f_p)N$ remaining items depends on the difference between α_g and β_g, as stated in the theorem below. Ideally, we would like profiling to derive β_g as

close to α_g as possible.

Theorem 3.4.1. *The overhead of executing P iterations using distribution ratio β_g instead of the ideal ratio α_g is proportional to the difference between β_g and α_g.*

Proof. The overhead is the difference of execution times with β_g and α_g.

$$\begin{aligned} T_{overhead} &= max[\frac{N\beta_g}{G_r} - \frac{N\alpha_g}{G_r}, \frac{N(1-\beta_g)}{C_r} - \frac{N(1-\alpha_g)}{C_r}] \\ &= max[N(\frac{\beta_g - \alpha_g}{G_r}), N(\frac{\alpha_g - \beta_g}{C_r})] \end{aligned} \tag{3.4.4}$$

The above difference is clearly proportional to $\beta_g - \alpha_g$. □

Another source of overhead with the naïve profiling algorithm is a suboptimal distribution of work during the profiling phase. The profiling time is the maximum of time to execute Nf_p items on each of CPU and GPU.

$$T_{prof} = max[\frac{Nf_p}{G_r}, \frac{Nf_p}{C_r}] \tag{3.4.5}$$

The profiling phase can impose significant overhead if the difference between G_r and C_r is very large. For example, assume that the GPU is 10x faster than the CPU ($G_r = 10C_r$). In this case,

$$T_{prof} = max[\frac{Nf_p}{10C_r}, \frac{Nf_p}{C_r}] = \frac{Nf_p}{C_r}$$

If we assume $f_p = 5\%$, the time taken by the profiling phase becomes $\frac{0.05N}{C_r}$. As the GPU is 10x times faster than the CPU, it completes its profiling run in one tenth of the time. If the GPU would continue working during the rest of the

profiling run it could complete nine times as much work, that is, another 45% of the total work. Even if the profiling is accurate, and the remaining work is divided between the CPU and GPU using the ideal distribution ratio α_g = 0.9, the time the GPU has spent idle during the profiling stage will have significant negative effect on total execution time.

3.4.2 Determining profiling size

Determining the right number of iterations or work-items to use for profiling is important. Picking *too many* items for the profiling phase can increase the profiling overhead since one device may have to wait for the other device to complete execution. On the other hand, picking *too few* work items for the GPU can underestimate GPU performance since it is underutilized. Ideally, the profiling size should be large enough to utilize all the available GPU parallelism. To illustrate this point, we demonstrate the impact of different profiling sizes on two kernels: one regular kernel and the other irregular.

Regular kernel: Fig. 3.10 shows the impact of choosing different profiling sizes on the performance of a simple synthetic kernel running on the integrated GPU (details of our experimental platform are provided in Sec. 3.6). This kernel computes the sum of integers from 1 to 2048 per work-item. There are no memory accesses and no thread divergences in the kernel. We report the GPU execution rate, G_r, as we increase the number of work-items offloaded to the GPU. As we increase the number of items, we see an increase in G_r as the GPU takes the same amount of time to process more items. The rate stabilizes at 2048 items, which implies that

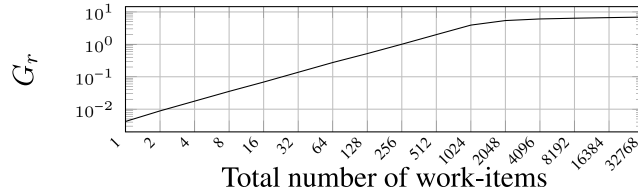


Figure 3.10: Plot showing the change in G_r with increasing number of work-items for a regular kernel (note the logarithmic scales) .

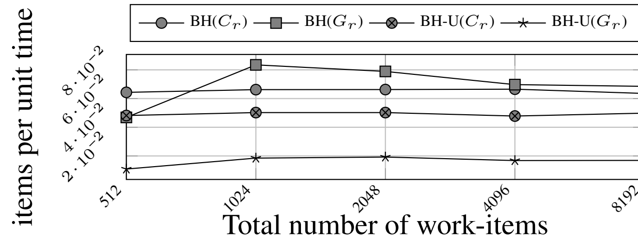


Figure 3.11: Plot showing average C_r and G_r for different values of Nf_p for BarnesHut (**BH**) and BarnesHut-unoptimized (**BH-U**).

the GPU hardware was underutilized when processing less than 2048 items. Ideally, we would like to have at least 2048 items to profile accurately and not underutilize this specific GPU.

Irregular kernel: Irregular kernel code may complicate the choice of a good profiling chunk size. Fig. 3.11 shows the average execution rates for both **BH-U** and **BH**[3] when we process items in chunk sizes ranging from 512 to 8192. We show the execution rates separately for the CPU (C_r) and GPU(G_r). The CPU performance is relatively stable for both **BH** and **BH-U** (the difference in time per item for the two versions is due to the improved locality with **BH**), and not affected

[3]**BH** is an optimized version of Barnes-Hut algorithm (**BH-U**), where the bodies are sorted in a breadth-first-search order from the root.

by increasing chunk size. However, for **BH-U** on the GPU, G_r increases sharply from 512 to 1024 and is relatively stable for larger values. A similar but less subtle trend is visible for **BH** on the GPU as well.

The impact of picking a small chunk size for GPU profiling is visible If we pick a chunk size of 512 for **BH-U**, we get the GPU rate $\frac{1}{92}$, and the CPU rate $\frac{1}{21}$, which leads to an offload ratio α_g of 0.17. However, if we pick 1024 as the chunk size, we get a different rate for the GPU $\frac{1}{54}$ and for the CPU $\frac{1}{19}$ leading to an GPU offload ratio of 0.27, which is close to the static optimal 0.30.

To summarize, we want to use the smallest number of items that fully utilize the GPU. The integrated GPU we use in our experimental evaluation has 20 execution units (EU), 7 threads per EU, each thread being 16-way SIMD for a total of 2240 work-items that can execute at one time. This information can be obtained automatically by querying the GPU device using the OpenCL API[4]. So, we use 2048 as the profiling size for our evaluation which agrees with our empirical observation above for both regular and irregular kernels.

3.5 Asymmetric profiling

In this section, we describe our asymmetric profiling algorithm that addresses the overheads of the naïve profiling algorithm and additionally, provides adaptive strategies that handle load imbalance due to irregularity and multiple invocations per kernel.

[4] CL_DEVICE_MAX_COMPUTE_UNITS, CL_DEVICE_MAX_WORK_GROUP_SIZE, and CL_DEVICE_NATIVE_VECTOR_WIDTH_INT.

The first overhead in naïve profiling was caused by waiting on a barrier when one of the devices finishes execution before the other. We address this overhead in asymmetric profiling as follows: we initially have a shared pool of work consisting of the entire parallel iteration space, and we pick a portion f_p of items to be offloaded to the GPU using the proxy GPU thread. The CPU workers continue to pick items from the shared global pool and locally collect profiling information. When the GPU proxy thread finishes profiling, it performs the following steps in order; it (1) computes the distribution ratio by reading each worker's local profiling statistics (during this time, the CPU workers continue to work on the shared pool), (2) empties the shared pool, (3) adds the CPU portion of the work to one of the CPU worker's work-stealing queue, and (4) finally offloads GPU portion of the work to GPU. Fig. 3.12 illustrates the algorithm. It is important to note that while the GPU is executing, CPU workers continue to work from the shared pool, thereby eliminating the overhead seen in naïve profiling.

One of the design choices for asymmetric scheduling is to start with a shared pool in the profiling phase instead of work-stealing. This is justified since at the beginning we have no knowledge of how to partition work among CPU workers and the proxy GPU worker. For example, for applications with irregularities, it may be costly to partition the work up front into the work-stealing queues.

If both the devices are kept busy in the profiling phase by having sufficient number of parallel iterations, it can be seen that the profiling phase will introduce zero overhead in the asymmetric profiling algorithm (theoretical analysis is provided below in Sec. 3.5.1). Similar to the naïve profiling, the vanilla version of

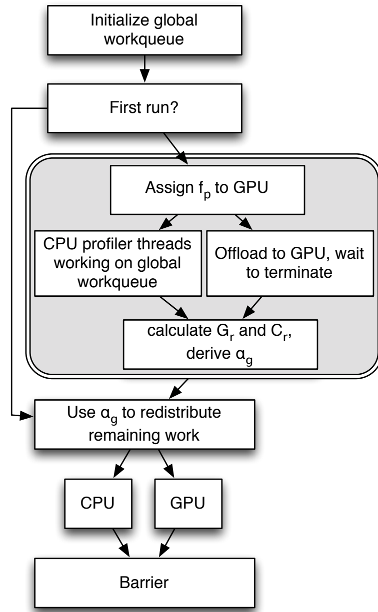


Figure 3.12: Heterogeneous execution through symmetric profiling.

asymmetric profiling may still introduce overhead in the execution phase if the iterations used in profiling phase are not representative of the entire iteration space. We address this in Sec. 3.5.2.

3.5.1 Analysis

We now analyze the overhead of asymmetric profiling compared to the ideal execution scenario. We concentrate on the analysis of the profiling phase, as the behavior of the execution phase in the base algorithm is identical to that of naïve profiling. Assuming the runtime offloads f_p fraction of the original items to the GPU and allows CPU threads to execute as long as the GPU is busy, the time taken by the GPU to complete Nf_p items is $t_p = \frac{Nf_p}{G_r}$. At the same time, CPU completes

an additional $t_p C_r$ items assuming there is enough work to keep it busy. In this case, the overhead of the profiling is zero, as both GPU and CPU work all the time. On the other hand, if the amount of work offloaded to GPU is so large that there is not enough work items for CPU, there is a non-zero overhead. This observation is formally stated by the following theorem.

Theorem 3.5.1. *Overhead of the profiling phase in the asymmetric profiling algorithm is zero if* $f_p < \alpha_g$.

Proof. The number of items executed by CPU is maximum of the amount of work it can perform in t_p time and the remaining work:

$$N_c = max(t_p C_r, (1 - f_p)N)$$

The first argument to max is greater than the second, when $t_p C_r > (1 - f_p)N$, which simplifies to $f_p > \alpha_g$. In this case, the total amount of work executed by CPU and GPU is $N_p = N f_p + \frac{N f_p C_r}{G_r}$. In the ideal case, this work can be executed by CPU and GPU working together in $\frac{N_p}{C_r + G_r}$ time. This time is equal to the profiling phase time $\frac{N f_p}{G_r}$ as can be shown by simple algebraic transformations. The overhead of the profiling phase is, thus, zero. □

In practice, f_p is likely to be greater than α_g only when CPU is significantly faster than GPU. For example, if we offload 5% of work to GPU for profiling, CPU should be at least 19x faster than GPU to trigger non-zero overhead. To handle such cases our work can be combined with static profiling techniques such as [O'Boyle et al., 2013] to determine if the workload has a strong CPU bias and use CPU-biased asymmetric profiling instead of the GPU-biased one.

3.5.2 Addressing load imbalance

In order to address bias due to the initial profiling, the profiling can be repeated until a certain termination condition is reached, after which the benefits of re-profiling diminish. This ensures that we profile more than once to understand the application-level imbalance better. We consider two termination conditions: *convergence* and *size*.

Convergence-based strategy repeats profiling until β_g converges. For example, we could re-profile until computed ratios differ by no more than 0.05. The assumption with convergence strategy is that once the distribution ratio stops changing, it is close to the ideal ratio α_g. Convergence strategy works well when distribution ratio indeed stabilizes after temporary converging. On the other hand, *size*-based strategy repeats the profiling until a certain portion of the work items has been completed. For example, we could re-profile till the number of remaining items is more than 50%. Size-based strategy works well when *i)* running CPU threads in profiling mode does not impose a lot of overhead, and *ii)* the irregularity of the workload gets amortized over fixed portion of the items.

When re-profiling, it is important to keep both CPU and GPU busy. As has been shown by Theorem 3.5.1, asymmetric profiling incurs no overhead only when there is sufficient amount of work to keep CPU busy while GPU performs its fixed portion of work. The profiling strategy, thus, should keep the total fraction of items offloaded to GPU below the ideal distribution ratio α_g, or, more practically, below its approximation, β_g, from a prior profiling run.

It is important to note that there can be scenarios where neither size-based or convergence-based strategies can determine the optimal partitioning – for example, triangular loop inside a *parallel_for* loop. We believe such scenarios are rare in practice – none of the sixteen benchmarks we studied exhibit this kind of behavior.

3.5.3 Multiple invocations per kernel

In programs where the kernel is executed multiple times, the programmer can assume the first invocation as a profile run to obtain the rate of execution on both devices. After the first run, whenever we execute some items on one or both devices, we observe the execution rates and update our offload ratio β_g according to the selected update strategies described below.

3.5.3.1 Update functions:

Given an execution of some items on the devices, we use (C_r, G_r, C_n, G_n), where C_r and G_r are rates and C_n, G_n are number of items processed by the CPU and GPU respectively. Two variants we propose are :

1. **Greedy**: Simply use the G_r, C_r to compute β_g. This is always used the first time a rate is computed since we assume no initial distribution of work-items.
2. **Sample weighted**: Compute $w = \frac{C_n+G_n}{C_n+G_n+T_n}$, where T_n is the number of items used to compute the rate so far, and $\beta_g^{new} = \frac{G_r}{G_r+C_r}$. And set the new ratio to be $\beta_g = w\beta_g^{new} + (1-w)\beta_g$. Update $T_n += C_n + G_n$.

We also evaluate a device-weighted variant, which weights the rate computed by

each device by the number of items processed by the device for that profiling run. We do not observe a significant improvement in the performance of device-weighted updates. We use the *sample-weighted* updates for our experimental results.

3.6 Evaluation

We now present an evaluation of our techniques. We begin with an overview of the hardware and software environment. We next describe the benchmarks used in the evaluation and their static and runtime characteristics. Finally, we present performance results and follow it up with a summary.

Runtime: During program execution, the runtime loads the embedded OpenCL code and just-in-time compiles it to GPU *ISA* using the vendor-specific OpenCL compiler. It also decides how to distribute the work between the CPU and GPU. It implements work-stealing on the CPU, with one of the CPU worker threads (the GPU proxy thread) offloading to the GPU as guided by online profiling. The observed distribution ratio, β_g, for our scheduling algorithms (described in Sec. 3.4 and Sec. 3.5) is computed in the GPU proxy thread which then redistributes the parallel iterations among the CPU and GPU cores.

3.6.1 Environment

We evaluated our scheduling techniques on a desktop computer with a 3.4GHz Intel 4th Generation Core i7-4770 Processor with four CPU cores and with hyper-threading enabled. The integrated GPU is an Intel HD Graphics 4600 with 20 ex-

Name	Input	# kernels	kernel-name	invocations	Oracle-time(s)	num iter per kernel	Dynamic. Instr.
BarnesHut (BH) [Barnes and Hut, 1986]	1M bodies, 1 step	1	Barneshut	1	6.852	1000000	1.96×10^{11}
BarnesHut(unoptimized) (BH-U) [Barnes and Hut, 1986]	1M bodies, 1 step	1	Barneshut	1	9.194	1000000	1.98×10^{11}
LavaMD (LMD) [Che et al., 2009]	-boxed1d 10	1	kernel	1	0.582	1000	2.02×10^{10}
Matrix Multiply (MM)	2048 by 2048 float matrix	1	testc	1	1.506	4194304	7.08×10^{10}
Ray Tracer (RT)	sphere=256,material=3,light=5	1	kernel	1	2.426	10240000	3.25×10^{11}
Breadth first search (BFS)	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	1	relax	1748	14.05	6262104	5.68×10^{11}
Black Scholes (BS) [Bienia et al., 2008]	64K	1	mainwork	2000	0.32	65536	6.17×10^{10}
Connected Component (CC)	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	1	merge	2147	22.37	6262104	1.54×10^{12}
Face Detect (FD) [OpenCV, 2006]	3000 by2171 Solvay-1927	1	HaarDetect	132	1.884	22-1370340	6.09×10^{10}
Heartwall (HW) [Che et al., 2009]	test.avi	1	kernel	5	0.924	51	4.62×10^{10}
N-Body (NB)	4096 bodies	1	Slowpar	101	4.506	20475	3.91×10^{11}
Seismic (SM) [TBB (Intel Threading Building Blocks), 2011]	1950 by 1326, 100 frames	1	UpdateStress	100	2.048	25791520	1.43×10^{11}
Shortest Path (SP)	W-USA ($\|V\|$=6.2M, $\|E\|$=1.5M)	1	relax	2577	30.66	6262104	9.17×10^{11}
BTree (BT) [Che et al., 2009]	big-command	2	kernel_cpu	1	0.586	1000000	1.66×10^{11}
			kernel_cpu_2	1		1000000	
Computational fluid dynamics (CFD) [Che et al., 2009]	missile - 0.3M	4	compute-time	6000	24.072	232704	1.49×10^{12}
			compute-flux	6000		232704	
			compute-step	2000		232704	
			init	1		232704	
Particle Filter (PF) [Che et al., 2009]	$(128, 128, 10)$, 80K particles	7	find-index	9	4.646	8000	3.50×10^{11}
			calculate-u	9		8000	
			divide-weights	9		8000	
			update-weights	9		8000	
			particle-filter-like	9		8000	
			initial-arrays	1		8000	
			initial-weights	1		8000	

Table 3.1: Key statistics for the kernels of benchmarks used in the evaluation. *Oracle-time* is the time taken by the best offline distribution, and is also used as the normalizing factor in Fig. 3.14 and Fig. 3.15.

ecution units (EUs), each with 7 hardware threads where each thread is 16-wide SIMD, and running at a turbo-mode clock speed from $350MHz$ to $1.2GHz$. The system has 8GB system memory and is running 64-bit Windows 7.

3.6.2 Benchmarks

We consider a diverse set of applications (sixteen in total) to evaluate our proposed runtime system. These applications span a spectrum of application domains and exhibit different behaviors: single kernel vs. multiple kernels, single invocation vs. multiple invocations, and regular vs. irregular. Most of these were ported from existing sources: TBB [TBB (Intel Threading Building Blocks), 2011],

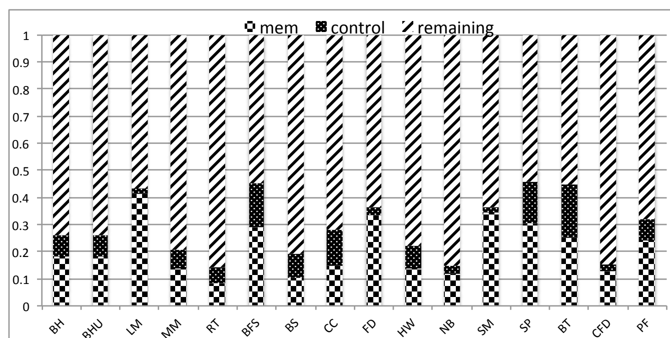


Figure 3.13: Total number of dynamic instructions of benchmarks divided into three categories: *memory*, *control*, and *remaining*, obtained via a serial CPU execution. The number of dynamic instructions is given in Table 3.1(column 9).

Rodinia [Che et al., 2009], and Parsec [Bienia et al., 2008]. We also developed some applications from scratch. The details of our benchmarks are shown in Table 3.1.

The compile-time and run-time characteristics of our benchmarks are tabulated in Table 3.1. The order of the benchmarks we use are as follows: (1) single kernel and single invocation (**BH**, **BH-U**, **LMD**, **MM**, **RT**); (2) single kernel and multiple invocations (**BFS**, **BS**, **CC**, **FD**, **HW**, **NB**, **SM**, **SP**); (3) multiple kernels and multiple invocations (**BT**, **CFD**, **PF**). The table shows the number of times a kernel is invoked in column 6. The *Oracle-time* in column 7 reports the absolute execution time for an *Oracle* approach described in Sec. 3.6.3 – this serves as the baseline for the runtime evaluation in Fig. 3.14 and Fig. 3.15, which are normalized to this time. Column 8 reports the number of parallel iterations per kernel invocation. Column 9 reports the number of instructions executed at runtime per benchmark.

To better understand the irregularities in our benchmarks, we also analyze

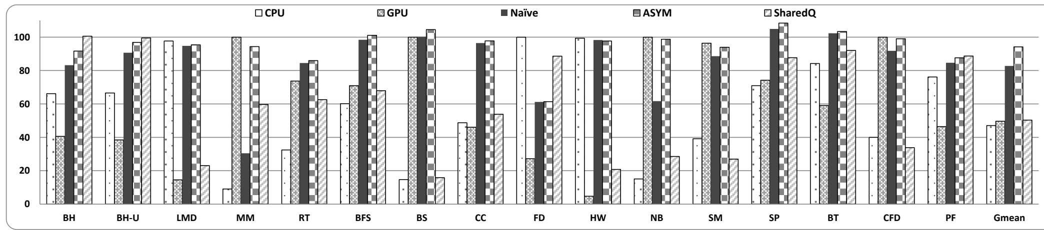


Figure 3.14: Relative speedup for all benchmarks compared to Oracle. Oracle is at 100% (higher is better).

the dynamic instruction traces for a single-threaded CPU execution via *Intel VTune Amplifier* to determine the run-time behavior of our applications[5]. Fig. 3.13 shows the division of the instructions into three categories: *memory* operations (load and store instructions), *control* instructions, and *remaining*, which can be a crude estimate for the *compute* instructions. The absolute number of dynamic instructions is given in column 9 of Table 3.1. Applications such as **BH**, **BH-U**, **BFS**, **CC**, **SP**, **BT**, and **PF** show considerable amount of control flow irregularities where as **FD**, and **SM** show significant amount of memory related operations. Since these applications are not hand-tuned for GPUs, the memory related operations may not necessarily be coalesced and give an indication of non-coalesced memory accesses on the GPU. Also note that benchmarks with large amount of control flow irregularities may not perform well on the GPU.

[5]Note that this measurement provides an upper-bound of the irregularities present in a benchmark.

3.6.3 Comparison schemes

We compare the following scheduling strategies to distribute the parallel iterations between the CPU and GPU:

1. **CPU**: Multi-core CPU execution based on TBB. The *parallel_for* invocation is simply forwarded to the TBB runtime to complete the execution.

2. **GPU**: GPU-only execution, where all items are offloaded to the GPU via OpenCL.

3. **Oracle**: The best performance obtained by exhaustive search of different amount of parallel_for iterations assigned to the CPU and GPU. The GPU works on some percentage of the parallel iterations while the CPU works on the remaining ones. The percentage is varied from 0% to 100% on the GPU in increments of 10% and the best performance obtained is selected. The best percentage for single kernel benchmarks is given in Table 3.2(row 1) and the absolute runtime for those percentages is given in Table 3.1(column 7). We select the runtime obtained by such a technique as the baseline for all comparisons.

4. **Naïve**: The naïve profiling scheme described in Sec. 3.4 where we use the same profiling size for both the CPU and the GPU.

5. **ASYM**: The asymmetric profiling scheme described in Sec. 3.5. We use the *sample weighted* variant for multiple invocation kernels.

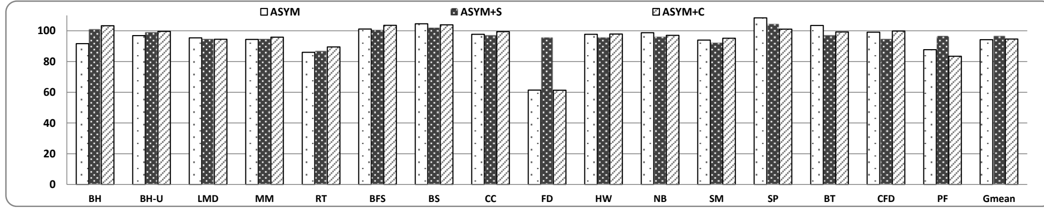


Figure 3.15: Comparison of different adaptive schemes. Vertical axis shows relative speedup vs. Oracle (higher is better).

