Accelerating Graph Processing on a Shared-Memory FPGA System

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Department of Engineering

Yu Wang

B.S., Engineering Department, Carnegie Mellon University Pittsburgh

M.S., Engineering Department, Carnegie Mellon University Pittsburgh

Carnegie Mellon University

Pittsburgh, PA

December 2018

ACKNOWLEDGEMENTS

The pursuit of my Ph.D. has been a long and meaningful journey for me. I would not have made it alone without the from many people. This is my attempt to express my gratitude for their contributions. First of all, I would likely to thank my advisor, James Hoe. Since my undergraduate years at CMU, James has constantly provided valuable guidance on my academic work and career direction. He taught me how to think critically, write clearly, and conduct high-quality research. I'm truly grateful to have him as my advisor.

I want to thank Prof. Brandon Lucia, Prof. Franz Franchetti and Dr. Eriko Nurvitadhi for serving on my thesis committee. Their valuable comments and constructive criticisms helped me to refine my thesis work. I would like to especially thank Eriko for his incredible mentorship during my internship at Intel Lab and the technical assistance and inspiration he provided in the early years of my Ph.D. when I know next to nothing about research and FPGAs.

During graduate school, I was fortunate to have the opportunity to collaborate with many brilliant students, researchers and professors. Prof. Mei Chen, Prof. Yaser Sheikh and his students Prof. Hyun Soo Park and Hanbyul Joo are the computer vision experts who

introduced me to the basics of structure from motion during our collaboration on the project of 3-D camera localization. Members of the CALCM group: Eriko Nurvitadhi, Michael Papamichael, Gabriel Weisz and Berkin Akin were great friends and colleagues who helped me in FPGA-related topics as well as polishing my writing and presentation skills. Prof. Tze men Low and Richard Veras provided much-needed feedback on my research work and taught me about graph-processing computing and performance from very different perspectives. Thom Popovich was always patient and helpful with my questions regarding research and thesis writing. I am also grateful for the company of the other CALCM members: Prof. Peter Milder, Prof. Peter Klemperer, Joe Melber, Marie Nguyen, Guanglin Xu and Zhipeng Zhao.

Beyond my life in the lab, I am grateful to and have enjoyed the companionship of my friends during my Ph.D. journey. I want to thank Kevin Chang for his long-time friendship since our college years. Special thanks to Richard Wang and Yvonne Yu for their hospitality during the Thanksgiving dinners and board game nights. I would like to thank Hongyi Xin for his great sense humor and the interesting discussions we had that kept me awake at night. I want to thank Kevin Tang and Andy Liang for the entertaining conversations we had online throughout the years. I also grateful to the great time with my friends in the A-300 office in Hamerschlag Hall: Jin Huang, Jiyuan Zhang, Ke Wang and Yong Zhuang.

I cannot thank my parents enough for their love, encouragement and tolerance. My family members, Wonder, Amy and Tina have provided unconditional support for every step

I have taken throughout my life. They cheered with me during my high times and encouraged me during my low times. Without them, I would not be who I am today.

Finally, I would like to acknowledge Intel Corporation for their generous financial and equipment donation.

ABSTRACT

Graph analytics are tools for determining the relationships among connected components in a graph. The wide variety of applications, including machine learning, social network analysis and computer-vision, mark their increasing importance. Due to the demand for energy efficiency, FPGA-based processing has gained much attention for accelerating graph analytics. However, in comparison to processors, FPGA-based solutions often fall short in terms of processing throughput due to the lack of off-chip memory bandwidth on most commercial platforms. In this thesis, we tackle this problem for FPGA-based graph traversal with two strategies: 1. improving data access throughput by optimizing on-chip data reuse and off-chip bandwidth utilization; and 2. adopting advance scheduling optimizations through a heterogeneous processor-FPGA collaboration.

In particular, in the first part of this thesis, we propose a highly optimized accelerator for Breadth-First-Search. By tailoring the on-chip buffering according to the access patterns of different data types and preprocessing the input graphs using a novel vertex remapping optimization to improve locality, we were able to maximize the on-chip data reuse

and reduce off-chip memory traffic. This allows the proposed accelerator to deliver computation throughputs comparable to state-of-the-art software implementation on a processor with significantly better memory bandwidth and latency. The second part of the thesis addresses an important class of scheduling optimizations that can significantly improve performance but is rarely used in accelerator designs due to the implementation complexity and memory access overhead. We present a heterogeneous processing approach for priority scheduling on a shared-memory CPU-FPGA platform. By exploiting the closely coupled integration of the host processor and the FPGA accelerator, our system dynamically offloads the task of scheduling to a software scheduler on the processor for its programmability, high-capacity cache and low memory latency, while the FPGA graph processing accelerator enjoys the scheduling benefit and delivers higher performance with excellent energy efficiency. To understand the effectiveness of our solution, we compared it with FPGA-only solutions for two scheduling schemes: the well-known Dijkstra scheduling for the Single Source Shortest Path and a new scheduling optimization we developed for improving the data locality of Breadth First Search. Whereas the FPGA-only solution requires an impractical amount of on-chip storage to implement a priority queue, the proposed processor-assisted scheduling that moves the task of scheduling to the processor consumes a negligible load on the processor and retains most of the performance benefit from priority scheduling.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Due to the increasing demand for energy efficiency, there have been large numbers of attempts to use FPGAs for accelerating graph analytics. However, the traditional performance optimization strategy that focuses solely on maximizing parallel memory accesses leads to suboptimal processing throughput for the mainstream FPGA systems that lack off-chip DRAM bandwidth. In this thesis, we demonstrate that high-bandwidth memory sub-systems that are exclusive to expensive high-end platforms is not a necessity for competitive FPGA-based graph-processing. This is achieved by exploiting previously overlooked optimizations on a shared-memory platform, an emerging type of system that offers close CPU-FPGA integration. Specifically, we explore novel hardware and software locality optimizations that are specialized for FPGAs to maximize on-chip data reuse, and we employ fine-grain processor-accelerator collaboration on a shared-memory platform to enable important scheduling optimizations that were previously cost-forbidden to realize on FPGAs. This allows us to reduce the reliance on memory bandwidth and achieve performance comparable to state-of-the-art software solutions on a processor with significantly superior memory bandwidth and latency. As the close CPU-FPGA integration on a shared-memory