6. **SharedQ**: We also compare against a shared queue implementation where a GPU proxy-thread atomically grabs a fixed chunk of work (profile size) and offloads it to the GPU while CPU threads also atomically obtain work-items from the same global queue. Since there will be contention on the shared queue, this approach may suffer as the number of workers increase[6].

We also evaluate two variants of our asymmetric profiling algorithm to address load imbalance, as described in Sec. 3.5.2:

- **ASYM+CONV**: A convergence-based update strategy, where half of remaining items are used for reprofiling until convergence is achieved. Convergence is reached when the β_g computed on two successive profiling steps do not differ by more than 0.05.

- **ASYM+SIZE**: A size-based update strategy, where we repeatedly profile with f_p items on the GPU until at least half of the total items are left.

[6]Note that our SharedQ implementation does not use the optimization strategies described in Sec. 3.5.3.

3.6.4 Results

Heterogeneous execution: First we highlight the significance of heterogeneous execution compared to single device execution. We compare the single-device execution (either on the CPU or the GPU) against the offline optimal model (Oracle). Fig. 3.14 shows that there is an obvious advantage to heterogeneous execution: (1) using only the multi-core CPU, we can achieve 47.07% of Oracle on average; (2) using only the GPU, we can achieve 49.67% of Oracle on average. Clearly, heterogeneous execution results in $2\times$ improvement in runtime compared to the best of executing on the CPU-alone or the GPU-alone. These results emphasize the improvement of heterogeneous execution over single device execution.

Naïve profiling: Naïve profiling as described in Sec. 3.4 performs online profiling on a fraction of the parallel iterations to determine the offload ratio, β_g. Although the profiling information obtained is able to outperform single-device execution as illustrated in Fig. 3.14, it only performs at 82.7% of the Oracle on average. There are two sources of inefficiency in naïve profiling: (1) The *overhead of profiling* can dominate the execution time for workloads that are highly biased towards a particular device. For instance **NB** and **MM** are both highly biased towards the GPU. With a fixed profiling size on both devices, the scheduler has to wait at the barrier for both devices to report the rates to determine the distribution. If the GPU profiling finishes early, the scheduler waits for the CPU to complete the profiling step resulting in a large overhead. (2) The *inaccuracy of profiling information*, although not evident, is another source of inefficiency. For instance, **FD** only obtains 61.2% of the Oracle, because the profiling information obtained is not

	BH	BH-U	LMD	MM	RT	BFS	BS	CC	FD	HW	NB	SM	SP
Oracle	40	40	0	100	70	70	100	70	0	0	100	70	50
Naïve	52.4	44.3	0	89.6	84.4	72.8	94.9	73.9	31.1	0	77.3	77	71
ASYM	47.9	41.7	0	92.5	84.3	72.8	96.2	73.8	32.9	0	89.6	92.1	70.6
ASYM+SIZE	39.6	40.2	0	92.2	69.7	61.1	96.0	68.8	18.4	0	88.3	92.2	59.2
ASYM+CONV	41.2	40.6	0	92.4	70.6	73.1	96	73.8	33.2	0	89	92.7	70.7

Table 3.2: GPU work percentages computed by different schemes for single-kernel applications.

representative of the entire iteration space. The inaccuracy can also be observed in optimal distribution percentages (as shown in Table 3.2) for benchmarks with single *parallel_for* kernels.

Asymmetric profiling: As described in Sec. 3.5, asymmetric profiling does not suffer from the overhead of waiting on a barrier to compare the rates for both devices. Most applications benefit from using asymmetric profiling compared to naïve profiling. That is, it performs on average at 94.2% efficiency compared to the Oracle. The biggest improvement is observed in applications that show a high GPU bias such as **NB** and **MM**. This is because the distribution ratio β_g can be decided quickly, and hence the overhead of profiling is reduced significantly (close to zero[7]). We note, however, that **BH** and **FD** do not benefit from the vanilla version of asymmetric profiling. This is, as explained before, due to the inaccuracy of the profiling information as evident from the distribution computed by asymmetric profiling differing from the optimal distribution obtained by the Oracle (as shown in Table 3.2). Note that benchmarks such as **BS**, **BT**, **BFS**, and **SP** improve perfor-

[7]This overhead in our implementation consists of : for N workers, we read a local counter from each worker and perform N additions and 3 divisions, which is negligible compared to the total execution time of an application.

mance better than Oracle using asymmetric profiling – this is because the Oracle is obtained in increments of 10%.

ASYM+CONV: In order to address the load imbalance and the inaccuracy of profiling information, we rely on repeated profiling to achieve more accurate information. The convergence based approach repeatedly profiles until the offload ratio, β_g, determined by consecutive profiling phases differs by less than 0.05. This does introduce an overhead of repeated profiling, but the benefit of more accurate profiling information should offset this overhead. **ASYM+CONV** can achieve on average 94.6% of Oracle as shown in Fig. 3.15. The biggest improvement is observed in the applications that have some irregularity, for instance **BH**. The convergence based approach determines a more accurate distribution ratio, as evident from the percentages reported in Table 3.2 (47% for **ASYM** versus 41% for **ASYM+CONV**) is more closer to the optimal value 40%. Similarly, for **RT**, the ratio reported becomes 69% versus the optimal 70%.

However, we observe that this approach still does not improve **FD**, and furthermore the efficiency of **PF** goes down. Both of them are false positives as they converge fairly quickly in **ASYM+CONV** approach. The source of inefficiency for **FD** stems from the algorithm: the cascade of classifiers is grouped to discard failing face images early. The profiling phase obtained from the initial rounds converges in two iterations on these cascade of classifiers, which is not representative of later stages that search further for faces in parts of the image not discarded by earlier stages. For **PF**, an inaccurate rate due to convergence makes the performance drop from 87% for **ASYM** to 83% for **ASYM+CONV**. Clearly, in order to improve the

efficiency of **FD** and **PF**, the profiling information has to be resilient to local optima. Also note that, **FD** shows a large amount of memory related irregularities and **PF** shows a large amount of control flow and memory irregularities (as shown in Fig. 3.13).

ASYM+SIZE: Instead of relying on convergence, we use half of the work-items to determine the profiling ratio and use this ratio to distribute the remaining items. This approach proves the best performing overall since it does not rely on an initial portion of the work-items to determine the rate and does not converge on a local maxima. There is an overhead to repeated profiling which reduces the efficiency for some applications, but not significantly. However, the biggest winner is **PF**, which performs at 96% of Oracle, followed by **FD**, which performs at 95% of Oracle. Overall, the size-based approach performs at 96.8% (geo-mean) efficiency of the Oracle, thereby clearly demonstrating that it is the best strategy across all benchmarks.

3.7 Summary

This chapter shows how to transform C++ code to execute efficiently on integrated GPUs. To facilitate a large set of applications written in a language such as C++, key features such as classes, virtual functions, and pointer-based codes have to be supported. Besides supporting these features, the translation has to ensure that the generated code does not incur large overheads through several optimizations. Some of these optimizations are extensively studies in literature, while some, as described in Sec. 3.2.5 are specific to the target architecture. Although a compiler

enables application kernels to be executed efficiently on an integrated GPU, runtime information is required to optimally balance workload between the multi-core CPU and integrated GPU in a heterogeneous execution. Specifically, irregular applications exhibit large variance in performance behavior which cannot be modeled at compilation time. These applications require runtime strategies to dynamically profile and adjust the workload distribution between the multi-core CPU and integrated GPU. A naïve profiling strategy has the potential to incur large overheads, especially for applications that are balanced towards either the multi-core CPU or the integrated GPU. An alternate approach, as described in Sec. 3.5 avoids this problem by not stalling the multi-core CPU during the profiling phase. Various heuristics can be used to improve the balance even further at the potential expense of repeated profiling phases.

Chapter 4

Data-parallel execution with discrete GPUs[1]

The previous chapter discussed execution on a heterogeneous system consisting of a CPU and an integrated GPU. These devices are attractive as they provide an accelerator in the form of an integrated GPU at a low power budget along with coherent memory obviating the need for data movement. However, these accelerators are constrained in their computational capability. When more powerful accelerators are required, or when multiple accelerators are required, a discrete GPU is used. In this chapter, we shift our focus to discrete GPUs to accelerate graph applications.

4.1 Introduction

GPUs, primarily designed to accelerate graphics workloads, can execute regular kernels efficiently. Irregular kernels, such as those for graph applications, can be very challenging. There are two aspects of graph applications that are addressed in this chapter. First, we discuss the implementation of a topology-driven

[1]Portions from this chapter have been published in [Kaleem et al., 2015, Kaleem et al., 2016] where the synchronization strategies for Stochastic Gradient Descent were first presented. All key ideas in these publications were conceived by the first author, as well as all the OpenCL implementations.

algorithm (Sec. 2.3.2), focusing on the scheduling of activities on a single device and how to harness an efficient single device execution in a heterogeneous system. Next, in Sec. 4.7, we discuss data-driven algorithms Sec. 2.3.2 which require efficient communication mechanisms across different devices in a heterogeneous system.

4.2 Graph algorithms on discrete GPUs

We begin with topology-driven graph algorithms. When writing code for such algorithms, GPU programmers are faced with many implementation choices, but the performance implications of these choices are usually not obvious. Some implementation choices can be easily explored by changing compiler flags or modifying a few lines of code, but others lead to entirely different programs so exploring the space of possibilities may involve substantial programmer effort. The choice of synchronization strategy is an example. At a high level, programmers have a choice between coarse-grain, barrier-style synchronization or fine-grain synchronization using constructs like atomics or locks. To use barrier-style synchronization, the program must be executed in rounds and the tasks in each round must be independent; with fine-grain synchronization, there may not be a notion of rounds, and concurrently executing tasks may read and write the same locations provided these memory accesses are properly synchronized. Not only are the resulting programs very different but each synchronization style can itself be implemented in many ways, as we show in this chapter. Furthermore, the performance of irregular graph programs can be very dependent on the structure of the input graph: a program

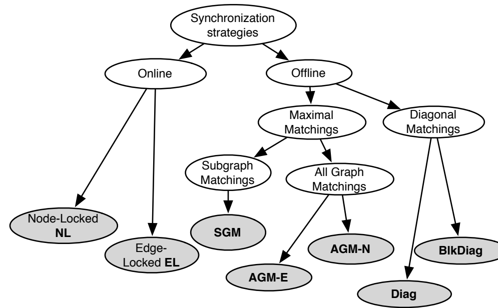


Figure 4.1: Taxonomy of scheduling strategies.

that performs well for power-law graphs may perform poorly for high-diameter graphs like road networks. Given all these complications, programmers would obviously benefit from guidelines that would help them make the right implementation choices.

As a step towards this goal, we consider the problem of implementing synchronization for graph algorithms, using non-negative matrix factorization (NMF) [Lee and Seung, 2001] as an exemplar. NMF is used to solve problems such as product recommendation and object recognition [Lee and Seung, 1999]. In Sec. 4.2.1, we describe a particular approach for solving NMF called *stochastic gradient descent* (SGD), which is an important general optimization method in machine learning. Fig. 4.1 shows a taxonomy of scheduling strategies for implementing SGD on GPUs.

In Sec. 4.3, we describe *offline* techniques, which pre-process the input to find independent tasks before executing the program, and generate code in which all synchronization is barrier synchronization. Some of these techniques compute *maximal matchings* in the graph to minimize the number of barrier synchroniza-

tions. We also explore a class of preprocessing techniques called *diagonal matchings*, which have lower preprocessing time but may require more barrier synchronization. In Sec. 4.4, we describe two *online* schedules that we call *Edge-locked* (EL) and *Node-locked* (NL) implementations, which use fine-grain synchronization to coordinate the parallel tasks. We discuss strategies to execute SGD in a heterogeneous section in Sec. 4.5. In Sec. 4.6, we evaluate the performance of these implementations of SGD on two platforms, an NVIDIA Tesla K40C and an AMD Hawaii Radeon R9-290X for both power-law graphs and road networks.

4.2.1 Stochastic Gradient Descent (SGD)

Recommendation systems [Adomavicius and Tuzhilin, 2005] solve problems like the Netflix challenge problem, which can be described abstractly as follows: given a set of users U, a set of movies M, and an incomplete database of movie ratings by users, predict how users will rate movies they have not yet rated.

One way to solve this problem is through non-negative matrix factorization, which is a kind of low-rank approximation. The database of ratings is represented as a sparse matrix R in which the rows represent users and the columns represent movies. Low-rank approximation finds two low-rank dense matrices W and H such that $R \approx W * H$ as shown in Fig. 4.2. That is, each non-zero entry in R must be roughly equal to the corresponding entry in $W * H$; the remaining entries in $W * H$ are the predictions for the missing ratings.

Low-rank approximation can be formulated as a graph problem. The database of ratings R is represented as a bipartite graph between users and items; if user u

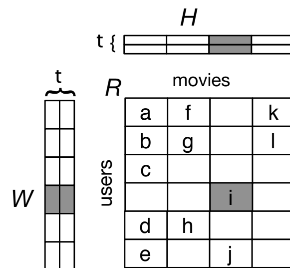


Figure 4.2: Low-rank approximation of a sparse matrix R by low rank matrices $W : |U| \times t$ and $H : t \times |M|$, usually $t \approx 16$.

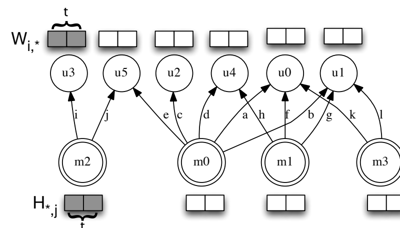


Figure 4.3: Sample bipartite graph between 6 users and 4 movies. Edge labels indicate ratings.

assigned a rating r to a movie m, there is an edge (u, m) in the graph with weight r. The matrices W and H are represented by unknown vectors of length t associated with the nodes representing users and movies respectively, as shown in Fig. 4.3 (these are known as *feature vectors*). The problem is to find values for these vectors such that for every edge (u, m) with weight r, the inner-product of the vectors on nodes u and m is roughly equal to r.

SGD is an iterative algorithm that computes feature vectors by making a number of sweeps over the bipartite graph. The vectors are initialized to some arbitrary values. In each sweep, all edges (u, m) are visited. If the inner-product of the vectors on nodes u and m is not equal to the weight on edge (u, m), the difference is used to update the two feature vectors. Sweeps are terminated when some heuristic measure of convergence is reached.

Parallelism can be exploited in each sweep by processing edges in parallel. Two edges can be processed in parallel provided they do not share a node; otherwise, they are said to *conflict* and must be processed serially. In our example, edges a and b conflict because they share the same movie m_0; similarly, edges a and f conflict because they share the same user u_0. Thus, the programmer needs to synchronize accesses to edges to avoid processing conflicting edges concurrently. The rest of this chapter explores the performance implications of different ways of implementing this synchronization.

4.3 Offline Schedules

In a given graph, a set of edges is said to constitute a *matching* if no two edges in that set have a node in common [Garey and Johnson, 1990]. Matchings are useful for parallel SGD computation because the edges in a matching can be processed in parallel without the need for synchronization. A *maximal matching* is a matching m such that every edge not in m conflicts with some edge in m.

Offline schedules pre-process the graph by partitioning its edges into a set of matchings. The SGD computation is then implemented as a series of super-steps separated by barriers; in each super-step, the edges in one matching are processed in parallel without synchronization. In Sec. 4.3.1, we describe *maximal-matching schedules* which partition the edges of the graph into a sequence of maximal matchings.

The second approach relies on the structural properties of the bipartite graph. If the graph is viewed as an adjacency matrix, entries along the diagonals of the matrix can be processed concurrently as they do not share any end-points. This observation allows us to utilize sparse linear algebra frameworks such as CUDA-CHiLL [Rudy et al., 2011] to synthesize scheduling routines for graph applications such as SGD. These *diagonal-matching schedules* are described in Sec. 4.3.2.

To illustrate the schedules, we use the graph of Fig. 4.3 and a hypothetical GPU with two threads. Our actual implementations run on an NVIDIA Tesla K40 and AMD R9-290X, as explained in Sec. 4.6, so the two-thread hypothetical GPU is used only for illustration.

4.3.1 Maximal matchings schedules

The first category of schedules rely on maximal matchings. Algorithm 1 shows the algorithm to construct a maximal-matching schedule. To build a conflict free schedule, *i)* a maximal matching m is constructed from the graph; *ii)* the edges belonging to the maximal matching are removed from the graph; and, *iii)* the process repeated until there are no edges left in the graph. We refer to this set of maximal matching as a *matchings-set* M. The number of matchings in M is greater than or equal to the max-degree of the graph D_{max} since all edges of that node must be processed in separate matchings. Fig. 4.4(a) shows the matchings-set M for the sample graph.

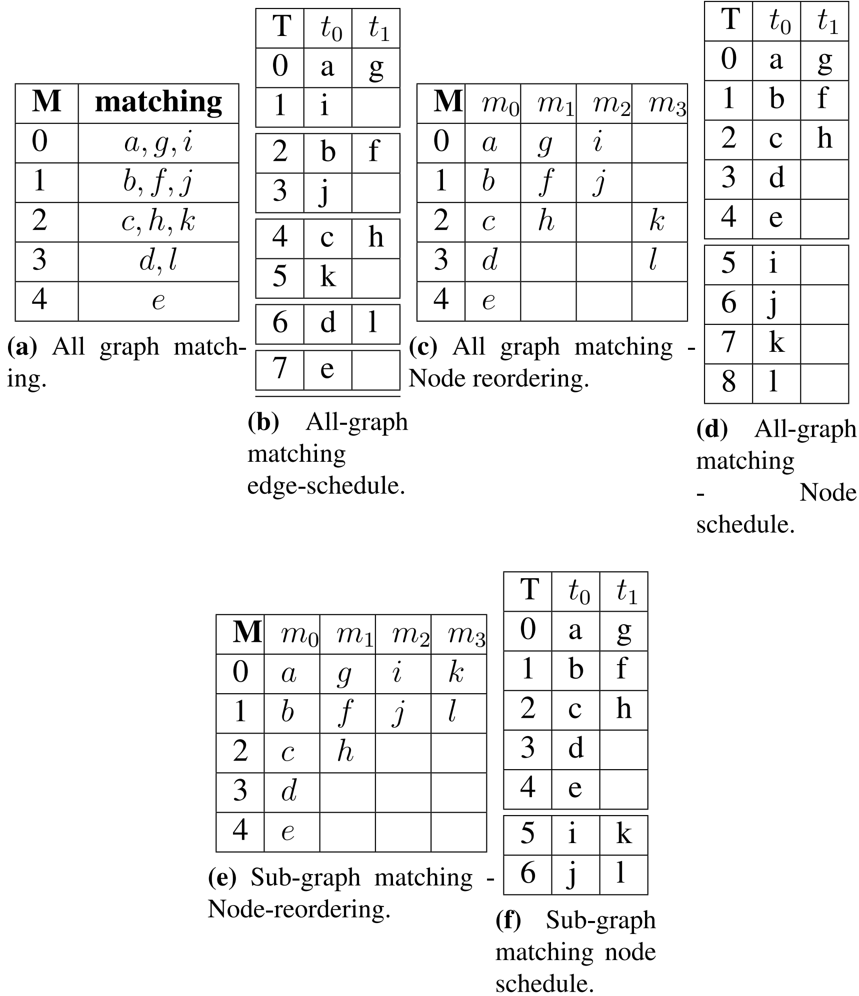
```
1   while{edges(g) > 0}{
2     m = maximal_matching(g)
3     g = g ∖ m
4     M = M ∪ {m}
5   }
6   return M
```

Algorithm 1: Algorithm for constructing a maximal-matching schedule. Given a graph g, returns the set of matchings M.

Given a matchings-set M, we describe three different strategies for scheduling edges within a set $m \in M$.

4.3.1.1 All-Graph Matching-Edge schedule (AGM-E)

In an AGM-E schedule, matchings are processed one at a time. Each thread grabs an edge, load the labels at the end-points of that edge, performs the SGD computations, and updates those labels. This process is repeated until there are



M	**matching**
0	a, g, i
1	b, f, j
2	c, h, k
3	d, l
4	e

(a) All graph matching.

T	t_0	t_1
0	a	g
1	i	
2	b	f
3	j	
4	c	h
5	k	
6	d	l
7	e	

(b) All-graph matching edge-schedule.

M	m_0	m_1	m_2	m_3
0	a	g	i	
1	b	f	j	
2	c	h		k
3	d			l
4	e			

(c) All graph matching - Node reordering.

T	t_0	t_1
0	a	g
1	b	f
2	c	h
3	d	
4	e	
5	i	
6	j	
7	k	
8	l	

(d) All-graph matching - Node schedule.

M	m_0	m_1	m_2	m_3
0	a	g	i	k
1	b	f	j	l
2	c	h		
3	d			
4	e			

(e) Sub-graph matching - Node-reordering.

T	t_0	t_1
0	a	g
1	b	f
2	c	h
3	d	
4	e	
5	i	k
6	j	l

(f) Sub-graph matching node schedule.

Figure 4.4: Maximal matchings schedules executed by different strategies for sample input. Tables with **M** in top-left cell indicate matching sets, whereas tables with **T** in top-left cell indicate schedules where each row indicates a time-step and each column lists the edges processed by a thread.

no more edges left to be processed in that matching. Note that AGM-E *makes no attempt to schedule edges connected to the same node on the same thread.*

For our sample graph, an edge schedule of this sort is shown in Fig. 4.4(b). In our model GPU, we can execute only two edges per step so the processing of the first matching takes two steps, and a sweep over all edges takes eight steps.

4.3.1.2 All-Graph Matching-Node schedule (AGM-N)

Unlike the AGM-E schedules, these schedules attempt to exploit locality in processing edges and utilize the local shared memory of the GPU to store the data associated with the nodes.

In our implementation, edges connected to a given movie node are all processed by the same thread. This is accomplished by processing movie nodes in blocks of T nodes, where T is the number of threads (the last block may have fewer nodes). Consider Fig. 4.4(c), which shows a matrix in which the rows are the matchings and the columns are the movie nodes. Conceptually, we divide the columns of this matrix into blocks of T nodes, and process these blocks sequentially. Since we have two threads in our example, m_0 and m_1 are in the first block, and m_2 and m_3 are in the second block. When processing a given block of nodes, we iterate over all matchings in sequence, processing the appropriate edges as shown in Fig. 4.4(d).

Each block column is processed by making a kernel call. Before a block column of movie nodes is processed, the associated movie node data is read into shared-memory. Global inter-thread block synchronization is used to separate the processing of edges from different matchings. After the processing is complete, the

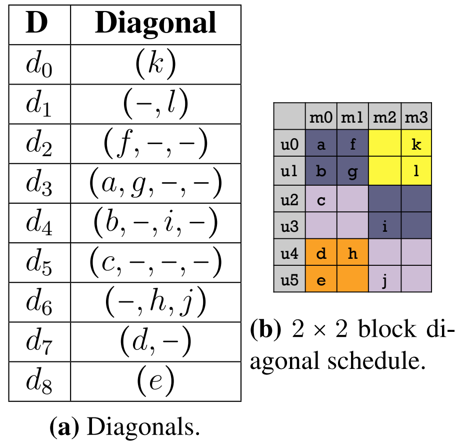
movie node data is written back into memory.