system is still a feature waiting to be proven for its commercial value, our heterogeneous collaboration of the two devices can serve as an important use case.

1.1 Accelerating Graph Analytics on FPGAs

FPGA. A Field-Programmable Gate Array (FPGA) is a type of integrated circuit capable of post-manufacturing configuration to its logics. A modern FPGA chip consists or different types of user-configurable components, namely storage devices (e.g., Block Rams, registers), reprogrammable lookup tables (LUTs) that implement combinational or sequential logics, and hardened peripheral/function blocks (e.g., I/O transceivers, DSP blocks). By configuring the functional mappings of the LUTs and the routing of a programmable interconnect matrix to combine those different components, a digital designer can realize large digital designs using FPGAs. Compared to Application Specific Integrated Circuits (ASICs), FPGA's programmability offers a shorter design-to-silicon development cycle. Compared to general-purpose processors that provide functional flexibility by maintaining pre-defined instruction semantics, FPGAs offer programmability by directly mapping computations to hardware, allowing transistors to be directly used for the application-related computations for better hardware utilization and energy efficiency. On the other hand, FPGA is not without limitations. FPGA designs are often dominated by the latency of long interconnect wirings, leading to constrained frequency of the application clock. In addition, the high complexity of hardware debugging and relatively long design compilation process limits the development productivity compared to software-based solutions.

Graph Analytics. Graph analytics are tools for determining the relationships among connected components in a graph. The wide variety of applications, including item recommendation [15], social network analysis [21], computer-vision [18] and so forth, mark their increasing importance. While FPGA accelerators have demonstrated huge success for applications with fixed memory access and compute patterns (e.g., Fast Fourier Transform, dense matrix-vector multiplication), the performance of FPGA-based graph processing is often less impressive. The memory access patterns and control flow of graph applications are governed by the input data. As the sparsity and topology of different graphs vary case by case and greatly change the behaviors of the program, this "irregularity" makes developing efficient, high-performance FPGA-based solutions difficult tasks. Specifically, the main challenges are associated with optimizations in two areas: memory and algorithm.

1.2 Challenges to Optimizing Graph Analytics

Memory Access Optimization. Graph analytics are fundamentally memory-latency-bounded applications due to their low compute-to-data ratio and the pointer-governed data. Traditionally, there are mainly two ways to tackle this bottleneck of memory latency: exploiting memory-level parallelism (MLP) and improving data reuse. The first approach reduces the impact of long memory latency by issuing large number of parallel memory accesses. This effectively converts the latency-bounded programs to be bandwidth-bounded. While it works for GPUs and some high-end FPGA platforms with high off-chip memory bandwidth, the memory performance of mainstream FPGA platforms is often limited, making the approach less viable.

As for the second strategy to hide memory latency, improving data reuse for graph processing is a difficult task due to the irregular memory-access patterns. Regular computations have fixed memory-access flows that can be extracted directly from their algorithmic description, which provides the opportunity of applying static, offline optimizations. For example, in the case of dense matrix-vector multiplication, the embedded data dependency for generating the output vector is independent from the values of the input data. At design time, this dependency can be extracted, analyzed and optimized using classic optimizations such as data tiling [25] and pre-fetching. In the case of pointer-based graph computations, these types of static optimizations are not applicable because of dynamic data dependency. In graph traversal algorithms like Breadth First Search, the memory-access patterns are dictated by the edge connections of the input graphs. As this information is not available at design time, the accelerator developers can only rely on dynamic techniques, such as CPU-style hardware caches, to capture reuses. Unfortunately, the performance of standard cache designs is often compromised due to the gather-scatter accesses during the traversals of sparse graphs, as the inter-node connections can be very random in many real-world networks.

Algorithmic Optimization. Graph processing is a domain with large amounts of research effort. New optimizations on the algorithm side are constantly developed to further reduce algorithmic complexity and improve performance. However, while those optimizations are often rapidly deployed in software-based solutions, they are less often seen in FPGA accelerators, due to the high complexity of hardware development.

In comparison to software development—which features fast compilation flow, easy-to-use debugging tools and highly abstract language to mask low-level system details from the developers—designing and implementing hardware tends to be slower, more time-consuming to debug and often requires planning at the register-transfer level (RTL). This overhead severely limits the productivity and the speed to adapt newer, better algorithms. Unfortunately, as those newer algorithms often provide significant speedups over their predecessors, basing the hardware design on outdated algorithms often prevents the FPGA accelerator from delivering competitive performance. One such example is the scheduling optimizations for worklist-based graph algorithms.