4.3.1.3 Sub-Graph Matching (SGM)

This strategy can be viewed as a refinement of AGM-N. For large graphs, the number of nodes will be more than the number of threads T. In that case, computing a matchings-set for the entire graph and then repackaging it for the AGM-N schedule can be inefficient. In Fig. 4.4(d), it takes four steps to execute the second block of nodes consisting of $\{m_2, m_3\}$ even though edges i and j can be processed in parallel with edges k and l respectively. Intuitively, if the number of threads is smaller than the number of nodes, the nodes will be processed in blocks, so matchings should be computed only for nodes in the same block.

This is accomplished by the SGM scheduling strategy. SGM first sorts the nodes in decreasing order of node degree. Then it partitions the nodes into blocks of size T. For each block, the matchings-set is computed, and edges are scheduled for that matchings-set as in AGM-N. The sub-graph matchings for the sample graph is shown in Fig. 4.4(e), and the SGM schedule is shown in Fig. 4.4(f).

The preprocessing time for SGM is different from the preprocessing time for the all-graph matching variants. When building all-graph matching, a single matchings-set is built for the entire graph. However, for SGM, we build the matchings-set for each block of nodes separately.



D	Diagonal
d_0	(k)
d_1	$(-, l)$
d_2	$(f, -, -)$
d_3	$(a, g, -, -)$
d_4	$(b, -, i, -)$
d_5	$(c, -, -, -)$
d_6	$(-, h, j)$
d_7	$(d, -)$
d_8	(e)

(a) Diagonals.

(b) 2×2 block diagonal schedule.

Figure 4.5: Diagonal matchings schedules for the sample input.

4.3.2 Diagonal matchings schedules

The schedules discussed in Sec. 4.3.1 are based on maximal matchings. To reduce the preprocessing overhead, schedules can be constructed using matchings that are not necessarily maximal.

One way to construct matchings cheaply is to exploit the matrix representation of the graph [Venkat et al., 2015]. In the matrix representation, edges along a diagonal do not share any nodes and can be processed concurrently. Different diagonals must be serialized, however.

Diagonal matchings schedules can be advantageous as they facilitate temporal reuse of the nodes, but the benefits must outweigh the overhead of the barrier synchronization between diagonals. We increase the granularity of work within a diagonal, and therefore reduce the frequency of barrier synchronization, using two diagonal variants: (1) Diag (Sec. 4.3.2.1) and, (2) BlkDiag (Sec. 4.3.2.2).

4.3.2.1 Diagonal (Diag) schedule

Diag also exploits the parallelism within a single edge by processing the update to the feature vector in parallel. This ordering also achieves global memory coalescing for accesses to the feature vector across threads. We launch a 2-D grid of threads of dimension F by E, where F is the size of feature vector (e.g., 16 floats), and E is the number of edges to be processed in a kernel call. A thread (i, j) processes the ith component of the feature vectors of the end points for edge j.

The maximum number of diagonals is $|M| + |U| - 1$ and the maximum width of a diagonal will be the number of columns. In our example, 4 movies (columns) and 6 users (rows) result in $4 + 6 - 1 = 9$ diagonals with the longest diagonal containing 4 entries. The complete list of diagonals, starting from the top is given in Fig. 4.5(a).

4.3.2.2 Block-Diagonal (BlkDiag) schedule

The BlkDiag schedule reduces the size of the matrix by blocking along both dimensions. This reduced matrix has a reduced number of diagonals – if the movies are blocked by a factor R, and the users by a factor C, then the total number of diagonals in the BlkDiag schedule is $|M|/R + |U|/C - 1$.

A diagonal schedule obtained after 2×2 blocking is shown in Fig. 4.5(b). There are now only 4 diagonals with each block consisting of at most 4 edges. Our implementation assigns each block to a thread. Within a block, the same set of movies and users are used repeatedly and the feature vectors corresponding to those

rows and columns are cached in registers or GPU shared memory.

Comparison to Matchings The diagonal matchings schedules are relatively easy to compute; the diagonal is determined by the difference in the row and column indices for an entry. It is, however, conservative because it will schedule entries concurrently *only* if they are along the same diagonal. In contrast, the maximal matchings schedules are more liberal but also costly to compute. Since the maximal matching does not constrain itself to any diagonal, it can discover more entries to schedule concurrently. For instance, in our example d_3 contains (a, g). We can also schedule either of i or j with these entries since they do not have any end point in common. The diagonal matchings schedule will not schedule i or j with d_3 since neither is along the diagonal d_3.

4.4 Online Schedules

Online schedules assign edges to threads without attempting to avoid conflicts. Therefore, synchronization primitives such as atomics must be used to ensure mutual exclusion.

We describe two strategies. The *Edge-locked* strategy EL, described in Sec. 4.4.1, assigns edges to threads. The *Node-locked* strategy NL, described in Sec. 4.4.2, assigns nodes to threads.

T	t_0	t_1
0	a	
1	c	
2	e	f
3	g	
4	i	
5	k	
6		b
7		d
8		h
9		j
10		l

(a) Edge-lock schedule without shuffle.

T	t_0	t_1
0	k	e
1	b	
2	j	c
3	h	
4	f	
5	i	l
6		d
7		g
8		a

(b) With shuffle.

T	t_0	t_1
0	a	
1	b	
2	c	h
3	d	
4	e	
5		f
6		g
7	i	k
8	j	l

(c) Node schedule.

Figure 4.6: Schedules observed for sample input under EL(without and with shuffle) and NL on the hypothetical GPU. Each row indicates edges scheduled at that time slot, and each column indicates the item processed, if any, by each thread.

4.4.1 Edge-locked (EL)

In each SGD sweep, threads make a number of passes over the set of edges until all edges have been processed. To process an edge, the thread attempts to acquire locks on its two nodes, and updates node labels if lock acquisition succeeds. Otherwise, the edge is deferred and retried in the next pass.

One possible schedule for the edges of Fig. 4.3 is shown in Fig. 4.6(a). We assume that the edges in the graph are stored in alphabetical order, which is similar to a CSR representation of Fig. 4.3. Since our hypothetical GPU can execute two tasks at once, it will pick chunks of two edges from the work-list and try to process them. The first two edges are $\{a, b\}$. Since they share the same source m_0, only

one thread succeeds in acquiring the lock, and edge b is delayed to the next pass. The next chunk to be executed is $\{c, d\}$, and only one edge gets processed while the other is moved to the next pass, and so on. The second pass, which starts at step 6, processes edges $\{b, d, h, j, l\}$ which could not be processed in the first pass.

The main problem with this strategy is that if edges connected to the same movie are tried concurrently, only one of the threads will make progress. The original ordering of the edges was derived from the CSR layout of the graph, which stores the edges of a given node in adjacent memory locations. This introduces a large number of conflicts, particularly for high-degree nodes.

To ameliorate this problem, we can shuffle edges randomly[2] before assigning them to threads. This lowers the likelihood that edges sharing the same movie are scheduled concurrently. For our sample graph, we shuffle the edges (for instance to $\{k, e, b, d, j, c, h, g, f, a, i, l\}$) and obtain a schedule as shown in Fig. 4.6(b). By mixing the edges of m_0 with edges from other nodes, we reduce the likelihood of conflicts. Experiments on actual input graphs confirm that shuffling can improve performance significantly.

For EL, preprocessing time involves the shuffling of edges to reduce the conflicts as described above. The execution time includes the time to perform kernel calls and the determination of whether all edges have been processed.

[2]Our implementation uses the *std::random_shuffle* call to shuffle the edges.

4.4.2 Node-locked (NL)

The Node-locked (NL) scheduling strategy assigns movie *nodes* to threads. This has two benefits. First, there is no need to acquire locks on the source node (i.e. movie) since a node is assigned to a single thread. Locks will still need to be acquired on the destination nodes (i.e. user). Second, unlike the EL schedule whose access patterns make it hard to exploit locality, the NL schedule can exploit reuse of the source node data.

Like the EL schedule, the NL schedule uses multiple passes to process all the edges of a graph.

Fig. 4.6(c) presents a possible NL-schedule. We first schedule nodes m_0 and m_1, and their edges are processed in order. In the first step, all the edges for m_0 are processed and marked while f and g are deferred to the next pass. In the next pass, f and g which were unmarked in the previous pass, are processed and marked. Next, we schedule the remaining nodes m_2 and m_3, which concludes without any conflict.

NL behaves similar to EL for the initial few passes as it can find work easily. But after the initial few passes, there is a large overhead of finding new work as each thread has to scan a node's entire edge-list. This can be prohibitive for high-degree nodes as the repeated scans become expensive.

The use of shared memory for storing the movie node's latent vector reduces the residency of the kernel, which means the number of edges that can be concurrently processed on the GPU is reduced. Further, since only one thread processes

all edges of a node, nodes with high degrees can lead to serialization and load imbalance. The use of marks implies that all edges must be scanned in every pass to determine if they must be processed. As we shall see in the evaluation, these factors play a major role in the performance of NL.

There is no preprocessing required for NL since the graph representation allows threads to traverse neighbors of each movie/user directly. The execution time includes the time to invoke the kernels as well as polling the number of edges processed to check for completion.

4.5 Heterogeneous schedules

As described above, matching based schemes decompose the graph into a set of edge-matchings. Each matching represents a conflict-free edge set which can be executed concurrently on the GPU. We now describe why each of the different matching schedules, AGM-E, AGM-N, and SGM, are unsuitable for heterogeneous execution.

1. AGM-E – since each matching represents an independent set of edges (a conflict-free set), we can partition the conflict-free set across multiple devices and execute them concurrently. This should be more efficient than processing the conflict-free set on a single device. However, the conflict free sets for most inputs is rarely large enough to maximally utilize a single device. A more critical issue has to do with managing a consistent global state. When a device needs to process an edge (i, j), it needs to load the node-data asso-

ciated with the end-points of the edge, perform the update and write back the results. Writing back is necessary since the next conflict free setting might allocate either end-point i or j to a different device. Since the node-data for all the edges in a conflict free set is not necessarily contiguous, the write-back stage is faced with a complex merging operation of updates from multiple devices.

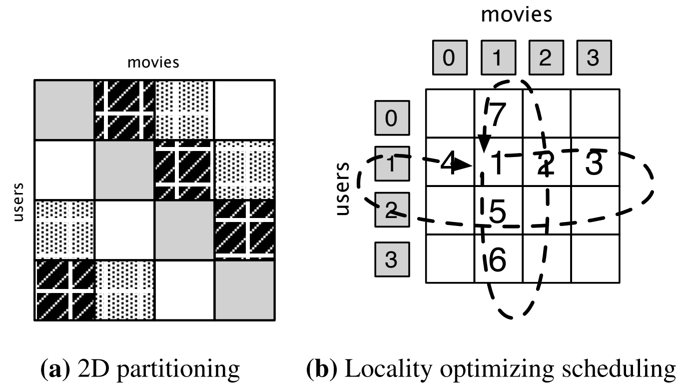
2. AGM-N – we can now partition the nodes along one dimension (movies or users) across all the devices. This reduces the number of nodes we have to merge at each step to half since one dimension remains fixed to each device. However, note that we cannot utilize the local memory on the device as we have to synchronize across device at each step. This is necessary to ensure we do not process edges across conflict-free sets.

3. SGM We make the same arguments against SGM as we did for AGM-E.

As described above, the matching schedules are too rigid for heterogeneous execution. Instead we resort to EL, the most efficient single device schedule. If we can partition the graph into a set of disjoint edges, we can schedule each device via EL to process all the edges in a partition. By adjusting the size of the partitions, we can balance the workload across different devices.

4.5.1 Graph partitioning

We build on the observation from diagonal schedules that the edges along a diagonal do not share any end-point and can be processed concurrently. For het-



(a) 2D partitioning **(b)** Locality optimizing scheduling

Figure 4.7: Partitioning the graph along movies and users for multi-device execution. Arrows indicate the search pattern for more work for a device which has just completed work on partition $(1,1)$. First it searches horizontally to maximize user-locality, and if it fails, it retries vertically. The numbers indicate the search order.

erogeneous execution, we would like the diagonals to contain sufficient edges to amortize the data movement with computation. To achieve this, we perform a 2D partitioning of the graph as shown in Fig. 4.7. Here, we partition the adjacency matrix along both dimensions, as well as partitioning the user features and movie features. Now the devices can process blocks along a diagonal in a bulk-synchronous manner. Each device picks a block along the diagonal, load the associate user features and movie features as well as the edges in the block. Once the device has finished processing the edges, it will write back the user and movie features to be read by other devices when the next diagonal is processed.

This approach suffers from under-utilizing the devices if one of the devices takes a long time to process its block. Since the diagonals are processed in a bulk-synchronous manner, all the devices have to wait at the barrier. Even if the barrier

were removed, since the number of diagonals equals the number of devices, at any point during the execution of a diagonal, all the user and movie features are in use by one of the devices. If a device finished execution earlier than other devices, it still has to wait for other devices to write the result of their block before it can load them and process a block in another diagonal.

We address the device underutilization by over-decomposing the graph. We create Z diagonals along each dimension where Z is the over-decomposition factor. Now, there are more diagonals than devices, so each device can now move on to a block in another diagonal once it has completed its current block. The device will simply scan the partitions for any unprocessed block along a diagonal and load its user and movie features. Assuming the size of each block is a, the amount of data transferred in processing the entire partitions is given by $2aZ^2$.

Instead of relying on a random search to look for more blocks to process, we can improve the search to optimize for locality. Once a block has been processed by a device, the device will first search for available blocks that share the same user features or movie features with the current block. This is illustrated by the Fig. 4.7(b). If block $(1, 1)$ has just been processed by a device, the device searches for new blocks along the arrows shown in the figure. If a block is found on this search path, only one of the user or movie features has to be copied into the device. This approach reduces the amount of data moved to $aZ(1 + Z)$.

Table 4.1: Specifications of the platforms used for evaluation.

	Host	Device
K40	Scientific Linux 6.6, Kernel 2.6.32 on Intel Xeon E5-2609 with 32G RAM	Tesla K40c with 12GB VRAM
R9-290X	Ubuntu 14.04, Kernel 3.16.0 on Intel i7-3770K with 8GB RAM	AMD R9-290X with 8GB VRAM

4.6 Evaluation

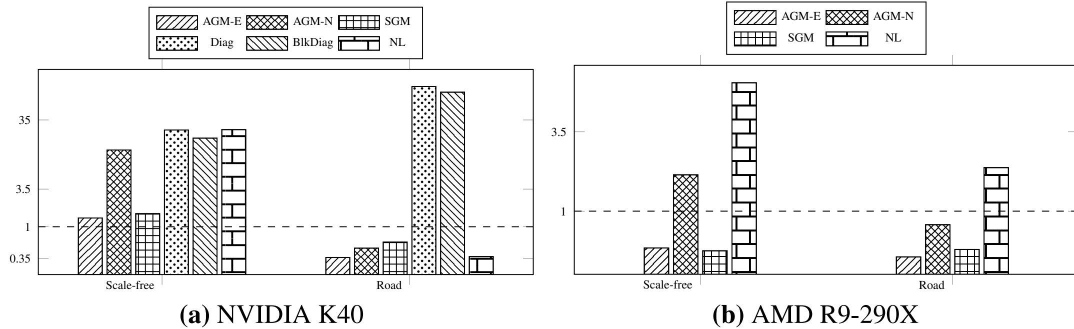
Our evaluation examines the performance of the different synchronization schemes on two hardware platforms described in Table 4.1. The online and maximal matching schedules are implemented[3] in OpenCL 1.2, the latest supported by NVIDIA The diagonal matchings schedules are generated via CUDA-CHiLL [Rudy et al., 2011].

We use twelve input graphs in our experiments (Table 4.2). Eight are scale-free networks which have a power-law degree distribution with the max-degree D_{max} shown in the table. These resemble real-life inputs to recommendation systems. To study the effect of input graph structure on performance, we also evaluate four road networks with relatively uniform degree distribution. The column labeled $EL(s)$ in Table 4.2 shows the running times of the EL version of SGD for each combination of input and platform. In the rest of this section, the running times of all other versions of SGD are normalized with respect to the running time of the EL version for that combination of input and platform.

[3]Source code is available from http://iss.ices.utexas.edu.

Table 4.2: Characteristics of the scale-free and uniform inputs. $|V|$ is the total number of vertices in the graph, $|E|$ is the number of edges in the graph, and D_{max} represents the maximum degree of any node in the graph. $EL(s)$ is the running time of the EL versions in seconds.

	$\|V\|$	$\|E\|$	D_{max}	$EL(s)$	
				K40	R9-290X
Scale-free					
STACK	$0.6M$	$0.1M$	6119	0.04	0.18
IMDB	$1.3M$	$3.7M$	1590	0.07	0.38
WIKI	$0.1M$	$5.0M$	100022	0.39	1.04
BGG	$0.1M$	$6.0M$	43331	0.22	0.53
CITP	$7.5M$	$16.5M$	779	0.32	1.69
POKEC	$3.2M$	$22.3M$	14734	0.42	2.3
LIVEJ	$9.6M$	$68.9M$	20293	1.5	7.2
NFLIX	$0.4M$	$99.0M$	227715	2.13	5.28
Road					
CAL	$3.7M$	$4.6M$	7	0.08	0.49
E	$7.1M$	$8.7M$	9	0.19	0.91
W	$12.5M$	$15.1M$	9	0.37	1.58
CTR	$28.1M$	$33.8M$	8	0.9	3.54



(a) NVIDIA K40 **(b)** AMD R9-290X

Figure 4.8: Geomean normalized runtime of scheduling schemes over the two input classes evaluated over two platforms. The runtimes are normalized to EL's runtime. Lower is better.

4.6.1 Overall performance

Since fine-grain synchronization on GPUs is believed to be expensive compared to barrier synchronization, we expected the offline implementations to perform better on both platforms (ignoring preprocessing costs).

Fig. 4.8 presents the running times of the different SGD implementations (ignoring preprocessing costs), normalized to the running time of the EL version.

The first important point to note is that on the K40, the online implementations are best for both scale-free and road networks, even if we ignore the preprocessing cost for the offline implementations.

Fig. 4.8 shows that on the K40, the best online implementation is $1.3\times$ and $2\times$ faster for power-law graphs and road networks respectively than the best offline implementation. In contrast, on the R9-290X, the maximal matching schedule is nearly twice as fast as the best online schedule, a result that is more in tune with

conventional wisdom.

To investigate this further, we measured the throughput of atomic operations on both GPUs [Elteir et al., 2011].Fig. 4.10 shows that for atomic writes to the same location (*i.e.*, atomics with the slowest throughput), the NVIDIA K40 achieves a throughput of roughly 600M atomics/s (nearly 1 atomic a clock) whereas the AMD R9-290X languishes far behind at 45M atomics/s.

This explains why the online implementations perform poorly on the AMD GPU: the NL and EL versions have to do at least one and two atomics per edge respectively, and atomics are relatively slow on this GPU. In contrast, the offline implementations execute a variable, but considerably fewer, number of atomics to implement the device-side barrier synchronization.

Nevertheless, the fact that atomics are relatively fast on the K40 does not explain why the EL version performs so much better than the offline ones for power-law graphs *even though it performs much more fine-grain synchronization*.

The explanation for this counterintuitive behavior is the following. Since offline schedules are based on matchings, they can process at most one edge connected to a given node between successive barriers. Let d_m be the number of edges connected to the highest degree node N in the graph. An offline schedule must have at least d_m matchings, so if p is the average time for processing an edge, the execution time of the program is at least $d_m(p + b)$ where b is the cost of a barrier.

In an online schedule on the other hand, it is possible for several edges connected to node N to be processed between successive barriers due to optimistic

concurrency. If on the average, a fraction f of edges connected to N are processed in each step and the cost of fine-grain synchronization to process one edge is l, the time to process all the edges connected to N is at most $(d_m f(p+l)+b)/f$ since it will take $1/f$ steps to process all the edges connected to N. This can be simplified to $d_m(p+l)+(b/f)$. The first term is the cost to process the edges, and the second term is the cost of barrier synchronization.

Therefore the relative costs are:

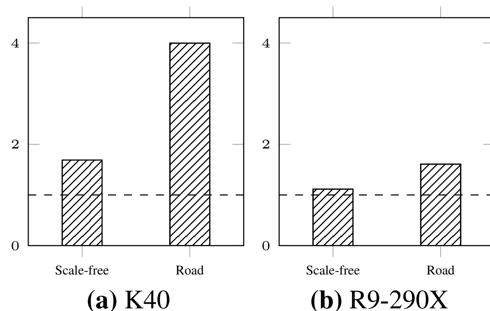
$$d_m p + b(d_m) \ \text{ vs. } \ d_m(p+l) + b(1/f)$$

If l, the cost of fine-grain synchronization, is very high, the reduction in barrier synchronization may not pay off, as on the R9-290X. However, if fine-grain synchronization is not very expensive and the online schedule can process multiple edges from high-degree nodes in each step, the total cost of barrier synchronization is lowered substantially, and the online schedule wins like on the K40.

Offline implementations Ignoring preprocessing time, the diagonal-based schemes are slower than the maximal matching schemes on the K40[4]. This is expected because the diagonal schedules process fewer edges between successive barriers, so they also execute more barrier operations.

However, if preprocessing time is taken into account, the diagonal schedules are faster than the matching-based versions for scale-free graphs. Thus, when pro-

[4]As our compiler produces CUDA code, we were unable to run these on the AMD GPU.



(a) K40 **(b)** R9-290X

Figure 4.9: Speedup of hybrid schedule over EL on the different GPUs across the two input classes. Higher is better.

cessing scale-free graphs once (or if the graph structure changes), diagonal schedules should be preferred over the offline matching-based versions.

4.6.2 Hybrid schedules

Fig. 4.8 also reveals that on the K40, while the online schemes outperform the maximal matchings schemes, the best-performing schedule varies by input class. EL performs better on scale-free inputs while NL suffers from load imbalance as it processes edges of a high-degree node serially which outweighs the benefit from shared memory reuse. However, NL performs better on road networks as it is able to better utilize the locality by scheduling nodes to threads. Since the degree of nodes in a road network is uniform and small, the overhead of scanning the edge-list of each node on each pass is small. We could choose between EL and NL using input characteristics by using a framework such as Nitro [Muralidharan et al., 2014]. Alternatively, a hybrid schedule could be used.

We investigated such a hybrid online schedule which runs NL as the first

pass and processes all the remaining edges with EL schedule. NL processes most of the edges while EL processes the remaining edges. NL also exploits shared memory. This combination of schedules produces better performance on both the scale-free networks as well as road networks compared to a single online schedule as shown in Fig. 4.9.

We also investigated if combining an online scheme with an offline scheme could improve overall performance. Essentially, we observed that the performance of maximal matchings schedule is limited by the highest degree nodes – the edges of these nodes must occur in different matchings and hence the highest degree degree node determines the length of the critical path. Therefore, we built a hybrid schedule which processes a set of high-degree nodes using an EL schedule and the remaining nodes using SGM. Unfortunately, while this improves the performance of SGM, the performance of EL is severely affected, since the high-degree nodes exhibit a large number of conflicts amongst themselves.

4.6.3 Offline schedules

On scale-free inputs, AGM-E performs best amongst the maximal matching schemes as it mimics EL without the overhead of locks. AGM-N suffers the most from the high degree nodes in a scale-free graph as all the threads have to go through at least $|M| \geq d_m$ time steps. This overhead is avoided by SGM, which produces better matching based on the number of hardware threads. Road network graphs which have uniformly low degree nodes allow AGM-E, AGM-N and SGM to outperform EL.

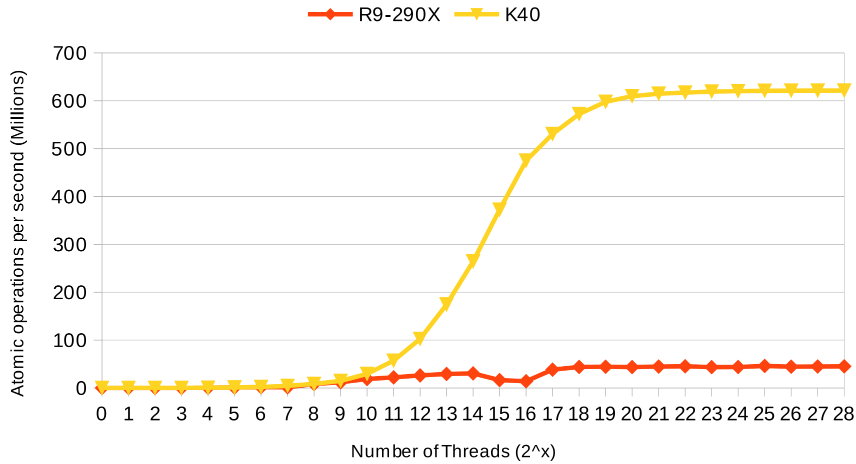


Figure 4.10: Atomic throughput of the NVIDIA K40 and AMD R9-290X. X-axis shows the number of threads launched, and Y-axis shows the throughput of atomics operations *achieved*. The K40 peaks at about 600M atomic operations per second, while the R9-290X saturates at about 45M.

The sparsity of the inputs affects the performance of the diagonal matchings schemes. The matching schedules greedily pack as many edges into a matching set producing a smaller number of matchings compared to a diagonal schedule which produces a matching for every non-empty diagonal.

4.6.4 Heterogeneous schedules

To evaluate SGD on a heterogeneous system, we execute the heterogeneous schedule described in Sec. 4.5 on five real-world inputs. The execution time are shown in Fig. 4.11. Each plot represents the absolute runtime for each input, the different bars show the device configuration, and the x-axis represents different decomposition factors. The inputs are ordered by the number of edges in the graph with STACK having the least number of edges and YAHOO having the most num-

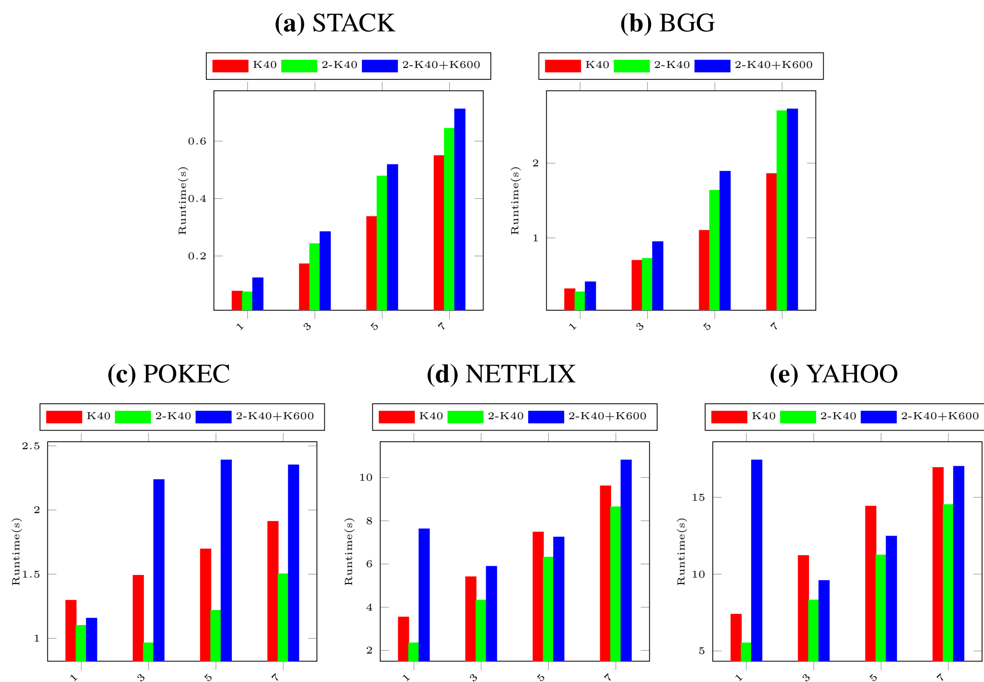


Figure 4.11: Absolute runtime of multi-device SGD on different inputs. Each plot represents an input with varying over-decomposition along the x-axis. Different colors represent different device configurations. Lower is better.

ber of edges.

The two smallest inputs, STACK and BFF, do not show any improvement by increasing the over-decomposition. There is a slight improvement in performance in distributing the computation to two K40 GPUs. Adding a third slower GPU leads to a load imbalance as the the K600 stalls the completion by working on a single partition of edges.

The three inputs POKEC, NETFLIX, and YAHOO shows more improvement when moving from a single device to two K40 GPUs. POKEC shows more reduction in runtime when using an over-decomposition of 3 on the two K40 GPUs. However, when we observe the same factor on NETFLIX and YAHOO, the runtime is higher for two K40 GPUs. This can be attributed to the high maximum degree of each graph ($227K$ and $463K$ respectively). This will lead the row or column containing the maximum degree node on the critical path. For POKEC with a maximum degree of $14K$, the critical path is significantly shorter allowing a 2-device with over-decomposition of 3 to outperform a 2 device execution.

4.7 Data driven algorithms

Graph applications that operate on nodes of the graph and do not modify the structure of the graph (local-computations) can be expressed as vertex programs. The user specifies the activity to be applied to a node and the runtime manages the application of tasks to all the vertices. For a data-driven application, the user also needs to specify the initial work items.

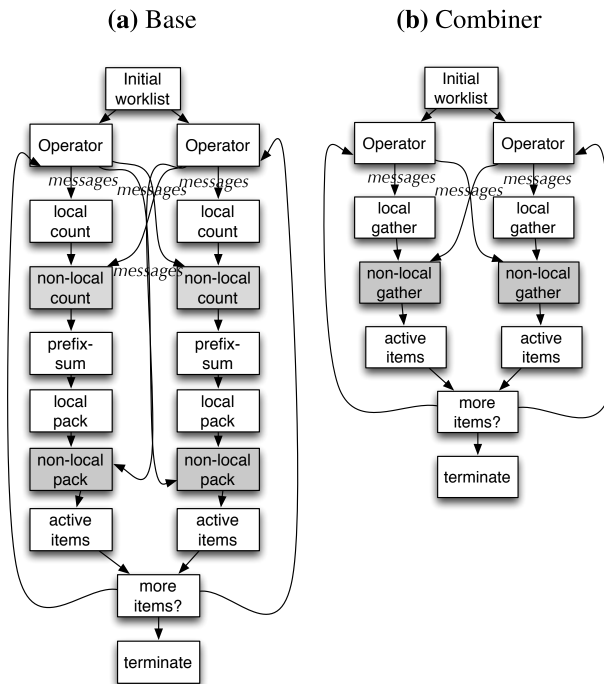


Figure 4.12: Baseline vertex-program implementations for 2 devices. Gray boxes indicate cross-device communication. (a) shows the baseline version, whereas (b) shows the *combiner* version.

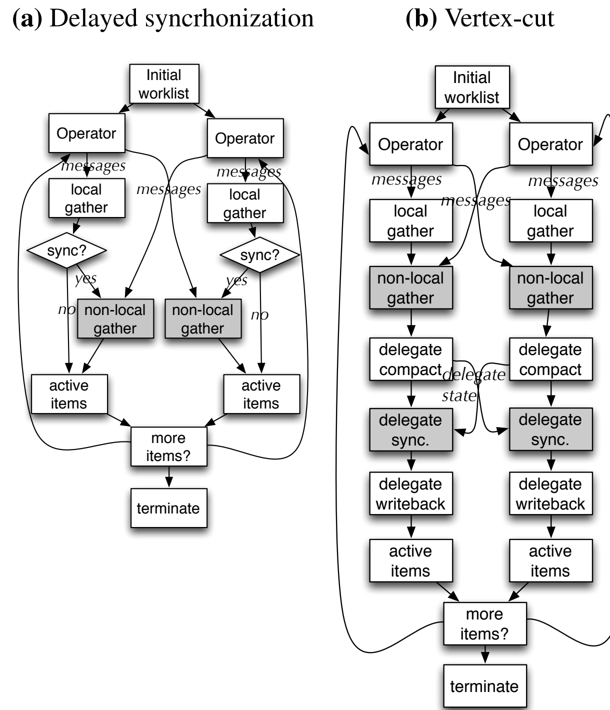


Figure 4.13: Different optimization for vertex-program implementations on 2 devices. Gray boxes indicate cross-device communication. (a) shows the delayed synchronization strategy and (b) shows the vertex-cut implementation.

```
void sssp-operator(Graph & g, Node & n, MsgQ & msg){
   min_dist = INFINITY;
   for(Msg m : msg[n]){
      min_dist = min(msg, min_dist);
   }
   if(n.dist > min_dist){
      n.dist = min_dist;
      for(Edge e : g.out_nbrs(n)){
         msgs.send(n, e.dest, e.wt+n.dist);
      }
   }
}
```

Figure 4.14: A *Pregel* program for single source shortest path.

In the *Pregel* programming model, the program is specified as an operation over a node and a list of messages. Each node communicates with its neighbors by sending messages. The execution of a program proceeds as follows. The operator is applied to the initial set of active nodes. Since there are no messages destined to the active nodes, the message list for each operator instance is empty. The operator instance goes over all the incoming messages, update the node, and if necessary, send messages to its neighbors. The act of sending a message to a node enables the node. These nodes receiving a message will be active in the next round. The runtime has to guarantee that all the messages in the current round will be delivered to their destinations before the next round starts. Termination occurs when no node is active and has no pending messages have to be delivered.

4.7.1 Base implementation

Now we describe a baseline implementation of vertex-programs on a heterogeneous system as illustrated in Fig. 4.12(a). We partition the nodes of the graph across all the devices such that each device has a range of consecutive identifiers. This makes looking up a nodes owner efficient. Each device keeps a copy of the whole graph, though it only need to keep the subgraph incident on the nodes of the its partition. Each device also maintains its own work-list of active nodes. First, each device will execute the operator over the nodes in its work-list. Since the work-list is private to each device, it does not need to check whether the node belongs to its partition. As the operator is applied to the active nodes, messages to neighbors of the node are also generated. Some of the neighbors may reside in the

same partition and others may not. Each device filters the outgoing messages to a local message queue and a non-local message queue.

Once all the items in the work-list have been processed by each device, the messages generated need to be *routed* to their destination. At this point, each device will send a copy of its non-local messages to all the peers. Now, each device d_i counts the number of local messages for each node it has generated for itself by scanning over the local message queue. Next, each device goes over all the non-local messages generated by every other device $d_j, (j \neq j)$ and update the count of number of messages destined for each node in its partition. Once this is done, the device has a count of the number of messages intended for each node in its partition. A prefix sum is performed to compute the indices of the compact incoming messages. This prefix sum is used to populate the incoming messages in a compact manner for each device. Finally, each node with a non-zero message count is added to the active work-list for the next round, and the operator is applied to all the active vertices, and the process repeats. Termination occurs when there are no messages generated for all devices at a particular round.

4.7.2 Combiners

```
1 void sssp-combiner(MsgQ & in_msg, Arr& out_msg){
2    for(Msg m : in_msg){
3       if(is_owned(m.dest)){
4          wl.push_back(m.dest);
5          out_msg[m.dest]= min(m.val, m.dest.dist);
6       }
7    }
8 }
```

Figure 4.15: A *combiner* implementation for single source shortest path.

For many algorithms, we can reduce the overhead of maintaining the messages by applying the combiner to each message. In the base implementation above, we need to copy the messages twice (or have dedicated buffers on each device for every other device) – once for counting the messages, and then after the indices have been computed to place the messages into the correct slots. By applying the combiner as shown in Fig. 4.12(b), we can reduce all the messages destined for a particular node to a single message. An *eager* strategy applies the combiner as soon as the message is generated. However, this requires separate tracking of the destination of the messages, which has to be managed for each device and this will be exchanged. There is a large overhead associated with this since the reduced messages from a device d_i targeted to device d_j are not coalesced, and the whole range of reduced messages has to be copied. A *lazy* approach is to not reduce the messages when they are generated. This is better for data transfer as the amount of data transferred across devices is proportional to the number of messages. Once the messages are delivered to a device, they are combined to a single message. The reduction at the target device also allows the target device to track the active nodes for the next round.

Fig. 4.15 shows the combiner implementation for single source shortest path algorithm. This is called, on the device, after the computation have been performed on all the devices, and the non local messages have been exchanged. Each device goes over all the non local messages it has received from its peers, and checks if a message is intended for a node owned by the current device. For such messages, the destination is added to the work-list for the next round. Note that this can be

implemented by a dense map to avoid duplicates. Since single source shortest path is only concerned with the least incoming edge or message, the combiner updates the value of the message for the destination to be the minimum of the original and the new message. The message can also be stored using a dense array as there will be at most one message for each node. The *out_msg* array however needs to be initialized with the correct value prior to each combiner round (for single source shortest path the *INFINITY* value).

Once the combiner has gone over all the messages from all the peers, including itself, the operator can be applied to the nodes in the work-list, and the process is repeated.

4.7.3 Synchronization

The primary bottleneck in a heterogeneous execution with discrete devices is the communication overhead. Both implementations described above synchronize the messages across all the devices at every round. For some algorithms, it is not necessary to synchronize at every round. Instead, different devices can delay the synchronization of messages without violating the correctness of the program. For example, in the case of single source shortest path, each device can process its local messages right away while accumulating the non local messages over rounds. As long as the non local messages are eventually delivered to the correct destination, the execution will produce the correct results. The benefit of this delayed synchronization is the reduced communication volume which allows devices to independently make progress, which can be very useful when devices are operating

at different speeds; faster devices can execute more rounds in the same time as slower devices. The principal drawback, however, is the wasted work that a device performs only to find out after a synchronization event that a message from a peer device requires the computation to be performed again. For instance, in the case of single source shortest path, suppose a device has started from a node x and propagated the update to several levels of its neighbors over several rounds. Now if upon a synchronization event, the distance of x has to be lowered due to a non local message, the updates propagated earlier are wasted as the new updates have to be propagated.

The algorithm specification can guide as to when a synchronization event should take place. For single source shortest path, we can maintain the minimum and maximum distance computed at each device and compare them across devices to track if any device is computing distances too large, potentially performing wasted work. There are implementation constraints as well which may trigger synchronization. The buffers that store outgoing messages are bounded, and if full should be delivered to peers to be processed. Similarly, if the work-list for a device is empty, it should request a synchronization across all the devices should take place so it does not sit idle.

Fig. 4.13(a) shows the structure of an implementation where synchronization is delayed. The *sync?* box identifies where the decision to synchronize or not is made.

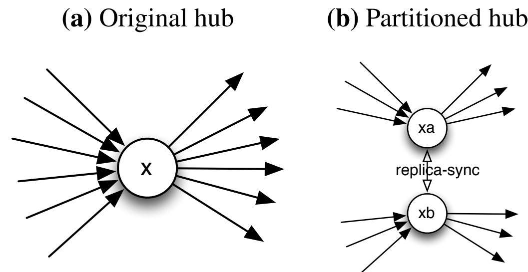


Figure 4.16: Vertex cut implementations. (a) shows a high degree node x, also known as a hub, whereas (b) shows the hub split into two low degree nodes xa and xb.

4.7.4 Partitioning

The implementations described above assign nodes to threads, and each thread goes over the incoming messages (equivalent to the number of incoming neighbors), as well as generate outgoing messages if required (equal to the number of outgoing neighbors). This works well for uniform degree graphs where the number of neighbors of a node does not vary greatly. For scale-free graphs however, the number of nodes with a particular number of neighbors obeys a power-law distribution. This means that most of the nodes have a small out-degree but a large number of high-degree nodes, called *hubs*, also exist. The hub nodes can introduce a large load imbalance on the GPU as the entire work-group has to wait for the thread to terminate. Furthermore, the device assigned these hubs will also suffer from poor performance since the device will take a longer time to generate messages, and if the hub has a high in-degree, also require a large number of messages from other devices.

To alleviate the impact of hubs on the performance of graph applications, we

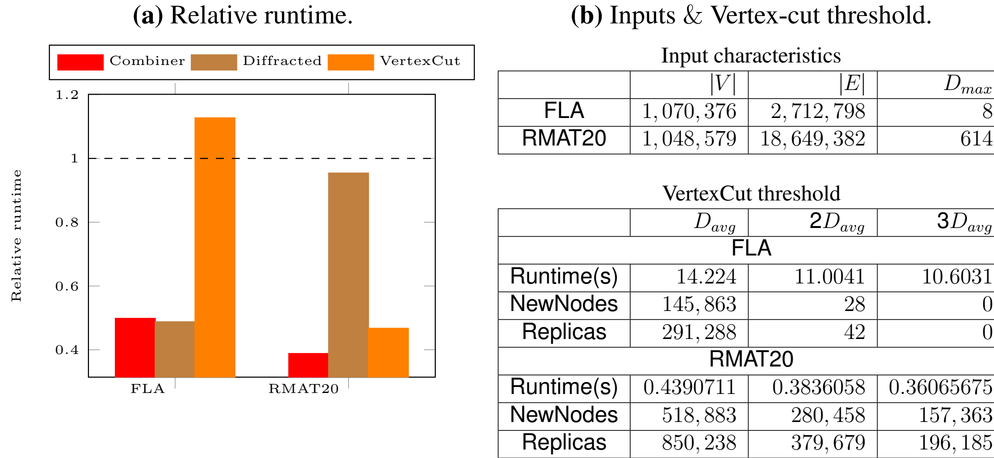
can perform a vertex cut on the graph prior to partitioning. The idea, as illustrated in Fig. 4.16, is to split each hub node x to multiple *delegate* nodes xa and xb, each having a smaller (close to average) number of neighbors. However, now the implementation needs to synchronize the delegates at each round to ensure that all the devices see the same version of the hub.

The synchronization of the delegates appears similar to message exchanges between devices. The delegates can be arbitrarily distributed in the partition of the devices. Furthermore, each delegate has a different number of peer delegates which may be resident on other devices. We address both of these issues by maintaining a separate replica-buffer, and a mapping from the delegates to locations in the replica-buffer. Now we can gather the delegates for each device to its replica-buffer, merge the replica-buffers from different devices and scatter the replica-buffer back to the delegates. This allows the remaining implementation of the system to remain agnostic of the existence of delegates as shown in Fig. 4.13(b).

4.7.5 Results

We compare the relative runtime for different schemes on two inputs – a road network (FLA) with uniform degree distribution and a scale-free input (RMAT20) with power-law degree distribution. The number of nodes, number of edges and the maximum degree of the inputs are tabulated in Fig. 4.17(b). The relative runtimes are displayed in Fig. 4.17(a). The baseline for the runtime is the base version.

The *combiner* version performs significantly better than the base version. This can be attributed to the reduced data-movement across the devices and reduced



(b) Inputs & Vertex-cut threshold.

Input characteristics

	$\|V\|$	$\|E\|$	D_{max}
FLA	1,070,376	2,712,798	8
RMAT20	1,048,579	18,649,382	614

VertexCut threshold

	D_{avg}	$2D_{avg}$	$3D_{avg}$
FLA			
Runtime(s)	14.224	11.0041	10.6031
NewNodes	145,863	28	0
Replicas	291,288	42	0
RMAT20			
Runtime(s)	0.4390711	0.3836058	0.36065675
NewNodes	518,883	280,458	157,363
Replicas	850,238	379,679	196,185

Figure 4.17: (a) Relative runtime for different schemes compared to a base-version. Lower is better. (b)Characteristics of the input and impact of degree threshold vertex-cut performance.

kernel invocations. The number of messages generated across the devices for the base version and the combiner version are very similar as shown in Fig. 4.18 and Fig. 4.19.