While differing in function, many graph algorithms can be abstracted as a worklist and an iterative processing routine [17]. The worklist is a data structure that stores a set of pending tasks, often implemented as a queue; the processing routine iteratively dequeues and completes those tasks until the worklist is empty. With the freedom to define the task and the processing routine, this worklist-based model can be generally applied to represent different computations. As a graph computation progresses, new tasks are generated in an order determined by the topology of the input graph and the execution phases, which tend to be highly irregular. While this irregularity imposes a great challenge for extracting parallelism and exploiting data locality, it presents an opportunity for worklist scheduling, as the naturally occurring insertion order of tasks is often not the ideal processing order.

A classic example of worklist scheduling is the Dijkstra Single-Source-Shortest-Path (SSSP) algorithm, which, by prioritizing the tasks that are more likely to produce the final

node value assignments, significantly reduces the number of task insertions and overall processing time. In many cases, the improvement of processing time can be significant. However, despite the performance benefit, worklist scheduling is rarely considered for accelerator designs due to the implementation effort. Most FPGA-based works as of today are still based on the standard Bellman-Ford SSSP without scheduling.

1.3 Shared-Memory CPU-FPGA Systems

Traditionally, commercial FPGA products are designed around the offloading-based processing paradigm, in which the FPGA is loosely coupled with the host CPU over an I/O bus as a peripheral component. In this type of system, each of the two devices has its own separate memory; hence, when offloading a task, the input data have to be manually duplicated and transferred from the CPU to the FPGA. This incurs huge initialization overhead for data-intensive applications like graph analytics.

In recent years, systems with shared off-chip memory between the FPGA and the processor through a cache-coherent interconnect have emerged. Unifying the disjointed memory spaces found in offloading-based platforms eliminates costly data transfers for graph applications. In addition, the low-latency CPU-FPGA communication points to heterogonous, fine-graph collaboration between the two devices, allowing each of the CPU and the FPGA to focus on the work at which it excels. While this new opportunity presents great potential in tackling many problems in FPGA processing today, it remains a design point waiting to be proven.

1.4 Motivation and Goal

The dissertation aims to demonstrate that the share-memory CPU-FPGA system can be used as a competitive platform for graph processing, even without the high-bandwidth memory systems that are exclusive to high-end systems. We achieve this goal by showing that our solution can deliver performance comparable to prior work on platforms with superior memory bandwidth through platform and application specializations, while consuming only a small fraction of the power, thus providing overall better energy efficiency. The main motivations behind this work are summarized as follows:

1. Graph analytics have gained significant attention in important applications such as data mining and machine learning. Those applications often run on servers/data centers where the power consumption is a major concern. While energy efficiency is the traditional strength of FPGA-based accelerators, their performance is often lower compared to other platforms due to the reasons presented in section 1.2. By demonstrating that an FPGA-based solution can also be competitive in performance with our optimization techniques, this work can help to improve the use of FPGAs in the server market today.
2. CPU-FPGA integration through a shared memory is a new feature that needs to be proven for its commercial value. Processor-assisted scheduling, a new collaborative processing paradigm in this work, exploits this close integration for scheduling optimizations that are difficult to realize on a standalone FPGA. This provides a use case for this emerging platform.

1.5 Approach

In this thesis, we tackle the main challenges to FPGA-based graph processing—namely, the absence of memory-access optimization and slow adaption of new algorithms—to improve performance for worklist-based graph applications on a shared-memory CPU-FPGA platform. The proposed solution to those challenges consists of two parts. In the remaining portion of this section, we will present high-level overviews to each of them.

Part 1. High-Performance FPGA-Only Accelerator. First, to reduce the dependency on high-memory bandwidth, which is not available to most commercial FPGA systems, we developed a high-performance, stand-alone accelerator for worklist-based graph traversal that improves on-chip data reuse with a specialized caching system. This caching system exploits application-specific information by partitioning the storage for different data types to minimize interference and conflict misses. In combination with a novel graph preprocessing technique that improves locality by re-indexing vertices based on the topology of the graph, our solution is able to minimize off-chip bandwidth consumption. This allows our accelerator to deliver comparable performance to the state-of-the-art software solution on a processor with much higher memory bandwidth and cache capacity, while consuming only a small fraction of the energy.

Part 2. Processor-Assisted Scheduling on Shared-Memory CPU-FPGA System. Using the optimized accelerator from part 1 as the baseline, the second part of the work targets a widely used algorithmic optimization: worklist scheduling on shared-memory

FPGA-CPU platforms. The worklist-based processing model centers around a data structure for storing pending tasks called "worklist." Very often, the processing of those pending tasks can be reordered to improve performance without compromising the function correctness. Despite being highly effective, worklist scheduling is rarely used for the FPGA accelerator. To understand why, we conducted a study to implement a priority scheduler on the FPGA next to our accelerator. We discovered that, in addition to the implementation complexity, this approach consumes a large amount of on-chip BRAM for caching the priority-queue. Without this cache, the long-latency accesses to the priority queue stored in memory overshadow the benefit of scheduling and degrade the overall performance. As the baseline accelerator also needs BRAM for caching vertex data, this results in a serious conflict for the limited FPGA resources.

To tackle this problem, we propose Processor Assisted Scheduling (PAS) on a shared-memory FPGA platform. A shared-memory system offers low-latency communication and coherence accesses to a shared-memory space for the FPGA and the host CPU. PAS harnesses the power of a shared-memory system by using FPGA for the standard graph-processing routine and dynamically offloading the scheduling task to the host processor. The software scheduler exploits the sophisticated cache system and lower-memory latency of the processor to deliver fast scheduling service, which helps the accelerator to obtain the benefit of worklist scheduling without consuming precious BRAM resources.

1.6 Our Contributions

The work of this thesis uncovers the necessary techniques to break the dependency on memory bandwidth for graph processing on shared-memory FPGA systems. Specifically, the contributions are as follows:

1. We developed a cache system that is specialized for worklist-based graph processing, which separates the buffering for different data types. We compared this design with the standard unified cache architecture from CPUs and showed that our approach can effectively reduce data interference and cache miss rates.
2. To reduce write traffic, we designed and implemented a coalescing module that combines writes to the same memory addresses to increase the effective bandwidth utilization.
3. We developed a novel graph-partitioning-based vertex re-indexing technique that adaptively selects the appropriate data-layout schemes on the vertex degrees of the input graph for maximizing data locality. In comparison to the state-of-the art preprocessing optimization, this approach is more effective for long-diameter, low-degree datasets while delivering comparable performance for other graphs.
4. To improve the cache utilization, we developed a novel locality-aware worklist-scheduling optimization for worklist-based graph applications.
5. We evaluated the practicality of implementing worklist scheduling for graph processing on an FPGA using our optimized accelerator, and we discovered that, to provide a satisfying scheduling rate, it requires an impractical amount of on-chip storage.
6. We developed PAS for shared-memory CPU-FPGA systems, which allows the CPU

to conduct the task of scheduling for the accelerator. We implemented and evaluated PAS on an Intel HARP. The results suggest that this approach is capable of retaining the majority of the theoretical benefit of worklist scheduling for BFS and SSSP, while the software scheduler consumes only a negligible load on the processor. In addition, PAS serves as a new use case for the shared-memory CPU-FPGA platform.

CHAPTER 2. OUTLINE

The remainder of this thesis is organized as follows. In Chapter 3, we present an overview of the benefits and the general challenges of accelerating graph algorithms on FPGA by surveying and analyzing prior publications.

The first part of the work, our locality-optimized graph traversal elastic pipeline, is covered in Chapters 4 to 6. In Chapter 4, we first present the architecture of a baseline BFS accelerator that represents a typical design focused on maximizing MLP. We then characterize this accelerator using a set of standard datasets and identify the major performance limitations. Chapter 5 presents our proposed solutions for the bottlenecks. Specifically, we focus on a hardware pipeline optimization to reduce stall and improve throughput, a BRAM-based cache architecture to exploit data reuse specifically for graph traversal applications, a write coalescing module to reduce DRAM write traffic, and a data preprocessing optimization that improves locality adaptively for graphs with different degrees and diameters. Chapter 6 presents our evaluation for the optimization and a

comparison with other works.

The second part of this thesis, covered in Chapters 7–9, is our work to introduce worklist scheduling to our graph traversal accelerator on a shared-memory CPU-FPGA system. Chapter 7 explains the basics of worklist scheduling and case studies of two examples, including the well-known Dijkstra optimization for SSSP and a new locality-aware scheduling we proposed for BFS to improve cache performance. Chapter 8 presents our limit studies to reveal the potential of worklist scheduling with zero-overhead schedulers and reveals the issue of implementing worklist scheduling on an FPGA through a simulation study. To tackle this problem, we developed PAS for shared-memory platforms. In Chapter 9, we present and evaluate PAS on an Intel HARP. Specifically, we discover that it is capable of retaining most of the theoretical performance benefit with minimal power overhead and interference with the host applications on the CPU.

Finally, Chapter 10 presents concluding remarks and future directions.

CHAPTER 3. THE INCENTIVE AND CHALLENGES OF FPGA-BASED GRAPH PROCESSING

In this chapter, we will discuss the background of FPGA processing. Specifically, we will first cover the basic architecture of a typical FPGA, how it provides reconfigurability, and its benefits and drawbacks compared to general-purpose processors. In the second half of this section, we will elaborate on graph processing and the main challenges of mapping it to an FPGA through an application case study.

3.1 What is an FPGA?

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based around arrays of Lookup Tables (LUTs). In most FPGA architectures, LUTs, registers and