The diffracted version performs better than the combiner version on FLA, but only by a very small amount. This is due to the large number of synchronization triggered by a wide graph such as FLA. As discussed earlier, the delayed synchronizations can potentially lead to useful messages being delayed. This leads to more wasted work. The x-axis of the schemes shows the number of passes made for the diffracted version is $6,000$ compared to $3,000$ for all the other version. Given that the number of rounds executed by the implementation doubles, but the overall performance is still better, the utility of diffracted synchronization is valuable if exploited properly. For RMAT20, the situation is similar, but since it is a low diameter

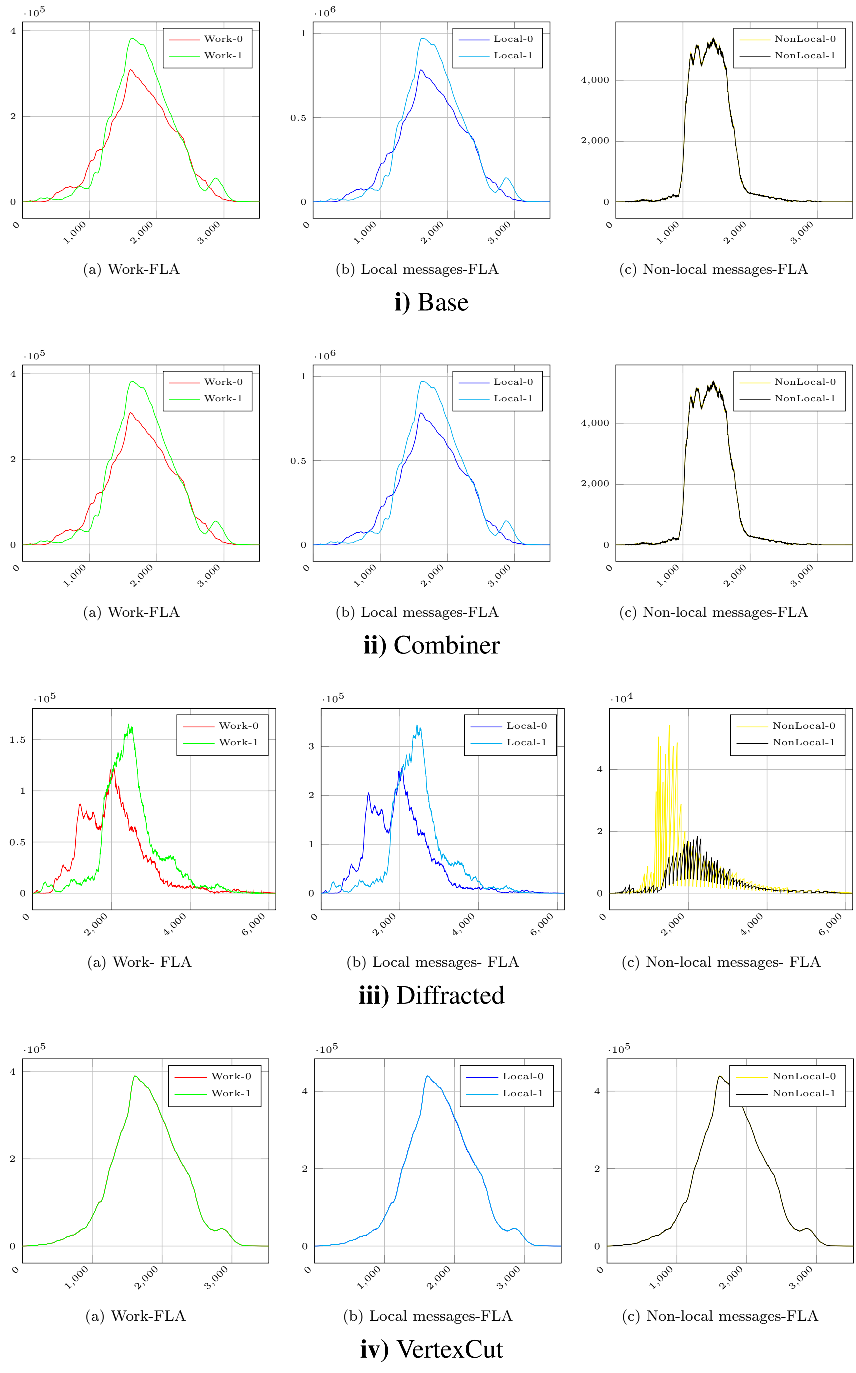
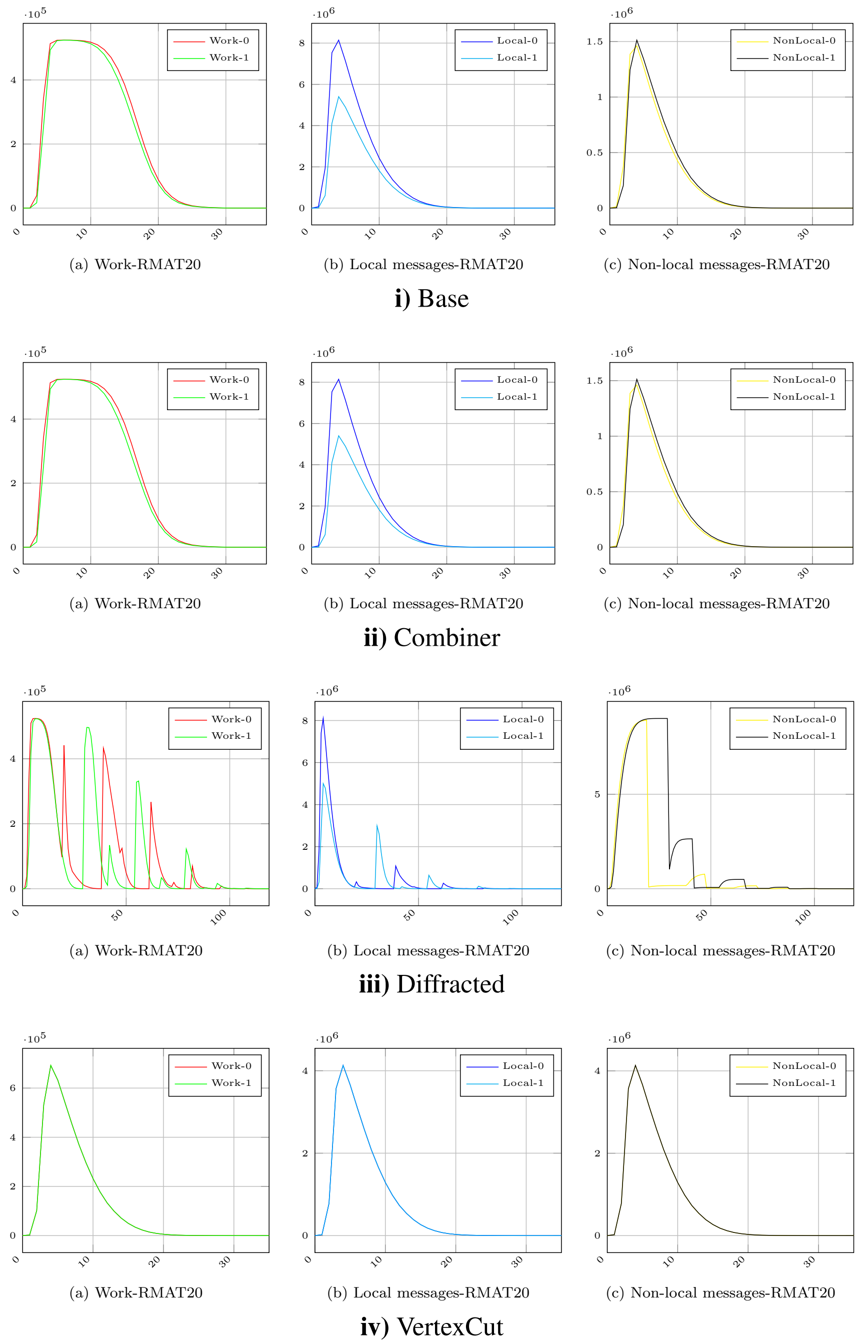


Figure 4.18: Performance metrics for SSSP on two K40s for FLA road input. The time-step is along the x-axis, and the size of the data structure (work queue - left, local messages - center, and non-local messages - right) are shown along the y-axis.



(a) Work-RMAT20 (b) Local messages-RMAT20 (c) Non-local messages-RMAT20

i) Base

(a) Work-RMAT20 (b) Local messages-RMAT20 (c) Non-local messages-RMAT20

ii) Combiner

(a) Work-RMAT20 (b) Local messages-RMAT20 (c) Non-local messages-RMAT20

iii) Diffracted

(a) Work-RMAT20 (b) Local messages-RMAT20 (c) Non-local messages-RMAT20

iv) VertexCut

Figure 4.19: Performance metrics for SSSP on two K40s for RMAT20 input. The time-step is along the x-axis, and the size of the data structure (work queue - left, local messages - center, and non-local messages - right) are shown along the y-axis.

graph, the impact of the delayed propagation of useful messages is larger. Here, the diffracted version takes 100 rounds compared to 35 for other versions.

Finally, for the vertex-cut, we observe that FLA performs poorly. This can be attributed to the overhead in delegate synchronization. Fig. 4.17(b) lists the number of new nodes (NewNodes) and the size of the buffer used for synchronization of delegates (Replicas). For large diameter graphs, this introduces additional synchronization at every step. If we lower the number of new nodes created by the vertex-cut by increasing the threshold from average-degree (D_{avg}) to twice or thrice, the runtime also reduces. For the scale-free input RMAT20, the performance is better than the diffracted state even though NewNodes and Replicas are relatively large. Similarly, the performance can be further improved by increasing the threshold.

4.8 Conclusion

In this chapter, we studied the impact of the choice of synchronization strategy on the performance of two graph applications. *First*, we evaluate SGD, a widely used topology-driven kernel in machine learning. It is a step towards the ultimate goal of providing guidelines to GPU programmers for making implementation choices when coding irregular graph programs. We implemented seven synchronization strategies for this application and evaluated them on two GPU platforms using both road networks and social network graphs as input. The synchronization strategies can be classified as offline strategies and online strategies. Offline strategies pre-process the graph to find independent tasks that can be run in parallel

with barrier synchronization. Online strategies do not require preprocessing and use fine-grain synchronization to ensure that tasks execute atomically. We build on these schedules to investigate performance on a heterogeneous system built with multiple GPUs.

Although conventional wisdom tells us that online strategies are not competitive because of the cost of fine-grain synchronization on GPUs, we found that this was true only on one of the GPUs in our study. On the other GPU, the cost of synchronization was small enough that online schedules could be competitive, and in fact they outperformed offline schedules, particularly for power-law graphs. Furthermore, our results showed that power-law and road networks required different online schedules because of an interaction between load-balancing and locality. This motivated us to invent a hybrid online schedule that dominated the other schedules.

Even on devices with slow atomics, the exact choice of offline schedule is not clearcut. For computations that involve scale-free graphs, customizing the lock-free schedule to the device, as we do with the SGM strategy, to better utilize the hardware, can improve performance significantly.

Next, we evaluate the execution of SSSP, a data driven graph analytics application on a heterogeneous system. For applications such as SSSP, where not all nodes are active in every iteration, synchronization can be tuned specifically for the application. We presented several strategies for reducing the synchronization overhead - reducing messages to be sent, delaying synchronization, and reducing the critical path in a vertex-program.

Chapter 5

Pipeline-parallel execution with FPGAs[1]

The previous two chapters have focused on integrated (Chapter 3) and discrete (Chapter 4) GPUs to accelerate graph applications. Although GPUs are designed to accelerate graphics applications, their architecture is very similar to a general purpose CPU. Both the CPU and the GPU have a Von-Neumann organization, executing instructions through a pipelined structure. An *Field Programmable Gate Array* (FPGA), on the other hand presents a different architecture compared to the CPU. In this chapter we describe accelerating graph algorithms through the use of an FPGA, demonstrating how the versatility in computational resources on a heterogeneous system can be utilized to address bottlenecks for each device.

5.1 Introduction

Heterogeneous platforms containing both CPUs and FPGAs present a unique challenge to the application developer. CPUs and FPGAs have very different performance characteristics as well as programming models. A CPU provides the illusion of sequential instruction execution, which is a natural fit to sequential program-

[1]Initial versions of this work have been presented as *work-in-progress* at *Design Automation Conference*'17.

ming languages. In CPU-based programming, threading and parallelism requires programming effort, whereas sequencing is built into the execution model. An FPGA contains a sea of parallel gates and registers, which is a natural fit to implicitly parallel programming used by hardware description languages like Verilog and VHDL. In FPGA-based programming, parallelism is free whereas sequencing must be implemented using state machines.

Modern CPUs operate at frequencies that are in the GHz range, whereas applications implemented on commercial FPGAs typically operate in the 0.1–0.3 GHz range. Core operations for both integer and floating-point arithmetic have dedicated datapaths in CPUs, and therefore significantly outperform their FPGA counterparts. Memory access is a major performance bottleneck in CPUs. To counteract this, modern processors have deep memory hierarchies with multiple levels of on-chip cache memory. Multiple pending memory requests are also supported via on-chip data structures such as miss status handling registers (MSHRs) [Kroft, 1981], and sophisticated pre-fetching approaches are employed that anticipate future memory access requests using both hardware and software techniques [Hennessy and Patterson, 2011]. When these mechanisms are effective, a naive implementation of the same computation on an FPGA could quickly become performance limited by off-chip memory access. However, an FPGA can compensate for its low frequency using two techniques: (i) using massive parallelism (sometimes referred to as spatial computing), and (ii) customization of the hardware to the application. A single clock cycle of the FPGA might perform a computation that take a CPU tens or hundreds of instructions.

Both CPUs and FPGAs have benefits and drawbacks, which is why heterogeneous platforms include both. However, making effective use of such heterogeneous platforms is a non-trivial task. CPUs and FPGAs have very different programming models, and also very different performance models. These differences present a high barrier to entry for users of heterogeneous hardware.

In this chapter, we explore the use of heterogeneous systems for graph analytics applications. These applications are interesting for parallel computing because although there is a lot of parallelism in graph applications, they are very memory-intensive and perform relatively little computation compared to computational science applications. We illustrate these points in Sec. 5.2 using the *Single Source Shortest Path*(SSSP) algorithm as our running example. One obvious way to use heterogeneous CPU/FPGA systems for graph applications is to divide the graph between the CPU and the FPGA, and let both devices perform the same computations but on different graph partitions. This *Data-parallel execution model* is described in Sec. 5.3.1. This approach off-loads some of the work from the CPU to the FPGA, so the FPGA can be considered to be an accelerator for the CPU. However this approach does not exploit the very different hardware characteristics of the CPU and FPGA described above.

In Sec. 5.3.2 and Sec. 5.3.3, we describe two different approach that we call the *Gather-Apply* and *Apply-Scatter* execution model respectively. In the *Gather-Apply* model, the FPGA marshals data from memory and performs some computations, leaving the CPU to work only on the compute-intensive part of the application. Intuitively, this system architecture treats the CPU as an accelerator for the

FPGA! Alternatively, for applications that perform scatters, the traditional *offloading* of writes to the FPGA accelerates the execution of these graph applications. In Sec. 5.4, we evaluate the presented approaches on four graph analytics applications, comparing performance with that obtained from a conventional Data-parallel approach. Our results show that this approach performs substantially better than the Data-parallel approach, and that the amount of performance improvement depends on the characteristics of the input graph and algorithm. Sec. 5.5 presents a summary of our findings.

5.2 Bottleneck Analysis

We now turn to observing the execution of the two different variants of *Single Source Shortest Path* (SSSP), pull-topology and push-topology, on two different classes of inputs. These variants are described in detail in Sec. 2.4 The classes of inputs represent different graph properties. The LKS input represents a road network characterized by uniform degree distribution. The RMAT18 input represent a small-world input, where the degree distribution follows a power-law distribution. We evaluate the applications on a CPU and measure the micro-architectural counters that represent the pipeline stalls (frontend and backend stalls). These are plotted in Fig. 5.1, where each bar represents the value of the measured metric across the kernel.

The results for the pull implementation, as described in Fig. 2.7, are shown in Fig. 5.1(a) for the two inputs. The bars labelled Pull represent the CPU implementation - the second bar will be discussed later in Sec. 5.3.3. The first observation

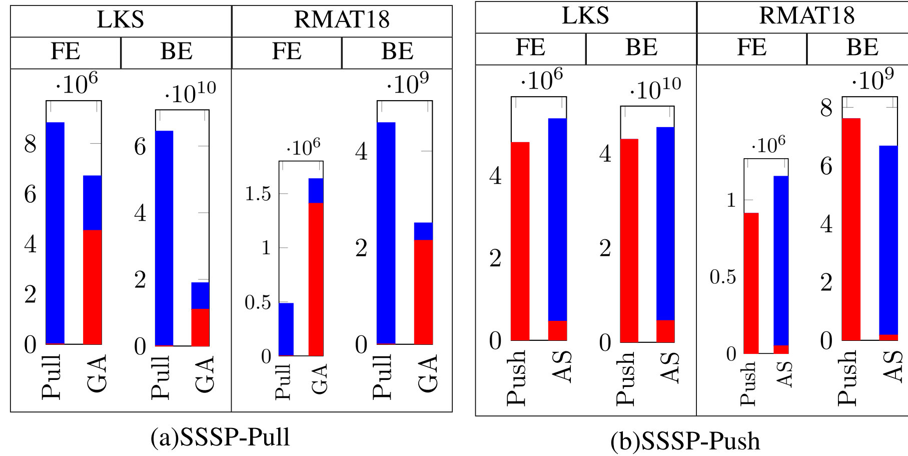


Figure 5.1: SSSP front-end and back-end stalls for pull and push versions on scale-free and road network inputs. The bars for GA and AS shows the breakdown of metric for the two phases using different colors.

we can make is the difference in the number of front-end(FE) and back-end stalls (BE). The back-end stalls are about 3 orders of magnitude greater than the front-end stalls. This suggest that the key performance bottleneck is in the operand fetch phase for the pipeline. The best performance gain can be obtained by addressing the back-end stalls instead of the front-end stalls. The result for the push implementations, as described in Fig. 2.10, are shown in Fig. 5.1(b) show a similar difference in the front-end and back-end stalls.

5.3 Execution Models

Given a graph algorithm such as Single Source Shortest Path, the device level parallelism inherent in a heterogeneous system can be exploited to speed up computations. We explore different strategies to distribute computations on such

```
1  void gather(Graph g){
2     for(int e=0; e< g.num_edges(); ++e){
3        int src = g.neighbors[e];
4        buf[e].first =g.node_data[src].dist;
5        buf[e].second=g.edge_data[e];
6     }
7  }
8  void apply(Graph g){
9     for(int n=0; n<g.num_nodes(); ++n){
10      int min_dist=INT_MAX;
11      for(int e=g.indices[n];e<g.indices[n+1];++e){
12        min_dist = min(min_dist, buff[e].first + buff[e].second);
13      }
14      g.node_data[n].dist = min(min_dist, g.node_data[n].dist);
15    }
16 }
```

Figure 5.2: A simple gather-apply implementation for Single Source Shortest Path.

devices next.

5.3.1 Data-parallel Execution

In the data-parallel execution model, all the hardware devices in a heterogeneous system are treated as similar computational resources. Work is divided between the devices by partitioning either nodes or edges between them. Thus all devices in the system perform the same type of computation, although more powerful devices may have more work assigned to them than less powerful ones. For example, in a system with an FPGA and a CPU, graph nodes can be partitioned into two sets: one processed by the FPGA, and the other processed by the CPU.

For graph-applications such as SSSP, the unit of work is more accurately approximated by the number of edges. Hence, a better strategy may be to partition the graph based on the number of edges. Partitioning based on nodes works well for uniform-degree graphs like those found in road networks, whereas partitioning

```
1 void gather(Graph g){
2    for(int e=0; e< g.num_edges(); ++e){
3       int src = g.neighbors[e];
4       buf[e]=g.node_data[src].dist+
5              g.edge_data[e];
6    }
7 }
8 void apply(Graph g){
9    for(int n=0; n<g.num_nodes(); ++n){
10      int min_dist=INT_MAX;
11      for(int e=g.indices[n];e<g.indices[n+1];++e){
12         min_dist=min(min_dist, buff[e]);
13      }
14      g.node_data[n].dist = min(min_dist, g.node_data[n].dist);
15   }
16 }
```

Figure 5.3: An optimized implementation of Single Source Shortest Path where some computation is performed in the gather implementation reducing the overall traffic to the apply.

based on edges works well for scale-free graphs where the edge distribution follows a power-law behavior.

We will explore both pull and push based data-parallel implementations for graph applications. Note that the push based implementations require atomic updates to be performed to the destination of an edge whereas the pull based implementations do not have concurrent writes.

As described earlier, the FPGA and the CPU have very different strengths and weaknesses. The data-parallel execution model is fairly oblivious to these differences. Ideally, we would like the execution model to optimize the utilization of the capabilities offered by each device.

5.3.2 Gather-Apply Execution

We start by observing the different actions required on the edges and the nodes in the graph. For Single Source Shortest Path pull implementation, the length of the path from each destination along an edge is collected, and the minimum of those distances is used to update the source of the edge. These two different access patterns can be separated, since the node loop, for a topology-driven algorithm, is regular. The inner loop, which goes over the edges, is irregular as the destination of the edges are not sequential. We can break the algorithm into two parts. The *first* part, which we call gather, goes over the edges and reads the contribution from each edge, placing them in a buffer sequentially. The *second* part, which we call apply, goes over the buffer sequentially, accumulating the updates for each node, and then applying them to the source node. This eliminates the irregularity encountered when accessing the destination of an edge.

Once we have rewritten the algorithm using the gather and apply as shown in Fig. 5.2, we can profile the execution of Single Source Shortest Path on the two inputs LKS and RMAT18. Fig. 5.1 shows the results of profiling the gather-apply (bars labelled GA) implementation and the serial implementation (bars labelled Pull). The gather-apply results are broken down into the two components representing the gather(red) and apply(blue) phase respectively. We note that while the overall front-end stalls do increase, the back-end stalls decrease. Furthermore, most of the back-end stalls are in the gather(red) phase. Recall that the gather phase performs the irregular accesses, whereas the apply phase performs a sequential update of the nodes hence reducing the back-end stalls.

The execution time for a gather-apply is higher than a serial implementation, which we discuss in more detail in Sec. 5.4. But we note that executing the two phases sequentially is not a good idea. Similarly, executing the two phases concurrently on different cores in a multi-core system will not address the performance bottleneck - the irregularity in accessing the destination of the edges. For an SoC equipped with an FPGA and a multi-core CPU, we can utilize the FPGA to perform the gather and let the mutli-core CPU perform the apply.

The FPGA's design, without a hardware cache, suits the irregular access patterns for the destination of the edges. By issuing multiple memory requests, we can keep many memory requests in flight and convert the irregular accesses to a regular access. The gather phase on the FPGA can perform these irregular accesses, and convert them to sequential accesses by appending the values returned from memory into a buffer. The apply phase can perform the computation by going over the buffer instead of the graph nodes themselves, avoiding the irregular accesses.

In case of Single Source Shortest Path, the irregular accesses read the dist field of the source of the edge as well as the edge weight. The gather phase of the partitioned operator goes over all the edges, and for each source of the edge, appends the two components (dist, wt) to the buffer as described in Fig. 5.2. The apply phase goes over all the nodes, and for each entry in the buffer corresponding to an incoming edge populated by the gather phase, applies the update. This converts the irregular access to a sequential access for the apply phase.

For an application such as Single Source Shortest Path, we also observe that the apply phase performs an addition on the two values read from the buffer.

Instead of storing both fields and performing the addition in the apply phase, we can perform the addition in the gather phase. This reduces the amount of data communicated between the two phases while simultaneously shifting computation from the apply to the gather phase. The pseudo-code for Single Source Shortest Path with this optimization is shown in Fig. 5.3. As described in Table 5.2, many graph applications have both computations per-edge as well as computations per-node. These computations can be moved between the gather and apply phases depending on the relative cost of data movement.

5.3.2.1 Implementation choices

Like the data-parallel execution model, the Gather-Apply execution model can be implemented on the CPU alone, on the FPGA alone, or on both the CPU and FPGA. However, the trade-offs are quite different.

Performing Gather followed by Apply on the CPU alone is unlikely to improve performance. While it does reduce the irregularity of accesses for the apply phase, the total number of memory accesses increases as values are written to and read from the buffer.

If both phases are implemented on the FPGA, there are several choices for implementing the buffer. We observe that the buffer needs to satisfy only one requirement: it should implement *first-in first-out* behaviour. This is necessary as the apply phase expects values in the same order as in the neighbors array. While Fig. 5.2 and Fig. 5.3 use an array, any data structure, such as a queue will work just as well. In fact, for an implementation where both the gather and the apply are

executed on the FPGA, we rely on OpenCL channels to communicate the gathered values. However, for a heterogeneous implementation, where such a channel is not available for communication, we rely on an array backed by coherent memory for communication.

For a heterogeneous systems equipped with a multi-core CPU and an FPGA, we can perform the gather on the FPGA and perform the apply on the CPU. This may be beneficial because the performance of the irregular accesses on the CPU is detrimental to performance, and by utilizing the FPGA to convert these to sequential accesses, the CPU can focus on performing the apply efficiently.

By partitioning the computation into gather and apply, we can also assign some arrays of the CSR representation of the graph to different devices instead of sharing them across both devices. Specifically, the neighbors and edge-data arrays are only accessed by the gather phase, whereas the indices array is only accessed by the apply phase. This permits the data-structure to use non-coherent memory for these data structures, and only the node-data and the shared buffer need to be backed by coherent memory as they are accessed by both phases. However, in a bulk-synchronous parallel (BSP) implementation, the apply phase can write to its private copy of node-data and commit/merge the changes at the end of a phase.

Finally, we note that we can increase the utilization of the CPU by partitioning the graph into multiple partitions and having a different thread perform a gather–apply phase pair.

5.3.3 Apply-Scatter implementation

As described earlier, `push` work-list driven algorithms are more work-efficient for many graph algorithms such as Single Source Shortest Path. While this is an improvement, we observe from Fig. 5.1 that the back-end stalls are significantly higher than front-end stalls. We can perform the same analysis as we did for the pull variants. A push algorithm has a slightly different structure as the computation on the source of the edge is performed first, and the result is propagated to its neighbors.

The presence of the inner loop, which goes over the outgoing edges of a node and updates the value on the destination of the edge, is likely an irregular access. Upon profiling the application, we see a pattern similar to the *pull* algorithms, where back-end stalls dominate the execution time.

We can proceed to decompose the *push* algorithm in a manner similar to the *pull* algorithm described earlier. The key differences here are that the *apply* phase is performed first to compute the update to be sent to the outgoing neighbors, and the operation to be performed over the neighbors is a write compared to a read for a *gather-apply* implementation. A *push* algorithm can also be broken down into two phases - *apply* and *scatter*.

The *apply* phase goes over nodes and computes the updates to be propagated to its outgoing neighbors, appending them to a shared buffer. The *scatter* phase reads the values from the shared buffer and sends them to the outgoing neighbors. Fig. 5.4 shows the pseudo-code for a SSSP work-list driven implementation using

```
1  SharedBuffer scatter_buffer = ;//initialize
2  void sssp_apply(Graph g){
3     for(Node n : g.nodes){
4        new_dist = n.dist;
5        scatter_buff.append(new_dist)
6     }//end for-n
7  }//end sssp_apply
8  void ssp_scatter(Graph g, WorkList wl){
9     for(Node n : g.nodes){
10       new_dist = scatter_buff.pop_back();
11       for(Edge e: n.out_edges() ){
12          int wt = e.wt;
13          if(e.destination.dist>new_dist+wt){
14             e.destination.dist= new_dist+wt;
15          }//end if
16       }//end for-e
17    }//end for-n
18    return ;
19 }//end sssp_scatter
20 ////////////////////////////
21 Graph g(...);//load graph
22 while(){
23    sssp_apply(g,wl);
24 }
```

Figure 5.4: Single Source Shortest Path topology-driven apply-scatter algorithm.

apply-scatter. A topology-driven implementation would eliminate the work-list, and go over all the nodes in both phases.

If we profile the execution of the *apply-scatter* implementation, we immediately observe that the number of instructions and the back-end stalls go down significantly. The decrease in the number of instructions executed is attributed to the work efficiency of a work-list driven algorithm. The topology-driven implementations, both push and pull versions, do approximately the same number of instructions. A work-list driven implementation significantly reduces the number of instructions executed, particularly for large diameter graphs such as road networks where only a small fraction of the nodes are active at any time.

The back-end stalls, as discussed earlier, are due to the cache lockups. We

```
1  SharedBuffer scatter_buffer = ;//initialize
2  void sssp_apply(Graph g, WorkList wl){
3     for(Node n : wl){s
4        new_dist = n.dist;
5        scatter_buff.append(new_dist)
6     }//end for-n
7  }//end sssp_apply
8  WorkList ssp_scatter(Graph g, WorkList wl){
9     WorkList next_wl;
10    for(Node n : wl){
11       new_dist = scatter_buff.pop_back();
12       for(Edge e: n.out_edges() ){
13          int wt = e.wt;
14          if(e.destination.dist>new_dist+wt){
15             e.destination.dist= new_dist+wt;
16             next_wl.push_back(e.destination);
17          }//end if
18       }//end for-e
19    }//end for-n
20    return next_wl;
21 }//end sssp_scatter
22 ////////////////////////////
23 Graph g(...);//load graph
24 WorkList wl(...);//initialize
25 while(!wl.empty()){
26    sssp_apply(g,wl);
27    wl = sssp_scatter(g,wl);
28 }
```

Figure 5.5: Single Source Shortest Path work-list driven apply-scatter algorithm.

Table 5.1: Application node and edge data. The initial work-list sizes for the work-list driven implementations are also shown.

App	WL	Data	
		Node	**Edge**
PR	|V|	$\frac{float:rank}{int:nout}$	void
BFS	1	int:level	void
SSSP	1	int:dist	int:wt
CC	|V|	int:label	void

Table 5.2: Application node and edge computations. The push implementations require atomic updates on the edges.

App	Compute			
	Pull		**Push**	
	Edge	**Node**	**Edge**	**Node**
PR	rank/nout	res*(1-α)+α	add	xchg,+,×
BFS	level	min(res,level+1)	min(res,level+1)	level
SSSP	wt+dist	min(dist,res)	min(res+wt,dist)	dist
CC	label	min(res,label)	min(res,lable)	label

see that decomposing a push-topo implementation into a *apply* and *scatter* shows that majority of the backend stalls are in the *scatter* phase. Note however, compared to a *gather* phase for a pull algorithm, the stalls are larger in a *scatter* phase. This is because the *scatter* contains irregular writes, which means that cache entries have to be written back upon eviction. This is not the case for a *gather* phases as the entries can be dropped since they have not been modified.

5.4 Evaluation

To evaluate the implementation schemes for graph analytics applications described in this chapter, we use a DE1-SoC from Terasic. This is an ARM based

Table 5.3: Inputs and their key properties.

	$\lvert V\rvert$	$\lvert E\rvert$	$\frac{\lvert E\rvert}{\lvert V\rvert}$	D_{in}	D_{out}
Scale-free					
rmat18	262K	4M	16	3,198	1,436,138
rmat19	524K	8M	16	4,016	2,716,495
Road					
LKS	2M	6M	2.49	8	8
E	3M	8M	2.43	9	9

system-on-chip with an Altera Cyclone V SoC. It comprises of a dual-core ARM Cortex-A9 MPCore processor as well as an Altera CycloneV FPGA. There is a 1G DDR3 SDRAM connected via a 32-bit data bus. The kernel has contiguous memory allocator (CMA) support which allows large virtual memories to be mapped to contiguous physical addresses. Memory accesses within a coherency window (512MB region starting at $2000000h$ physical address) from the CPU are intercepted by the snoop control unit (SCU). All OpenCL allocations are served by the coherent memory region, and hence limited to 512MB. The host code is compiled using gcc-4.6.3, and runs on Linux (kernel version 3.13). The device (OpenCL) code is compiled using Altera OpenCL SDK version 14.0.196.

5.4.1 Applications

We evaluate the performance of our scheme using four different graph analytics applications.

1. Page-Rank (PR): computes the rank of each node in a graph representing pages. The edges represent hyper-links from one page to another.

2. Breadth first search (BFS): we use the node with identifier 0 as the start node.

3. Single source shortest path (SSSP): the running example for the chapter.

4. Connected components (CC): a node labelling algorithm in which all nodes in a component are given the same label. We use a label-propagation algorithm in which initially each node is in its own component. In each round, the node scans its immediate neighbors for the minimum label, and assigns the minimum of its neighbors' labels and its own label to itself.

The data associated with the nodes and edges for each application is shown in Table 5.1 and the computations performed in each application are summarized in Table 5.2. The applications also have different number of nodes initially active, as shown in the column labeled **WL**. For SSSP and BFS, only the source node is active initially whereas PR and CC have all the nodes active in the initial phase. This does not affect the topology-driven implementations. We also tabulate the different read and write accesses for both regular and random accesses for topology-driven implementations of applications in Table 5.4. We only account for the node labels and the updates propagated for the *gather-apply* and *apply-scatter* versions.

We use two classes of inputs for the evaluation: *scale-free*(RMAT) graphs and *road*(ROAD) networks. Key features of the inputs are given in Table 5.3. We present results for the smaller of the two inputs. Results on the larger input are similar.

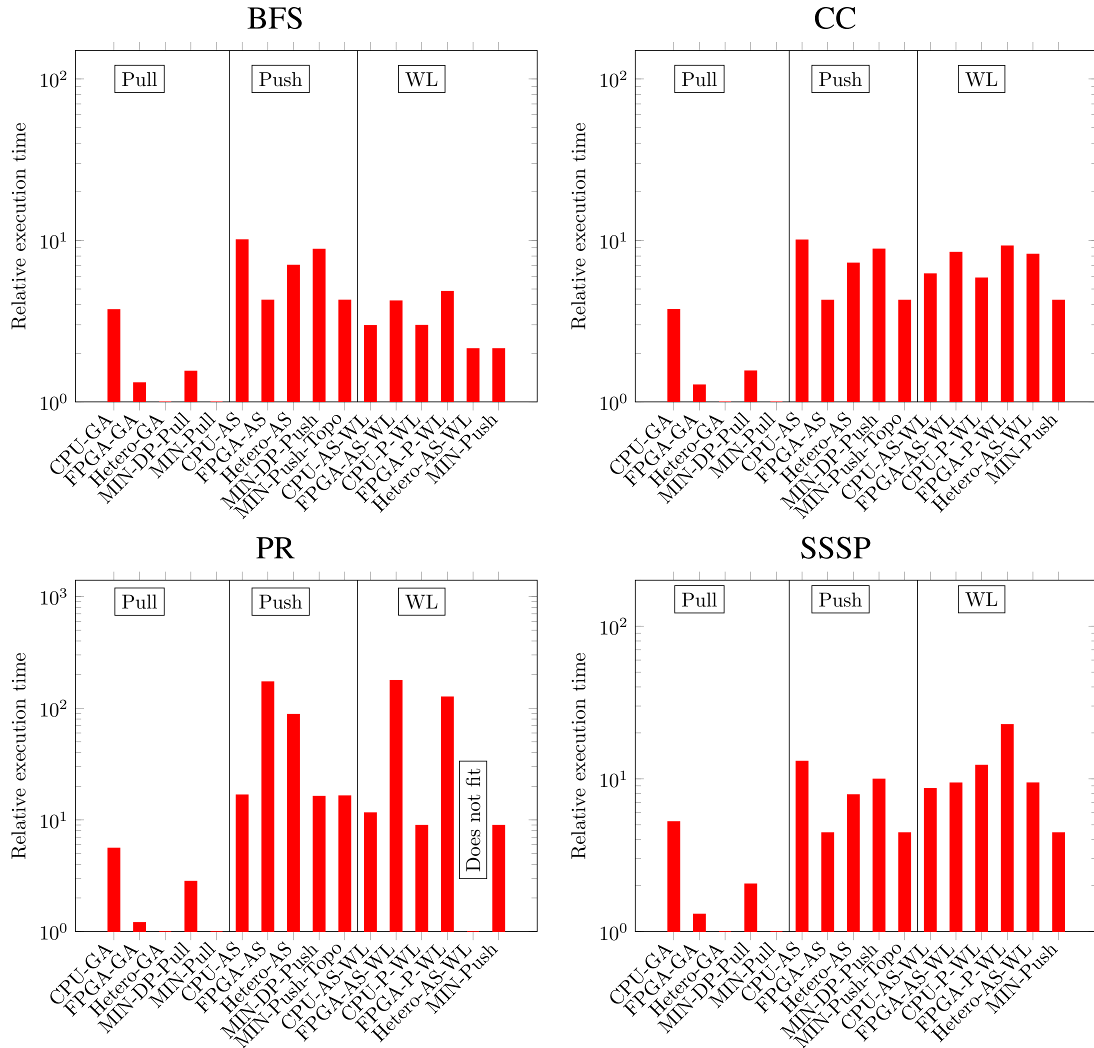


Figure 5.6: Geometric means of relative execution time for all variants of different applications on RMAT networks. The fastest variant is at 1. The logic synthesized for Hetero-AS-WL variant for PR did not fit the area, hence was not executed. Lower is better.

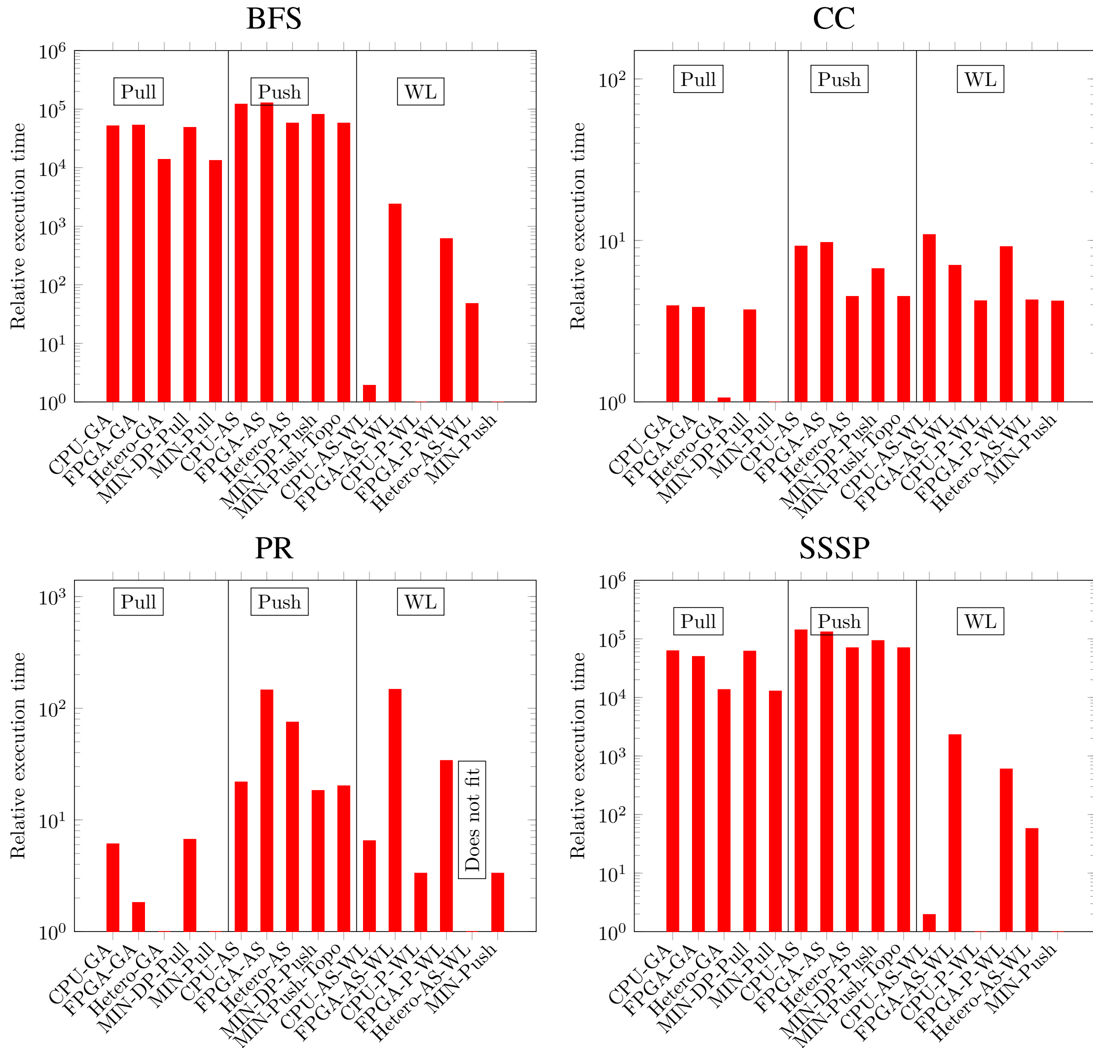


Figure 5.7: Geometric means of relative execution time for all variants of different applications on ROAD networks. The fastest variant is at 1. The logic synthesized for Hetero-AS-WL variant for PR did not fit the area, hence was not executed. Lower is better.

Table 5.4: Differences in the number of regular and random reads and writes for the node labels of each variant.

Variant	Regular		Random	
	Reads	Writes	Reads	Writes
Pull	-	V	E	-
Push	V	-	-	E
GA	E	E,V	E	-
AS	V,E	E	-	E

5.4.2 Overall performance

We first present an overall comparison of the different variants for the four applications. The relative execution time for ROAD are presented in Fig. 5.7 and for RMAT in Fig. 5.6. For PR, the Hetero-AS-WL could not fit on the FPGA we used for the evaluation, hence the results cannot be reported.

The RMAT variant benefits greatly from the reduced synchronization in a *pull* implementation. The *push* variants, as well as the *data-driven* variants suffer from the overhead of synchronization and large work per round respectively.

The ROAD network, two of the applications, BFS and SSSP, benefit from a *data-driven* approach. This is attributed to the work-efficiency of a *data-driven* for these applications on a high-diameter graph such as the road networks. For these applications, a small number of the nodes are active at each round on the execution. In contrast, for CC, a large fraction of the nodes are active during execution, making a *pull* implementation more efficient as the work-list management overhead is eliminated and synchronization is reduced. PR performs best with a *pull* implementation as the push versions uses an atomic add over floats. This is not supported

natively and requires a *compare-and-exchange*e loop over type casted values which presents a large performance overhead.

5.4.3 Data-parallel implementations

We first explore the performance of a data-parallel heterogeneous execution in which graph edges are partitioned between the FPGA and the CPU. Fig. 5.8 show the overall execution time for all applications on one input from the two categories. The applications are labelled on top of the grid, and the input as well as variant (push or pull) is labelled on the left. For each of the 16 plots, the x-axis denotes the percentage of edges offloaded to the FPGA and the y-axis denotes the total execution time in seconds. The overall execution time is the larger of the execution time for the FPGA and the CPU.

We see a similar patter for all *pull* variants on the two inputs. For the RMAT18 input, a larger portion of the edges on the FPGA gives the best performance.This can be attributed to the skewed degree distribution in the graph. For the road network LKS which has a more uniform degree distribution, we see a more balanced distribution of edges between the CPU and the FPGA. This is because of the uniform distribution of nodes as well as edges between the two partitions leading to a more balanced workload.

For the *push* variants, the situation is similar for both input categories for all applications except for PR. PR performs best when executed on the CPU because of the atomic addition of a floating point required as part of the computation. Since residuals are atomically added to the destination, an atomic_add(float) is required.

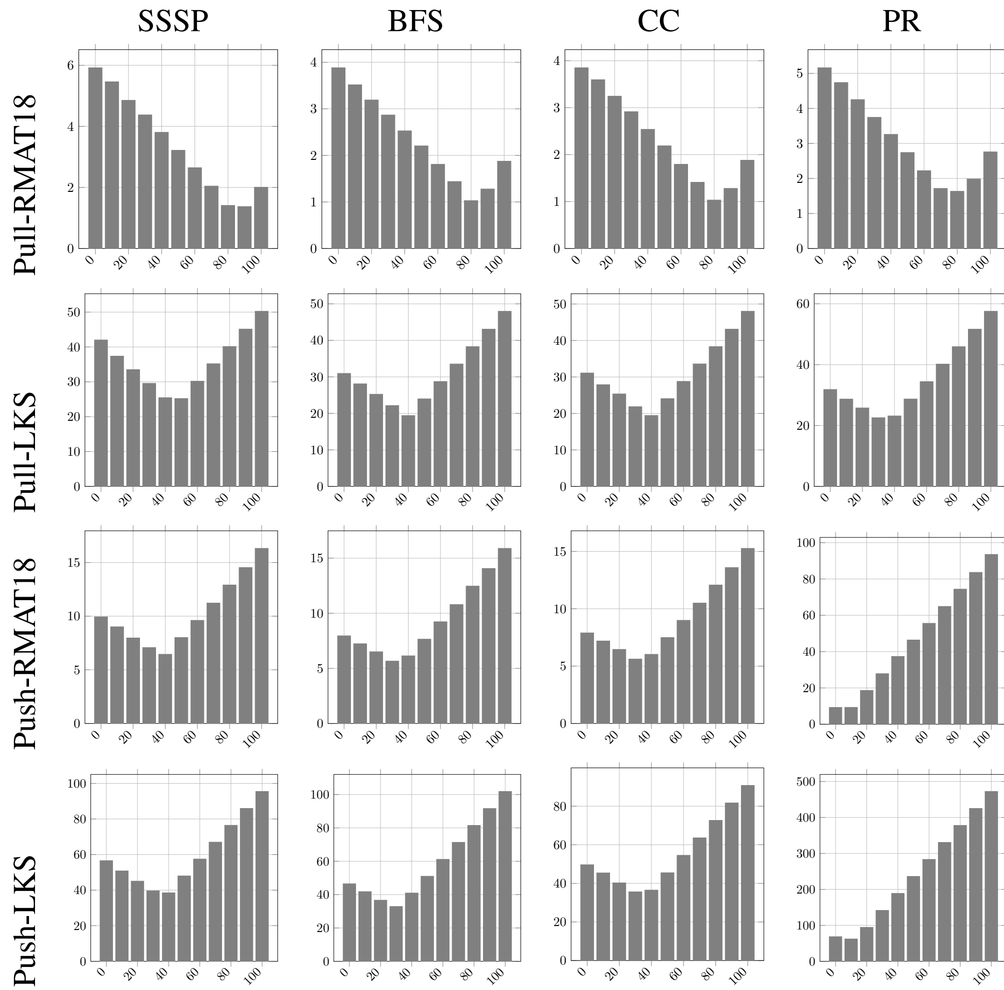


Figure 5.8: Absolute runtime for different edge-distribution of the graph on a heterogeneous system. x-axis shows the percentage of edges processed on the FPGA and y-axis shows the absolute execution time.

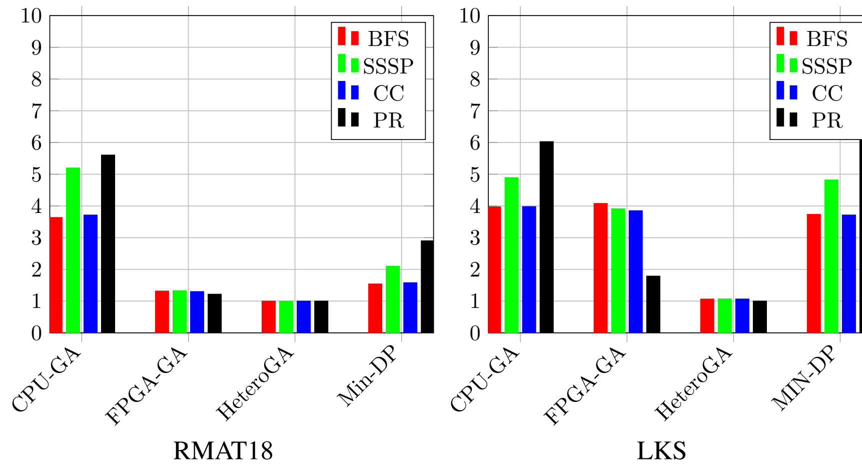


Figure 5.9: Relative execution time for different configurations of gather and apply relative to the best implementation, lower is better.

OpenCL does not provide such a construct natively, so a atomic_cmp_xchg with unions is used instead. Native support of the operation is expected to improve the performance on the FPGA. All the other applications use integer atomic operations which are supported natively in OpenCL.

In general, all the pull variants are faster than the push variants for topology-driven applications because of less synchronization required.

5.4.4 Gather-Apply implementations

There are three gather-apply implementations we consider.

1. CPU-GA: A complete CPU implementation of the gather and the apply where the CPU first goes over the destination of the edges, gathers the results into an array and then goes over the nodes applying the update. This is the baseline

for Fig. 5.9.

2. FPGA-GA: A scheme where both the gather and apply are performed on the FPGA.

3. Hetero-GA: A heterogeneous implementation of gather-apply where the gather is performed on the FPGA and the apply is performed on the CPU. The two phases communicate via a memory buffer in coherent memory. This implementation exploits both cores on the CPU, and moves non-shared arrays to non-coherent memory as described in Sec. 5.3.2.1.

4. MIN-DP: The best execution time for a data-parallel *pull* implementation as shown in Fig. 5.8.

Fig. 5.9 presents the relative execution time for the different versions of gather-apply across all applications for the two classes of inputs. We present the relative execution time for the different combinations separately for each class of inputs (RMAT and ROAD). The baseline for the bar is the best execution time of any version of the algorithm.