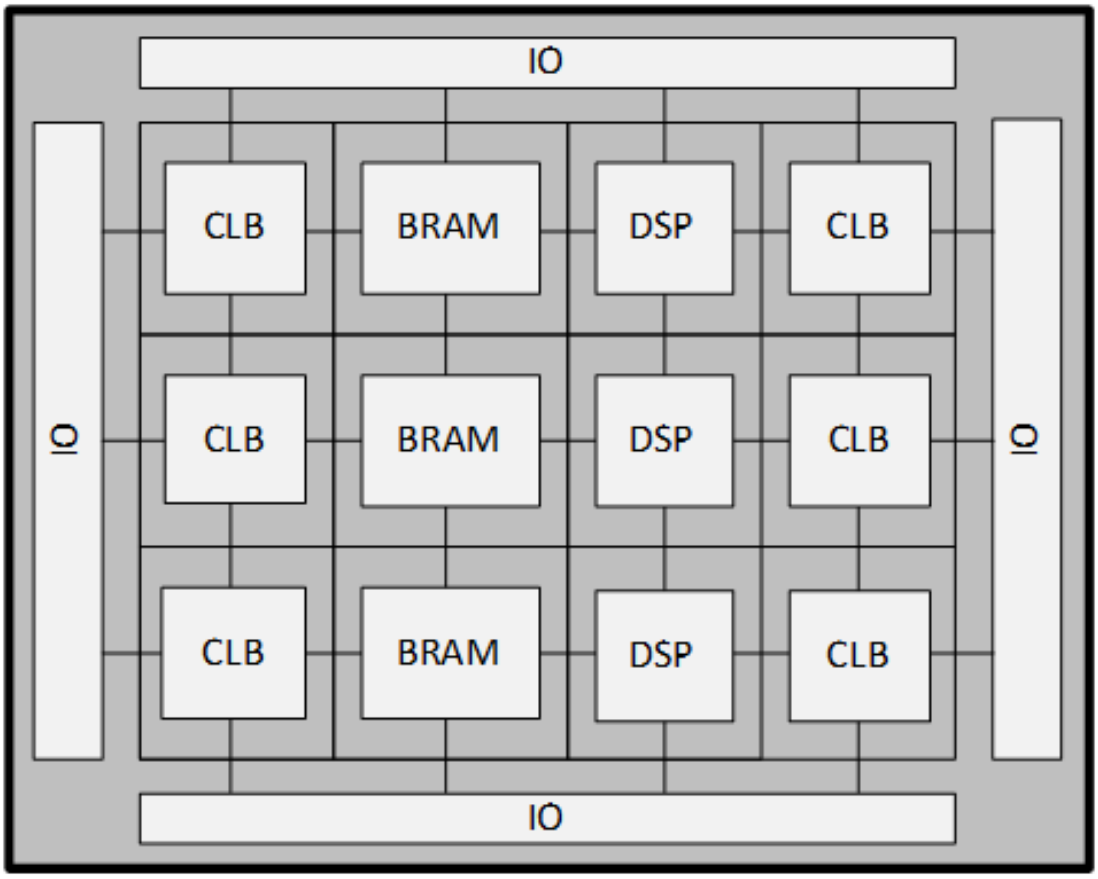


Figure 1. Block diagram of an FPGA chip.

multiplexers are organized to form the basic building blocks of logics—namely, Configurable Logic Blocks (CLBs) [26] and Arithmetic Logic Modules (ALMs) [27] in Xilinx and Alter/Intel terminologies, respectively. These basic blocks are integrated with other components on an FPGA, including Block RAM (BRAMs) and hardened functional blocks, such as arithmetic and DSP modules, via programmable interconnects. Together, these hardware resources can be reconfigured to implement different functions after manufacturing by altering the routing of interconnects and the functional mappings of LUTs. Figure 1

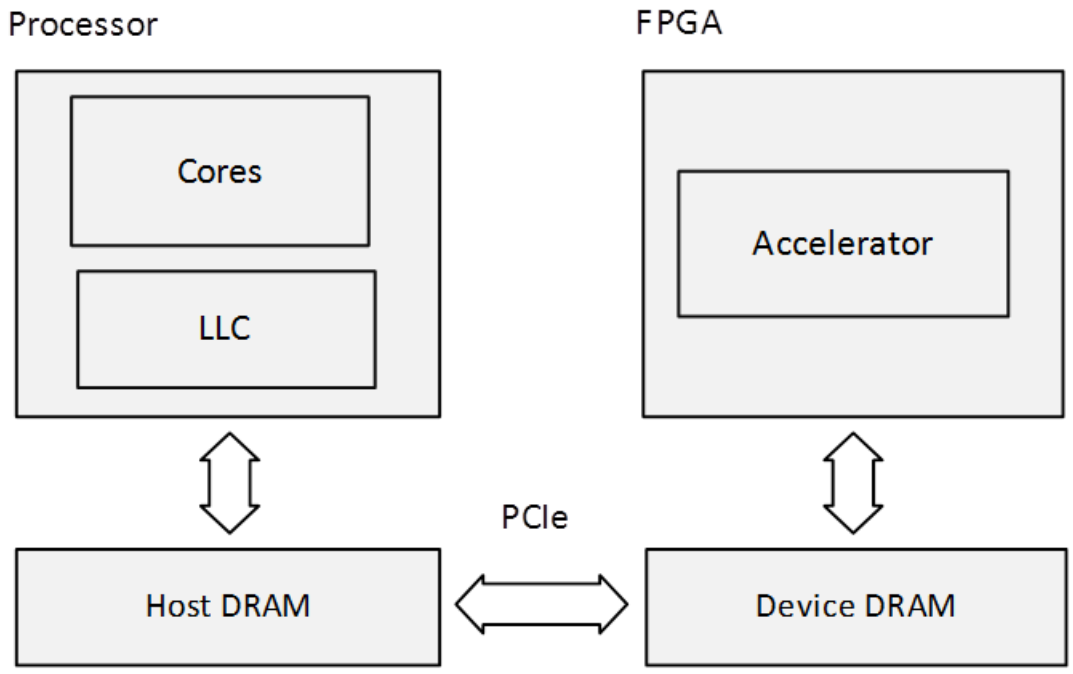


Figure 2. Offloading-based processor-FPGA platform.

shows the block diagram of an example FPGA with its basic components.

The feature of post-manufacturing programmability distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are hardened for specific application logics. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM-based, which can be reconfigured as the design evolves. On the other side of the hardware spectrum, general-purpose processors rely on architecture support for a set of instructions to deliver software-based programmability. In contrast, FPGAs' programmability is achieved at the hardware level through the LUT and interconnect reconfiguration. As transistors are directly used for the computations associated with the user application, this delivers high utilization of both silicon area and energy, resulting in better

performance and higher efficiency.

3.2 Shared-Memory FPGA

Traditionally, commercial FPGA platforms are designed around an offloading-based processing model. In this model, the host processor, which dispatches tasks to be accelerated, and the FPGA accelerator, which processes the dispatched tasks, are loosely connected via an I/O bus (e.g., PCIe, which is often severely limited in bandwidth and latency). In this type of system, the FPGA and processor have separated memory spaces; hence, the input data and the results of the offloaded tasks need to be explicitly copied between the two devices, incurring a large overhead. While it works for lengthy tasks that can amortize this overhead over its long processing time, it is not suitable for applications that have short tasks and/or require frequent interactions of the two devices. The block diagram of an offloading-based CPU-FPGA system is shown in Figure 2.

As early as 2010, FPGA vendors have taken a "System-on-Chip" approach to integrating processor cores and reconfigurable fabric with cache-coherent shared memory within the same chip (e.g., Xilinx Zynq [3] and Altera Arria [4]). The choices of processor cores, fabric capacity, and DRAM interfaces in these devices were optimized for embedded applications. On the higher end, some of the earlier attempts at using cache-coherent FPGA acceleration for HPC include DRC and Xtremedata [9], and the Convey HC-1 [10]. More recently, Intel and IBM have respectively announced server-class products that integrated FPGAs and processors at the board level using proprietary cache coherent interconnects (e.g., IBM CAPI

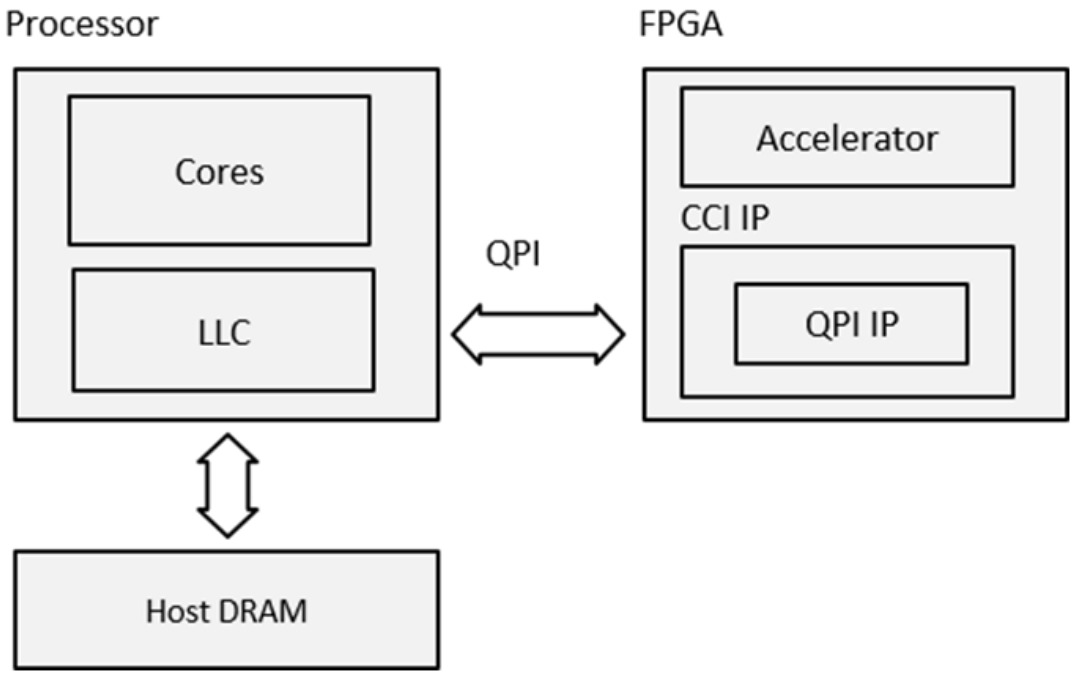


Figure 3. Intel HARP, an example of a shared-memory CPU-FPGA platform.

[5] and Intel QuickPath [6]), enabling low-latency communication and coherent accesses to a shared off-chip memory. The Intel Heterogeneous Architecture Research Platform (HARP) [6] is a pre-production prototype in this category. Intel HARP pairs a server-class multi-core Xeon processor with an Altera Stratix V FPGA using Intel's QPI interconnect, which maintains the coherence for CPU and FPGA memory accesses at the shared last-level cache (LLC).