We see that the heterogeneous version Hetero-GA outperforms the other versions on both categories of inputs and applications. Compared to the best data-parallel implementation, the additional edge-data access for SSSP benefits most from offloading the gather phase onto the FPGA since the data can be combined on the FPGA and the result shipped to the CPU via a shared buffer. BFS and CC are very similar in performance as the gather portion of these two applications is

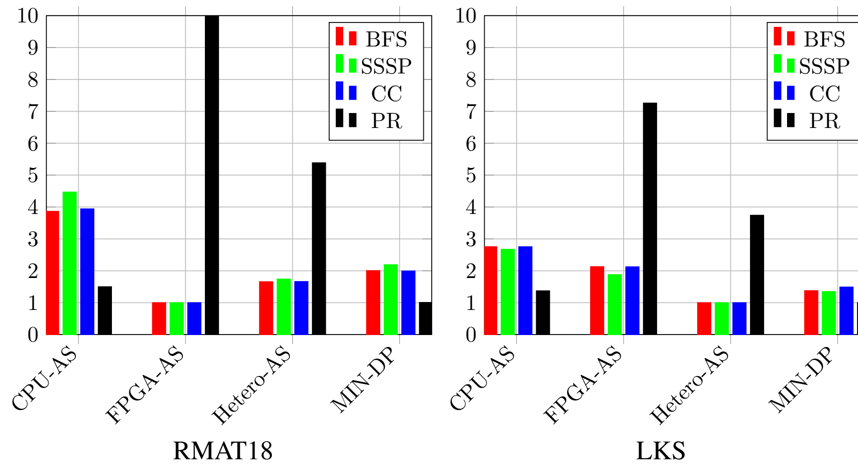


Figure 5.10: Relative execution time for different configurations of apply and scatter relative to the best implementation, lower is better.

identical and the apply is very similar. Here too, shifting the irregular accesses to the FPGA helps the performance.

One feature that stands out for all but PR is the performance of the Hetero-GA and FPGA-GA on RMAT inputs. As described in Table 5.3, the number of edges per node for RMAT inputs is 16 compared to ROAD inputs. Offloading the gather to the FPGA moves a larger number of irregular accesses to the FPGA thus reducing the overall execution time. Hetero-GA implementations provide an average of $7\times$ speedup (average geomean) compared to $6\times$ for FPGA-GA on RMAT inputs. On the other hand, LKS input benefit from the exploitation of multiple threads with average speedup of $4\times$ for Hetero-GA compared to $3\times$ for FPGA-GA.

5.4.5 Apply-Scatter implementations

We consider four different versions of the apply-scatter implementations:

1. CPU-AS: A complete CPU implementation of the apply and scatter phases. The CPU first goes over the graph performing the apply, collecting the updates to be scattered in a buffer. Next, the CPU goes over the shared buffer and performs the writes in a scatter phase.

2. FPGA-AS: A complete FPGA implementation where the apply and scatter are executed on the FPGA. We use a memory buffer for the scatter buffer.

3. Hetero-AS: A heterogeneous implementation where the CPU performs the apply, and the FPGA performs the scatter. Both devices use a common shared buffer.

4. MIN-DP: The best execution time for a data-parallel *push* implementation as shown in Fig. 5.8.

Fig. 5.10 presents the relative execution time for apply-scatter implementations. We observe that PR performs poorly on the FPGA. This, as explained earlier, is due to the atomic addition over floats required in the edge update (scatter). The MIN-DP implementation of PR prefers to offload most of the computation to the CPU.

The apply-scatter versions perform more random reads as shown in Table 5.4. Furthermore, the writes from the CPU to the shared buffer in the apply phase are more costly than the reads for a gather-apply implementation. For

applications other than PR on the scale-free input RMAT18, we observe that the FPGA-AS implementation performs the best. In contrast to the *pull* versions, the *push* implementations require atomic updates to the destinations. This significantly increases the overhead of the edge computation part of the *scatter* operation.

On the road network LKS, we see that the Hetero-AS implementation performs the best for all applications except PR. Here, the workload on the CPU in the apply phase which includes the node computation as well as the writes to the shared buffer, is reduced leading to more balanced execution and hence better performance than the FPGA-SA implementation.

The heterogeneous apply-scatter implementation does not utilize both cores on the CPU as the presence of atomics in the scatter implementation utilizes significanlty more logic. This inhibits instantiation of mutliple instaces of the scatter logic on the FPGA which can be utilized in a multithreaded implementation.

5.4.6 Work-list driven implementations

We also evaluate the *work-list* driven implementations for the applications. We group the applications into two based on the number of items in the initial work-list as shown in Table 5.1.

For the RMAT18 input, there isn't a significant benefit to using a work-list driven implementation on BFS and SSSP because of the small diameter. The benefit of using a work-list is minimal as most of the vertices are active during the execution, instead the overhead of maintaining the work-list dominates the execution. The performance on CC and PR is similar. PR shows poor performance on

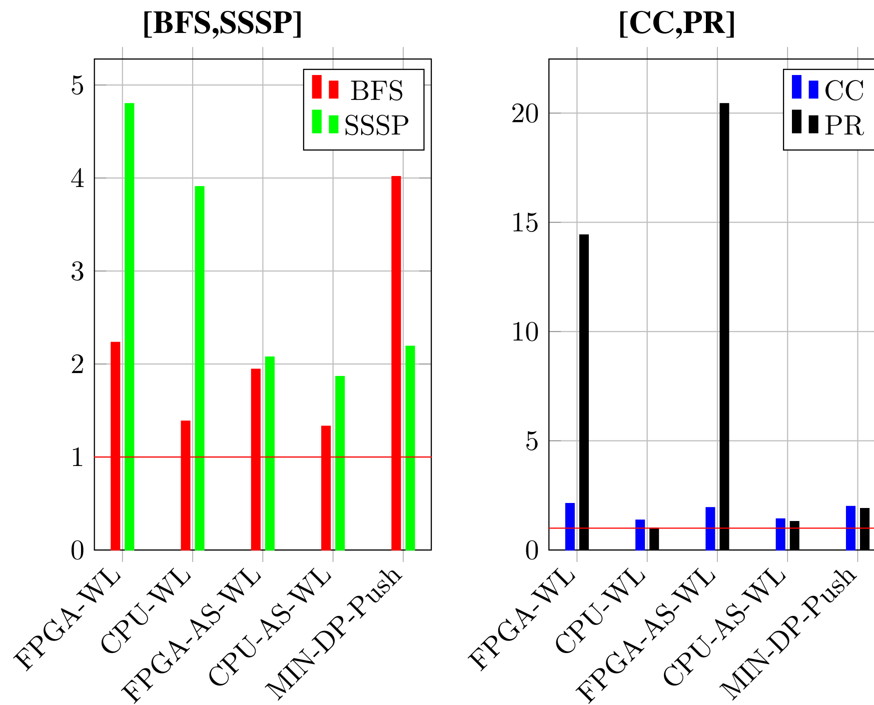


Figure 5.11: Relative execution time for different configurations of apply and scatter relative to the best implementation on RMAT18, lower is better. The *red* horizontal line indicates 1.

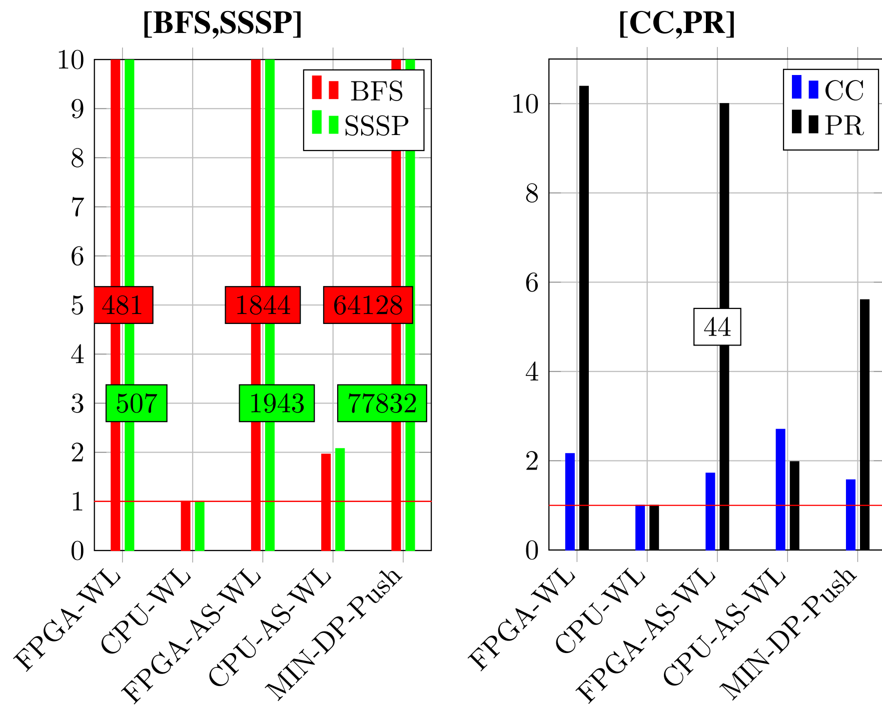


Figure 5.12: Relative execution time for different configurations of apply and scatter relative to the best implementation on LKS, lower is better. The *red* horizontal line indicates 1.

FPGA as already discussed.

The work-list driven implementation show order of magnitude improvement for LKS input on BFS and SSSP because of the large diameter meaning a small fraction of nodes are active for these algorithms.

5.5 Conclusion

With the emergence of heterogeneous systems featuring FPGAs and CPUs, it has become necessary to explore the optimal implementation of applications on these systems. Graph applications present an opportunity for harnessing the heterogeneity in these systems. In this chapter, we described a novel execution model in which the FPGA is a peer of the CPU instead of an accelerator. Graph applications are expressed in a *producer-consumer* model, where one device produces values, while the other consumes it. The roles of the producer and consumer are determined by the application characteristics. For *pull* algorithms, the application essentially runs under the control of the FPGA, and the CPU is treated as an accelerator for compute-intensive tasks. Conversely, in a *push* algorithm, the applications primarily executes on the CPU, streaming updates to the FPGA to be committed to memory. In particular, by performing irregular accesses on the FPGA, we free up the CPU to perform computations more efficiently. Our experimental results showed that this approach does a better job of exploiting the different strengths of components in a heterogeneous system than the more conventional data-parallel execution model in which the FPGA is treated essentially as an accelerator for the CPU.

While coherent memory and accessible APIs such as OpenCL greatly reduce the job of the programmer, our study also suggests that *designers of heterogeneous systems should consider implementing mechanisms that give the FPGA the ability to launch computations on the CPU, since this will allow even greater flexibility in scheduling of computations in a heterogeneous system*. With such support, execution of gather on the FPGA can directly launch the associated apply computation on the CPU, and the CPU can be better utilized by not having to busy poll on the shared buffer. This will not only free up the CPU to perform other computations, but also save power consumption on the CPU. Alternatively, the current *offload* model works sufficiently well for *push* algorithms when expressed as *apply-scatter* instead of the conventional data-parallel implementations.

Chapter 6

Related work

6.1 Graph programming models

The work in this dissertation relies on amorphous data parallelism [Pingali et al., 2011] to express parallelism in graph algorithm. For local computation that can be expressed as vertex-programs, *Pregel* [Malewicz et al., 2010], *Gemini* [Zhu et al., 2016] and *PowerGraph* [Gonzalez et al., 2012] describe frameworks for efficient execution of multi-core systems. Lu et. al. [Lu et al., 2014] compare several large-scale distributed graph computing systems including *PowerGraph*[Gonzalez et al., 2012], *GPS* [Salihoglu and Widom, 2013], and *GraphChi* [Kyrola et al., 2012] and conclude that no single system is the best at large scale. Nguyen et. al. [Nguyen et al., 2013] show that the *Galois* framework can be used to implement most of these systems for shared-memory systems, and provides more flexibility, allowing it to outperform other systems. For large graphs that do not fit into memory, *X-stream* [Roy et al., 2013] describes a streaming engine for graphs.

6.2 GPU programming

Accelerator [Tarditi et al., 2006] describes on of the earliest systems to support general purpose computations on GPU. The system translates side-effect free

operations over arrays to shader programs, which can be executed on the GPU. *Exo-CHI*[Wang et al., 2007] describe an architecture and programming model for GPUs, where some tasks (address translation and exception handling) are executed on the CPU. Owens et.al [Owens et al., 2008] present a comprehensive survey of general purpose computations on the GPU prior to 2008. With the introduction of CUDA, many applications have been ported to the GPU including machine learning [Raina et al., 2009, Catanzaro et al., 2008, Steinkrau et al., 2005, Zastrau and Edelkamp, 2012] and graph analytics [Merrill et al., 2012, Davidson et al., 2014]

MapGraph [Fu et al., 2014] explains an implementation of Pregel for GPUs. *Medusa* [Zhong and He, 2012] provides a more general graph processing framework for local-computations on the GPU. *Gunrock* [Wang et al., 2015] describes a graph processing framework for GPUs. It consists of two key components — advance-traversals and filter-traversals. An advance traversal creates a new frontier of either the current edges or nodes by traversing their neighbors. A filter traversal removes items in the current set using a validation test. The traversal patterns use two optimizations; Merrill [Merrill et al., 2012] (depending on size of work-list, mapping to a thread, warp or block) and Davidson [Davidson et al., 2014] (assigning edges to threads instead of vertices). Pointer analysis has also been ported to the GPU in [Mendez-Lojo et al., 2012] showing the benefit of utilizing GPUs even for irregular applications. Recent work [Egielski et al., 2014] has investigated performance of applications that use GPU atomic constructs which write to one location. A performance prediction based on modeling of multiple GPUs is presented in [Schaa and Kaeli, 2009].

There has also been work on providing higher abstractions on top of *OpenCL*. *VirtCL* [You et al., 2015] describes a single device abstraction for multiple OpenCL devices. Similarly, *Maestro* [Spafford et al., 2010] aims to provide a single device abstraction and dynamically distribute the work items across the difference devices. Abstracting multiple GPUs as a single virtual device [Kim et al., 2011] has also shown to be a useful way to support multiple GPUs.

In order to evaluate the performance of applications on heterogeneous systems, several benchmark suites such as *Rodinia* [Che et al., 2009], *SHOC*[Danalis et al., 2010], and *Paraboil* [Stratton et al., 2012] have been proposed.

6.3 FPGA programming

The memory system is often a bottleneck for algorithms that have irregular data accesses. The Impulse memory controller [Carter et al., 1999] and active memory controller [Kim et al., 2002] both propose feature-rich memory controllers that support scatter/gather operations and improve cache performance. The only communication between the memory controller and host processor is through the standard address/data mechanism, with special addresses used to initiate scatter/-gather operations.

A number of systems attempt to provide support for implementing graph algorithms on FPGAs. GraphOps [Oguntebi and Olukotun, 2016] describes a collection of hardware blocks that can be used by hardware-developers to build graph analytics applications. FPGP [Dai et al., 2016] is a system for vertex centric graph applications on the FPGA. The Lime[Auerbach et al., 2010] language describes

functional extensions to support imperative languages on FPGAs. Prabhakar et. al.[Prabhakar et al., 2016] describe transforms for improving the performance of functional operators on FPGAs. GraphGen [Nurvitadhi et al., 2014] allows programmers to write their graph application kernels and generates logic for vertex programs as well as partitions for specified inputs. None of these systems relies on the heterogeneity in modern CPU/FPGA systems. Recent work[Weisz et al., 2016] has shown the potential for utlizing both the CPU and FPGA for irregular applications, especially for small payload workloads.

[Umuroglu et al., 2015] describes a data-driven implementation of BFS which treats the frontier as a dense vector. They also suggest using the CPU for small frontier sizes. Our approach uses both the CPU and FPGA regardless of the frontier size for data-driven implementations. Our approach also proposes to use both the CPU and the FPGAs for all steps, whereas they switch between the two devices depending on the frontier size.

6.4 Heterogeneous execution

A number of papers describe prior work to allow the same code to run on both the CPU and GPU with minimal programmer effort. *TwinPeaks* [Gummaraju et al., 2010] and *Ocelot* [Diamos et al., 2010] allow GPU code to execute efficiently on CPUs. [Kessler et al., 2012] compares the advantages of different approaches to portability: a library-based approach, a language-based approach, and a component-based approach. [Virlet et al., 2011] describes a task-scheduling model and evaluates it on synthetic tasks across heterogeneous architectures. [Silberstein

et al., 2011] describe runtime assignment of computation to a GPU or a CPU for one specific application from probabilistic networks.

Merge[Linderman et al., 2008] describe a dynamic work-distribution strategy on top of Exo-CHI. Recently, there has been increasing efforts [Luk et al., 2009, Augonnet et al., 2011, Phothilimthana et al., 2013, Song and Dongarra, 2012] to make heterogeneous execution of applications more efficient. In order to determine the ideal work-distribution between the CPU and the GPU, Qilin[Luk et al., 2009] describes a linear performance model and offline profiling to build the model. *StarPU*[Augonnet et al., 2011] describe a system that supports heterogeneous execution of workloads. It includes a data-management library and a runtime for coordinating the execution. The programmer has to provide implementations of the kernel for all of the devices. Dandelion[Rossbach et al., 2013] describes a programming model for programming heterogeneous systems using the *LINQ* API. The implementation generates a set of data-flow graphs where the vertices are abstracted as *PTasks*[Rossbach et al., 2011]. A load balancing scheme for discrete devices, *HDSS*, is presented in [Belviranli et al., 2013], and is similar to convergence based approach described in [Kaleem et al., 2014]. [Ogata et al., 2008] describes a model-driven 2D-FFT scheduling for matrix computations across two devices. [Ravi and Agrawal, 2011] describes a dynamic scheduling strategy for heterogeneous execution for generalized reductions and structured grids based on a cost model. *SKMD* [Lee et al., 2013] supports execution of single kernel on multiple devices. It relies on static analysis of kernels to determine workload distribution.

Making GPUs more accessible to programmers has been an increasingly ac-

tive research area [Che et al., 2011, Barik et al., 2014, Dubach et al., 2012, Rossbach et al., 2013]. Below we discuss prior work that particularly focus on scheduling and load balancing in heterogeneous architectures. There have been increasing efforts [Luk et al., 2009, Augonnet et al., 2011, Phothilimthana et al., 2013, Song and Dongarra, 2012] to make heterogeneous execution of applications more efficient. A load balancing scheme for discrete devices, *HDSS*, is presented in [Belviranli et al., 2013], and is similar to our convergence based approach. [Ogata et al., 2008] describes a model-driven 2D-FFT scheduling for matrix computations across two devices. [Ravi and Agrawal, 2011] describes a dynamic scheduling strategy for heterogeneous execution for generalized reductions and structured grids based on a cost model.

More recently, [Song and Dongarra, 2012] presents a heterogeneous library for executing dense linear algebra. [Phothilimthana et al., 2013] presents a compiler and runtime to address portable performance across heterogeneous systems. They utilize evolutionary algorithms to search optimal algorithms from *PetaBricks* [Ansel et al., 2009] specifications. Pandit et al. [Pandit and Govindarajan, 2014] balance CPU and GPU workload by restricting CPU to executing work in coarse-grain chunks with all CPU threads synchronizing at the end of each chunk. While this approach works well for their regular PolyBench workloads, our work targets both regular and irregular applications.

Ravi et al. [Ravi et al., 2010] use work-sharing to distribute work between the CPU and a discrete GPU. Grewe et al. [Grewe et al., 2013] use machine learning to divide work between the CPU and GPU when there is contention from other

programs. Scogland et al. [Scogland et al., 2012] present several scheduling techniques for systems with discrete devices. In most of these schemes, offloading large chunks to the GPU helps amortize the communication cost. Ravi et al. [Ravi and Agrawal, 2011] describe how to determine the optimal chunk size.

Cederman et al. [Cederman and Tsigas, 2008] and Chatterjee et al. [Chatterjee et al., 2011] address load balancing of workloads across different execution units on a GPU. In particular, they use work-stealing between tasks running on the different streaming multiprocessors (SM) of a discrete GPU. The host CPU populates the initial work-stealing queues. Each SM maintains its own work-stealing queue and steals [Blumofe and Leiserson, 1999] work from other SMs. This is possible due to the availability of an atomic *CAS* operation on the GPU between its SMs. However, since no current hardware supports those operations between the CPU and GPU, this approach does not extend to the general case, in particular to integrated GPUs like ours. [Chen et al., 2010] describe a fine-grained load balancing scheme by running persistent kernels which communicate with the host via task-queues.

6.5 Data partitioning and layout

The 2D partitioning describe earlier is based on [Pearce et al., 2013, Pearce et al., 2014] which addresses scaling performance of graph applications on scale-free graphs. The authors identify two key challenges –high-degree vertices (hubs) and dense communication between partitions. In order to address the scale-free nature of the graphs, the adjacency matrix of the graph is partitioned along both di-

mensions. This can split the edge-list of the hubs across multiple partitions, each of those partitions is an owner of the hubs edge-list. One of these owners is chosen as the master, and all the other partitions contain replicas of the hub. The dense communication is addressed by overlaying a 2D communication network and replacing the point-point communication by a two-hop communication, which perform local aggregation at each level. The programming model is based on a vertex-program, and requires explicit declaration of ghost usage for hubs. The ghosts are not globally synchronized and represent only the local partitions' view or remote hubs. [LeBeane et al., 2015] describe a set of graph partitioning strategies and propose using skewed partitions for heterogeneous systems.

The *Sequoia*[Fatahalian et al., 2006, Knight et al., 2007] system provides a programming model to optimize for locality on large distributed systems with multiple levels in the memory hierarchy. *Dymaxion* [Che et al., 2011] describe general techniques for improving memory accesses on the GPU via data-restructuring and memory-remapping. *G-Streamline* [Zhang et al., 2011] describe dynamic techniques to reduce irregularities in control-flow and memory accesses to improve performance on GPUs. *CuSha*[Khorasani et al., 2014] describes a *GShards*-representation for graph-applications on the GPU based on *Shards*[Kyrola et al., 2012]. The CSR representation is replaced by a partitioned representation where each partition represents the set of incoming edges for a set of nodes, sorted by the source. The DIA format has been parallelized on a GPU [Bell and Garland, 2009] in the context of the SpMV kernel. The block diagonal format has also been used in stencil based solvers for partial differential equations [Lowell et al., 2013]. The derivation of the

blocks containing non-zero entries in BlkDiag is similar to the Block-CSR format in OSKI [Vuduc et al., 2005], where the column and row indices of each entry of the matrix are traversed to identify the corresponding block position of the non-zero entry.

Chapter 7

Future work

The work presented in this dissertation is a stepping stone towards the goal of better utilization of emerging hardware for the masses. The key challenge towards achieving this goal is the need for powerful abstractions, that help domain experts express their algorithm conveniently, that can provide portable performance on the diverse range of hardware platforms.

As hardware architects explore different forms of heterogeneity, isolating the application programmer from the details while allowing them to utilize the benefits of each device is a challenging problem. The *offload* model of computation, where the processor offloads computations to the accelerator, has been very popular. In a data-parallel approach, a dynamic approach provides efficient execution with respect to an off-line optimal, different optimizations can be performed to further increase overall performance. One of the key challenges in a data-parallel execution is to utilize the locality of work - iterations assigned to a device should access data disjoint from other devices. This reduces the overhead of synchronization between the devices. While the iteration space has a certain *optimal* partitioning across the devices, there may exist reordering of the iterations that perform better. For instance, in a graph application, nodes with similar number of neighbors will perform

better on a GPU. Another alternative is to rely on min-cut partitions to reduce the data shared between the different devices, and relying on coherent memory only for shared data thus accelerating accesses to non-shared data.

Supporting graph computations that modify the structure of the graph on the GPU by adding or removing edges and nodes requires memory management addressing the large number of threads on the GPU. One approach, as currently adapted by programmers, is to manually manage memory allocations. An alternate is to support dynamic memory allocation on the GPU, which will allow allocating and deallocating edges and nodes on the GPU.