A block diagram of the Intel HARP is shown in Figure 3. We believe this close integration offers a new opportunity of fine-grain, heterogeneous collaboration. With the low-latency inter-socket communication and memory coherency for eliminating inter-device data copying, the accelerator and the processor can dynamically, synergistically work

together to process different parts of the same task, providing the energy efficiency and performance that neither can achieve alone. While prior work has attempted this idea on shared-memory systems [20] [29], their task partitioning are mostly at a coarse application-level granularity and have shown limited success. This fine-grain FPGA-CPU co-processing paradigm remains a design point to be proven.

3.3 The Challenge of Irregularity

The behavior of an irregular application is governed by the large numbers of pointers in the input data. In terms of memory operations, the data-access patterns cannot be determined at design/compile stage without the knowledge of the input data. This prevent the hardware developers from applying static optimization, such as data tiling or mapping the application to a dataflow architecture on FPGAs. Due to the large number of pointer-incurred gather-scatter operations, the performance of off-chip memory accesses of an irregular application is usually worse compared to its regular counterparts.

In terms of computation, irregular applications usually have much lower compute intensity relative to their regular counterparts, as the large number of pointer retrievals increase the number of memory accesses and lower the compute-to-memory ratio. This makes performance optimization for FPGA-based accelerator a challenge, as there is a relatively small amount of computations to exploit the compute resources on FPGAs. In addition, the number of parallel computations in irregular applications is governed by the control flow and can change with different input data and execution phases. As the degree of

```
1:   FIFO_worklist.enqueue(source);
2:   for(int i=0;i<num_vtx;++i)vtx_array[i].dist=INF;
3:   while !FIFO_worklist.empty() do
4:      vtx curr_task=FIFO_worklist.dequeue();
5:      int task_idx=curr_task.idx;
6:      int task_dist=curr_task.dist;
7:      int curr_edge_offset=vtx_array[task_idx].edge_offset;
8:      int num_edge=vtx_array[task_idx].num_edge;
9:      for num_edge to 0 do
10:        int curr_nbr=edge_array[curr_edge_offset].dst;
11:        int edge_length=edge_array[curr_edge_offset].length;
12:        int curr_dist=edge_length+task_dist;
13:        vtx nbr_vtx=vtx_array[curr_nbr];
14:        if vtx_array[nbr_vtx].dist>curr_dist then
15:           nbr_vtx.dist=curr_dist;
16:           vtx_array[nbr_vtx.idx]=nbr_vtx;
17:           FIFO_worklist.enqueue(nbr_vtx);
18:        end if
19:     end for
20:  end while
```

Algorithm 1. Baseline BFS Algorithm.

parallelism dynamically varies, the allocation of compute resource and task scheduling for load-balancing can be difficult.

Graph Processing. Among irregular computations, graph analytics have gained much attention recently. Graph analytics are tools for determining the relationships among connected components in a graph. The wide variety of applications, including item recommendations in e-commerce [15], social network analysis for ad placement [21], computer-vision [18], and so forth, mark their importance and result in an ever-growing demand for better performance and energy efficiency. In this work, we focus on an FPGA-based solution for graph computations. In the next subsection, we will examine and compare the state-of-the-art implementation of a graph algorithm on different platforms, which would

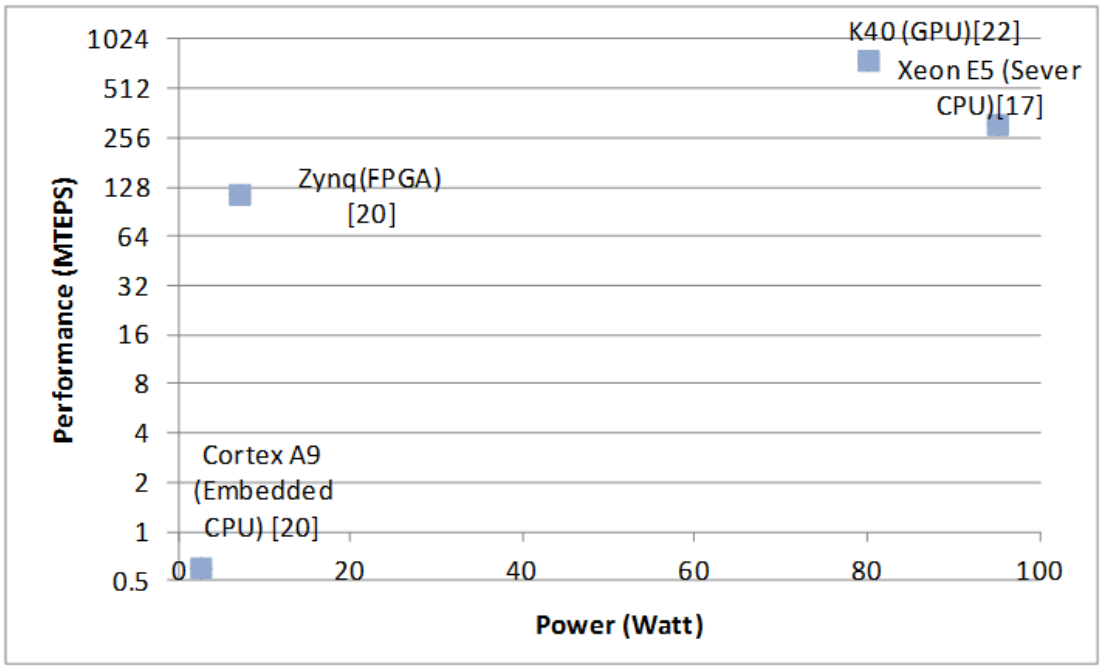


Figure 4. Forward BFS performance on a different platform.

help us understand the incentive behind FPGA-based graph processing.