For heterogeneous systems composed of FPGAs and CPUs, our work highlights the need of having hardware and API support for channels that allow the FPGA and the CPU to communicate efficiently for producer-consumer patterns such as *Gather-Apply* and *Apply-Scatter*. A compiler can be used to split algorithms expressed as a single kernel into the appropriate pair to execute on the heterogeneous system. Furthermore, exposing the graph data-structure to the compiler can allow for a broader range of optimization such as not enforcing coherency for the graph structure in the *Gather-Apply* and *Apply-Scatter* implementations as the structure is not modified for these local-computations.

There are two key limitations of FPGAs make them unattractive for data-center deployment – programmability and performance. Programming FPGA is a tedious and time consuming task, often requiring long compilation/synthesis times. For many application domains, especially those of interest to data-centers, the general-purpose programmability of FPGAs becomes a bottleneck. For instance, Google's

Tensor Processing Unit (TPU) [Jouppi and et al., 2017] is an ASIC specifically designed to accelerate *TensorFlow* [Abadi et al., 2016] programs in the data center. The TPU provides better power-efficiency and performance compared to an FPGA and GPU respectively for TensorFlow programs. Alternatively, Microsoft's *Catapult* [Putnam et al., 2014, Caulfield et al., 2016] uses FPGAs to provide a latency critical platform for applications. However, programming general purpose applications for FPGAs is still a challenging task, with most of the programming on the Catapult done through Verilog. While APIs like OpenCL make programming FPGAs easy, understanding and tuning performance remains a challenging problem as the high level specification is compiled down to *Verilog*. Debugging and performance tuning at the gate level defeats the purpose of programming in high level abstractions. Furthermore, support for multiple kernels executing simultaneously on the FPGA requires the OpenCL kernels to be recompiled as a single unit. Due to the high compilation overhead, this leads to resource wastage when multiple kernels can be executed on the FPGA simultaneously by sharing resources. Support of multiple kernels executing simultaneously on the FPGA requires many features such as dynamic partial reconfiguration and memory protection.

In terms of scaling out graph applications, the work described in this dissertation can be utilized in building distributed systems which utilize a diverse range of processors to accelerate applications. The complexity of non-uniform memory and communication latencies increases the difficulty of efficient execution of applications on distributed heterogeneous platforms.

Chapter 8

Conclusion

Parallel programming on symmetric multi-core processors, and homogeneous distributed systems has been enabled by decades of research and development in both the software stack as well as hardware. Today, an application developer can conveniently write programs that can be executed on multiple threads on a multi-core system using simple abstractions. While this approach works well for simple applications, complex applications, especially those involving irregular memory accesses and control paths require significantly more effort to be implemented efficiently.

Heterogeneous hardware platforms pose a more challenging problem as reasoning about the performance on different platforms can be a strong deterrent to application programmers. As described in Chapter 3, a simple strategy is to build on top of existing abstractions such as the *parallel_for* and provide facilities for the programmer to execute their applications on these heterogeneous platforms. A compiler can transform the applications to native ISAs, and a runtime can dynamically schedule the work-items for efficient execution.

While this approach provides a reasonable black-box solution for programmers, device specific optimizations may be beneficial to programmers looking to

exploit accelerators. As demonstrated in Chapter 4, these optimizations can have a large impact on the performance of irregular applications. Specifically, we show that for graph applications on discrete GPUs, the choice of synchronization strategies is highly dependent on the underlying device as well as input characteristics.

Utilizing heterogeneity in systems without coherent memory, such as those with discrete GPUs, requires special attention to communication patterns. Communication strategies can be tuned through input specific optimizations such as vertex cuts and application specific optimizations such as reductions for message aggregation and delayed synchronization.

Finally, in Chapter 5, we show how increased heterogeneity, where the accelerator design is fundamentally different from the master processor, can benefit applications by addressing the key performance bottlenecks. For graph applications, one of the key bottlenecks is the inability of the memory sub system to handle a large number of irregular memory requests. The FPGA accelerator can be used to perform these irregular accesses allowing the CPU to perform computations. This approach can be conveniently expressed as a *producer-consumer* computations where one device produces a stream of values, and the other device consumes those values.

Bibliography

[Abadi et al., 2016] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.

[Adomavicius and Tuzhilin, 2005] Adomavicius, G. and Tuzhilin, A. (2005). Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.*, 17(6):734–749.

[Ansel et al., 2009] Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A., and Amarasinghe, S. (2009). Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, NY, USA. ACM.

[Auerbach et al., 2010] Auerbach, J., Bacon, D. F., Cheng, P., and Rabbah, R. (2010). Lime: A java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 89–108, New York, NY, USA. ACM.

[Augonnet et al., 2011] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198.

[Barik et al., 2014] Barik, R., Kaleem, R., Majeti, D., Lewis, B., Shpeisman, T., Hu, C., Ni, Y., and Adl-Tabatabai, A.-R. (2014). Efficient mapping of irregular C++ applications to integrated GPUs. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO).*

[Barnes and Hut, 1986] Barnes, J. and Hut, P. (1986). A hierarchical $O(N \log N)$ force calculation algorithm. *Nature*, 324:446–449.

[Baskaran et al., 2010] Baskaran, M. M., Ramanujam, J., and Sadayappan, P. (2010). Automatic c-to-cuda code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg. Springer-Verlag.

[Beamer et al., 2015] Beamer, S., Asanovic, K., and Patterson, D. (2015). Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, pages 56–65, Washington, DC, USA. IEEE Computer Society.

[Bell and Garland, 2009] Bell, N. and Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of SC '09*.

[Belviranli et al., 2013] Belviranli, M. E., Bhuyan, L. N., and Gupta, R. (2013). A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20.

[Bienia et al., 2008] Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT ’08, pages 72–81, NY, USA. ACM.

[big.LITTLE, 2013] big.LITTLE (2013). big.little technology: The future of mobile.

[Blumofe and Leiserson, 1999] Blumofe, R. D. and Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748.

[C++AMP, 2015] C++AMP (2015). Microsoft c++ accelerated massive parallelism.

[Carter et al., 1999] Carter, J., Hsieh, W., Stoller, L., Swanson, M., Zhang, L., Brunvand, E., Davis, A., Kuo, C.-C., Kuramkote, R., Parker, M., Schaelicke, L., and Tateyama, T. (1999). Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, HPCA ’99, pages 70–, Washington, DC, USA. IEEE Computer Society.

[Catanzaro et al., 2008] Catanzaro, B., Sundaram, N., and Keutzer, K. (2008). Fast support vector machine training and classification on graphics processors. In *Proceedings of the 25th International Conference on Machine Learning*. ACM.

[Caulfield et al., 2016] Caulfield, A. M., Chung, E. S., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., et al. (2016). A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE.

[Cederman and Tsigas, 2008] Cederman, D. and Tsigas, P. (2008). On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 57–64, Aire-la-Ville, Switzerland.

[Chatterjee et al., 2011] Chatterjee, S., Grossman, M., Sbirlea, A., and Sarkar, V. (2011). Dynamic task parallelism with a GPU work-stealing runtime system. In *Languages and Compilers for Parallel Computing*, volume 7146 of *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin Heidelberg.

[Che et al., 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA. IEEE Computer Society.

[Che et al., 2011] Che, S., Sheaffer, J. W., and Skadron, K. (2011). Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, NY, USA. ACM.

[Chen et al., 2010] Chen, L., Villa, O., Krishnamoorthy, S., and Gao, G. (2010). Dynamic load balancing on single- and multi-GPU systems. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12.

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

[Dai et al., 2016] Dai, G., Chi, Y., Wang, Y., and Yang, H. (2016). Fpgp: Graph processing framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 105–110, New York, NY, USA. ACM.

[Danalis et al., 2010] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., Tipparaju, V., and Vetter, J. S. (2010). The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 63–74, New York, NY, USA. ACM.

[Davidson et al., 2014] Davidson, A., Baxter, S., Garland, M., and Owens, J. D. (2014). Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, Washington, DC, USA. IEEE Computer Society.

[Dennard et al., 1974] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268.

[Diamos et al., 2010] Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. (2010). Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, NY, USA. ACM.

[Dijkstra, 1965] Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–.

[Dubach et al., 2012] Dubach, C., Cheng, P., Rabbah, R., Bacon, D. F., and Fink, S. J. (2012). Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, NY, USA. ACM.

[Egielski et al., 2014] Egielski, I. J., Huang, J., and Zhang, E. Z. (2014). Massive atomics for massive parallelism on gpus. In *Proceedings of the 2014*

International Symposium on Memory Management, ISMM '14, pages 93–103, New York, NY, USA. ACM.

[Elteir et al., 2011] Elteir, M., Lin, H., and Feng, W.-C. (2011). Performance characterization and optimization of atomic operations on amd gpus. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 234–243, Washington, DC, USA. IEEE Computer Society.

[Fatahalian et al., 2006] Fatahalian, K., Knight, T. J., Houston, M., Erez, M., Horn, D. R., Leem, L., Park, J. Y., Ren, M., Aiken, A., Dally, W. J., and Hanrahan, P. (2006). Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*.

[Fu et al., 2014] Fu, Z., Personick, M., and Thompson, B. (2014). Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA. ACM.

[Garey and Johnson, 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

[Gonzalez et al., 2012] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating*

Systems Design and Implementation, OSDI'12, pages 17–30, Berkeley, CA, USA. USENIX Association.

[Grewe et al., 2013] Grewe, D., Wang, Z., and O'Boyle, M. (2013). OpenCL task partitioning in the presence of GPU Contention. In Rajopadhye, S. and Mills Strout, M., editors, *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg.

[Gummaraju et al., 2010] Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B. R., and Zheng, B. (2010). Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, NY, USA. ACM.

[Ham et al., 2016] Ham, T. J., Wu, L., Sundaram, N., Satish, N., and Martonosi, M. (2016). Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13.

[Hennessy and Patterson, 2011] Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

[Herlihy and Moss, 1993] Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: architectural support for lock-free data structures. In *Proc. of International Symposium on Computer Architecture (ISCA)*.

[Hong and Kim, 2010] Hong, S. and Kim, H. (2010). An integrated GPU power and performance model. *SIGARCH Comput. Archit. News*, 38(3):280–289.

[Jouppi and et al., 2017] Jouppi, N. P. and et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual Symposium on Computer Architecture*, ISCA '17. IEEE Computer Society Press.

[Junkins, 2014] Junkins, S. (2014). The compute architecture of intel processor graphics gen 7.5.

[Kaleem et al., 2014] Kaleem, R., Barik, R., Shpeisman, T., Lewis, B. T., Hu, C., and Pingali, K. (2014). Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA. ACM.

[Kaleem et al., 2015] Kaleem, R., Pai, S., and Pingali, K. (2015). Stochastic gradient descent on gpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 81–89, New York, NY, USA. ACM.

[Kaleem et al., 2016] Kaleem, R., Venkat, A., Pai, S., Hall, M., and Pingali, K. (2016). Synchronization trade-offs in gpu implementations of graph algorithms. In *30th IEEE International Parallel & Distributed Processing Symposium*.

[Kessler et al., 2012] Kessler, C., Dastgeer, U., Thibault, S., Namyst, R., Richards, A., Dolinsky, U., Benkner, S., Traff, J., and Pllana, S. (2012). Programmability

and performance portability aspects of heterogeneous multi-/manycore systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1403–1408.

[Khorasani et al., 2014] Khorasani, F., Vora, K., Gupta, R., and Bhuyan, L. N. (2014). Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA. ACM.

[Kim et al., 2002] Kim, D., Chaudhuri, M., and Heinrich, M. (2002). Leveraging cache coherence in active memory systems. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 2–13, New York, NY, USA. ACM.

[Kim et al., 2011] Kim, J., Kim, H., Lee, J. H., and Lee, J. (2011). Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288, NY, USA. ACM.

[Knight et al., 2007] Knight, T. J., Park, J. Y., Ren, M., Houston, M., Erez, M., Fatahalian, K., Aiken, A., Dally, W. J., and Hanrahan, P. (2007). Compilation for explicitly managed memory hierarchies. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 226–236, New York, NY, USA. ACM.

[Knoop et al., 1994] Knoop, J., Rüthing, O., and Steffen, B. (1994). Optimal code

motion: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155.

[Kroft, 1981] Kroft, D. (1981). Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Kyrola et al., 2012] Kyrola, A., Blelloch, G., and Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA. USENIX Association.

[LeBeane et al., 2015] LeBeane, M., Song, S., Panda, R., Ryoo, J. H., and John, L. K. (2015). Data partitioning strategies for graph workloads on heterogeneous clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 56:1–56:12, New York, NY, USA. ACM.

[Lee and Seung, 1999] Lee, D. D. and Seung, H. S. (1999). Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791.

[Lee and Seung, 2001] Lee, D. D. and Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562.

[Lee et al., 2013] Lee, J., Samadi, M., Park, Y., and Mahlke, S. (2013). Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA. IEEE Press.

[Leiserson, 2009] Leiserson, C. E. (2009). The cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 522–527, NY, USA. ACM.

[Linderman et al., 2008] Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. (2008). Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 287–296, New York, NY, USA. ACM.

[Lowell et al., 2013] Lowell, D., Godwin, J., Holewinski, J., Karthik, D., Choudary, C., Mametjanov, A., Norris, B., Sabin, G., Sadayappan, P., and Sarich, J. (2013). Stencil-aware GPU optimization of iterative solvers. *SIAM J. Scientific Computing*.

[Lu et al., 2014] Lu, Y., Cheng, J., Yan, D., and Wu, H. (2014). Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8(3):281–292.

[Luk et al., 2009] Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In

Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pages 45–55, New York, NY, USA. ACM.

[Macken et al., 1990] Macken, P., Degrauwe, M., Paemel, M. V., and Oguey, H. (1990). A voltage reduction technique for digital systems. In *1990 37th IEEE International Conference on Solid-State Circuits*, pages 238–239.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA. ACM.

[Mendez-Lojo et al., 2012] Mendez-Lojo, M., Burtscher, M., and Pingali, K. (2012). A gpu implementation of inclusion-based points-to analysis. *SIGPLAN Not.*, 47(8):107–116.

[Merrill et al., 2012] Merrill, D., Garland, M., and Grimshaw, A. (2012). Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA. ACM.

[Moore et al., 1998] Moore, G. E. et al. (1998). Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85.

[Muralidharan et al., 2014] Muralidharan, S., Shantharam, M., Hall, M., Garland, M., and Catanzaro, B. (2014). Nitro: A framework for adaptive code variant

tuning. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 501–512, Washington, DC, USA. IEEE Computer Society.

[Nguyen et al., 2013] Nguyen, D., Lenharth, A., and Pingali, K. (2013). A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA. ACM.

[Nurvitadhi et al., 2014] Nurvitadhi, E., Weisz, G., Wang, Y., Hurkat, S., Nguyen, M., Hoe, J. C., Martinez, J. F., and Guestrin, C. (2014). Graphgen: An fpga framework for vertex-centric graph computation. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Field-Programmable Custom Computing Machines*, FCCM '14, pages 25–28, Washington, DC, USA. IEEE Computer Society.

[NVIDIA, 2010] NVIDIA (2010). CUDA technology. `http://www.nvidia.com/CUDA/`.

[O'Boyle et al., 2013] O'Boyle, M. F. P., Wang, Z., and Grewe, D. (2013). Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA. IEEE Computer Society.

[Ogata et al., 2008] Ogata, Y., Endo, T., Maruyama, N., and Matsuoka, S. (2008). An efficient, model-based CPU-GPU heterogeneous FFT library. In *IEEE*

International Symposium on Parallel and Distributed Processing. IPDPS., pages 1–10.

[Oguntebi and Olukotun, 2016] Oguntebi, T. and Olukotun, K. (2016). Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 111–117, New York, NY, USA. ACM.

[OpenACC, 2011] OpenACC (2011). The openacc application program interface.

[OpenCL, 2009] OpenCL (2009). The opencl specification, khronos group.

[OpenCV, 2006] OpenCV (2006). Opensource computer vision library.

[Owens et al., 2008] Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., and Phillips, J. (2008). Gpu computing. *Proceedings of the IEEE*, 96(5):879–899.

[Ozdal et al., 2016] Ozdal, M. M., Yesil, S., Kim, T., Ayupov, A., Greth, J., Burns, S., and Ozturk, O. (2016). Energy efficient architecture for graph analytics accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 166–177, Piscataway, NJ, USA. IEEE Press.

[Pandit and Govindarajan, 2014] Pandit, P. and Govindarajan, R. (2014). Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, NY, USA. ACM.

[Pearce et al., 2013] Pearce, R., Gokhale, M., and Amato, N. M. (2013). Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 825–836, Washington, DC, USA. IEEE Computer Society.

[Pearce et al., 2014] Pearce, R., Gokhale, M., and Amato, N. M. (2014). Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 549–559, Piscataway, NJ, USA. IEEE Press.

[Phothilimthana et al., 2013] Phothilimthana, P. M., Ansel, J., Ragan-Kelley, J., and Amarasinghe, S. (2013). Portable performance on heterogeneous architectures. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 431–444, NY, USA. ACM.

[Pingali et al., 2011] Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M. A., Kaleem, R., Lee, T.-H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., and Sui, X. (2011). The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11. ACM.

[Prabhakar et al., 2016] Prabhakar, R., Koeplinger, D., Brown, K. J., Lee, H., De Sa, C., Kozyrakis, C., and Olukotun, K. (2016). Generating configurable

hardware from parallel patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 651–665, New York, NY, USA. ACM.

[Putnam et al., 2014] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Prashanth, G., Jan, G., Michael, G., Hauck, H. S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Yi, P., and Burger, X. D. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, Piscataway, NJ, USA. IEEE Press.

[Raina et al., 2009] Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09. ACM.

[Ravi and Agrawal, 2011] Ravi, V. and Agrawal, G. (2011). A dynamic scheduling framework for emerging heterogeneous systems. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10.

[Ravi et al., 2010] Ravi, V. T., Ma, W., Chiu, D., and Agrawal, G. (2010). Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM*

International Conference on Supercomputing, ICS '10, pages 137–146, NY, USA. ACM.

[Rossbach et al., 2011] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. (2011). Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA. ACM.

[Rossbach et al., 2013] Rossbach, C. J., Yu, Y., Currey, J., Martin, J.-P., and Fetterly, D. (2013). Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA. ACM.

[Roy et al., 2013] Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA. ACM.

[Rudy et al., 2011] Rudy, G., Khan, M. M., Hall, M., Chen, C., and Chame, J. (2011). A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing*, LCPC'10, pages 136–150, Berlin, Heidelberg. Springer-Verlag.

[Salihoglu and Widom, 2013] Salihoglu, S. and Widom, J. (2013). Gps: A graph processing system. In *Proceedings of the 25th International Conference on*

Scientific and Statistical Database Management, SSDBM, pages 22:1–22:12, New York, NY, USA. ACM.

[Saltz et al., 1997] Saltz, J., Chang, C., Edjlali, G., Hwang, Y.-S., Moon, B., Ponnusamy, R., Sharma, S., Sussman, A., Uysal, M., Agrawal, G., et al. (1997). Programming irregular applications: Runtime support, compilation and tools. *Advances in Computers*, 45:105–153.

[Sbîrlea et al., 2012] Sbîrlea, A., Zou, Y., Budimlíc, Z., Cong, J., and Sarkar, V. (2012). Mapping a data-flow programming model onto heterogeneous platforms. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 61–70, NY, USA. ACM.

[Schaa and Kaeli, 2009] Schaa, D. and Kaeli, D. (2009). Exploring the multiple-GPU design space. In *IEEE International Symposium on Parallel Distributed Processing. IPDPS.*, pages 1–12.

[Scogland et al., 2012] Scogland, T., Rountree, B., chun Feng, W., and De Supinski, B. (2012). Heterogeneous task scheduling for accelerated OpenMP. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, pages 144–155.

[Shavit and Touitou, 1995] Shavit, N. and Touitou, D. (1995). Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA. ACM.

[Silberstein et al., 2011] Silberstein, M., Schuster, A., and Owens, J. D. (2011). Applying software-managed caching and CPU/GPU task scheduling for accelerating dynamic workloads. In Hwu, W. W., editor, *GPU Computing Gems*, volume 2, chapter 36, pages 501–517. Morgan Kaufmann.

[Song and Dongarra, 2012] Song, F. and Dongarra, J. (2012). A scalable framework for heterogeneous GPU-based clusters. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 91–100, NY, USA. ACM.

[Spafford et al., 2010] Spafford, K., Meredith, J., and Vetter, J. (2010). Maestro: Data orchestration and tuning for opencl devices. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, Euro-Par'10, pages 275–286, Berlin, Heidelberg. Springer-Verlag.

[Steinkrau et al., 2005] Steinkrau, D., Simard, P. Y., and Buck, I. (2005). Using GPUs for machine learning algorithms. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition*. IEEE Computer Society.

[Stratton et al., 2012] Stratton, J. A., Rodrigues, C., Sung, I.-J., Obeid, N., Chang, L.-W., Anssari, N., Liu, G. D., and Hwu, W.-M. (2012). Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*.

[Tarditi et al., 2006] Tarditi, D., Puri, S., and Oglesby, J. (2006). Accelerator: Using data parallelism to program gpus for general-purpose uses. In

Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XII, pages 325–335, New York, NY, USA. ACM.

[TBB (Intel Threading Building Blocks), 2011] TBB (Intel Threading Building Blocks) (2011). TBB (intel threading building blocks).

[Umuroglu et al., 2015] Umuroglu, Y., Morrison, D., and Jahre, M. (2015). Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8.

[Venkat et al., 2015] Venkat, A., Hall, M., and Strout, M. (2015). Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 521–532, New York, NY, USA. ACM.

[Virlet et al., 2011] Virlet, B., Zhou, X., Giacalone, J. P., Kuhn, B., Garzaran, M. J., and Padua, D. (2011). Scheduling of stream-based real-time applications for heterogeneous systems. *SIGPLAN Not.*, 46(5):1–10.

[Vuduc et al., 2005] Vuduc, R., Demmel, J. W., and Yelick, K. A. (2005). OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521–530.

[Wang et al., 2007] Wang, P. H., Collins, J. D., Chinya, G. N., Jiang, H., Tian, X., Girkar, M., Yang, N. Y., Lueh, G.-Y., and Wang, H. (2007). Exochi:

Architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 156–166, New York, NY, USA. ACM.

[Wang et al., 2015] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. (2015). Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, New York, NY, USA. ACM.

[Weisz et al., 2016] Weisz, G., Melber, J., Wang, Y., Fleming, K., Nurvitadhi, E., and Hoe, J. C. (2016). A study of pointer-chasing performance on shared-memory processor-fpga systems. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 264–273, New York, NY, USA. ACM.

[You et al., 2015] You, Y.-P., Wu, H.-J., Tsai, Y.-N., and Chao, Y.-T. (2015). Virtcl: A framework for opencl device abstraction and management. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 161–172, New York, NY, USA. ACM.

[Zastrau and Edelkamp, 2012] Zastrau, D. and Edelkamp, S. (2012). Stochastic gradient descent with gpgpu. In *Proceedings of the 35th Annual German*

Conference on Advances in Artificial Intelligence, KI'12, pages 193–204, Berlin, Heidelberg. Springer-Verlag.

[Zhang et al., 2011] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. (2011). On-the-fly elimination of dynamic irregularities for gpu computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 369–380, New York, NY, USA. ACM.

[Zhong and He, 2012] Zhong, J. and He, B. (2012). An overview of medusa: Simplified graph processing on gpus. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 283–284, New York, NY, USA. ACM.

[Zhu et al., 2016] Zhu, X., Chen, W., Zheng, W., and Ma, X. (2016). Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 301–316, Berkeley, CA, USA. USENIX Association.