3.4 Survey of Prior Works on Breadth First Search

To understand how an FPGA-based solution for graph processing compares to other platforms, in this section, we case-study the performance of an important graph traversal algorithm, Breadth First Search (BFS), on CPUs, GPUs and FPGAs. BFS is an iterative algorithm for traversing a graph. It determines the shortest paths from a source vertex to the other vertices in a non-weighted graph by iteratively expanding the frontier of the search. It serves as an important building block for many graph analytics and has been the focus of researches in the high-performance computing community. The pseudo-code of a worklist-

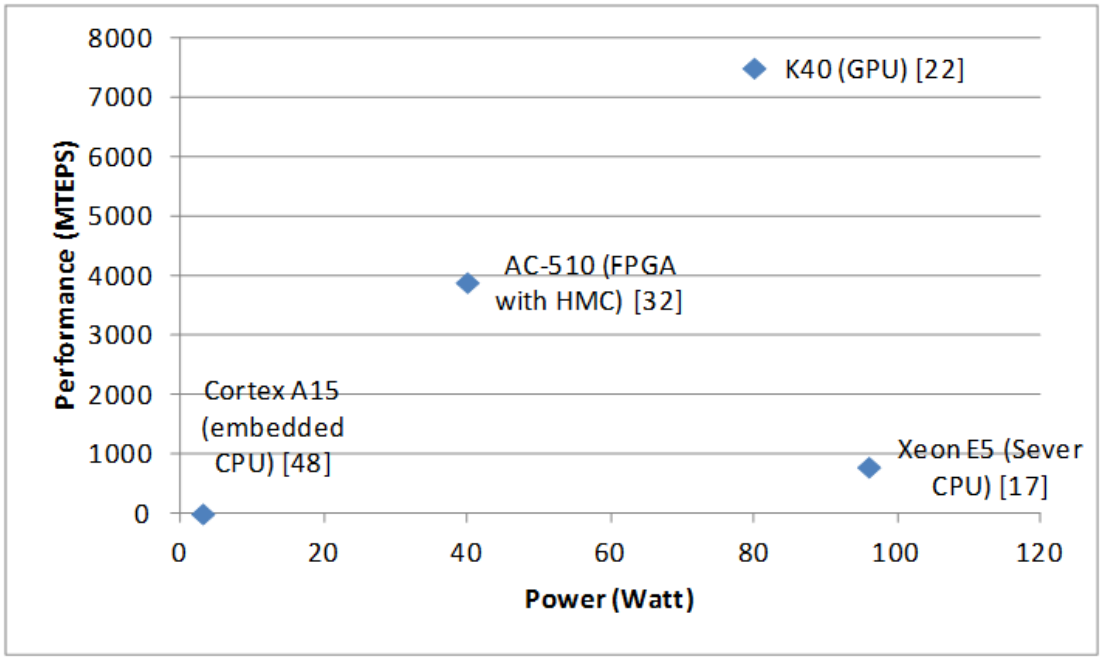


Figure 5. Energy efficiency of forward BFS on different platforms.

based BFS [17] supporting graphs in Compressed-Sparse Row (CSR) format [30] is shown in Algorithm 1, which is known is the forward BFS. In this implementation, the label to each vertex stores the index of its predecessor in its shortest path from the source. Before BFS starts, all vertices are initialized to a value representing "unknown," (in this case -1), as the shortest paths have not been discovered yet. When BFS completes, the shortest path to each vertex can be determined by simple backtracking.

The main routine of BFS iteratively processes the "tasks" stored in a queue-based data structure called worklist (line 4 in Algorithm 1), which represents the frontier of exploration, containing destination vertices of the newly discovered shortest paths. In each iteration, BFS processes a task vertex by attempting to "relax" its neighbors that have not been visited,

which is the process of overwriting a neighbor's label with the index of its connected task vertex. As the vertices that are relaxed in an earlier iteration certainly contain shorter paths from the source compared to the vertices that are relaxed later, this process of relaxation represents a greedy approach to extend known short paths. Relaxed vertices become the new tasks and are inserted into the worklist to represent the expanded frontier.

BFS is an effective example for demonstrating the challenges highlighted in section 4: low compute intensity, irregular, locality-unfriendly gather-scatter accesses during neighbor visits, data/phase dependent parallelism in the outer loop, and control flow-governed relaxations. Due to these constraints, application developers traditionally prefer GPUs, which offer higher DRAM bandwidth, and server-grade CPUs, which provide a large cache for memory-latency hiding.

Forward BFS on Different Platforms. Figure 4 shows the performance comparison of published state-of-the-art forward BFS implementations on different platforms. The overall trend suggests that the performance scales with the overall bandwidth of the system, as the GPU-based work [22] delivers the highest performance, followed by a server CPU. On the FPGA side [20], its performance is severely limited by the low off-chip memory bandwidth on the platform, which is a common constraint for many off-shelf FPGA boards, as they rely on LUT-based memory controllers running at low clock frequencies. As for the embedded implementation that uses both the mobile GPU and CPU [43], the performance is lower than the FPGA work, despite having a slight edge in memory bandwidth. When the power consumption is factored in, the benefit of the FPGA's lower clock frequency and application-

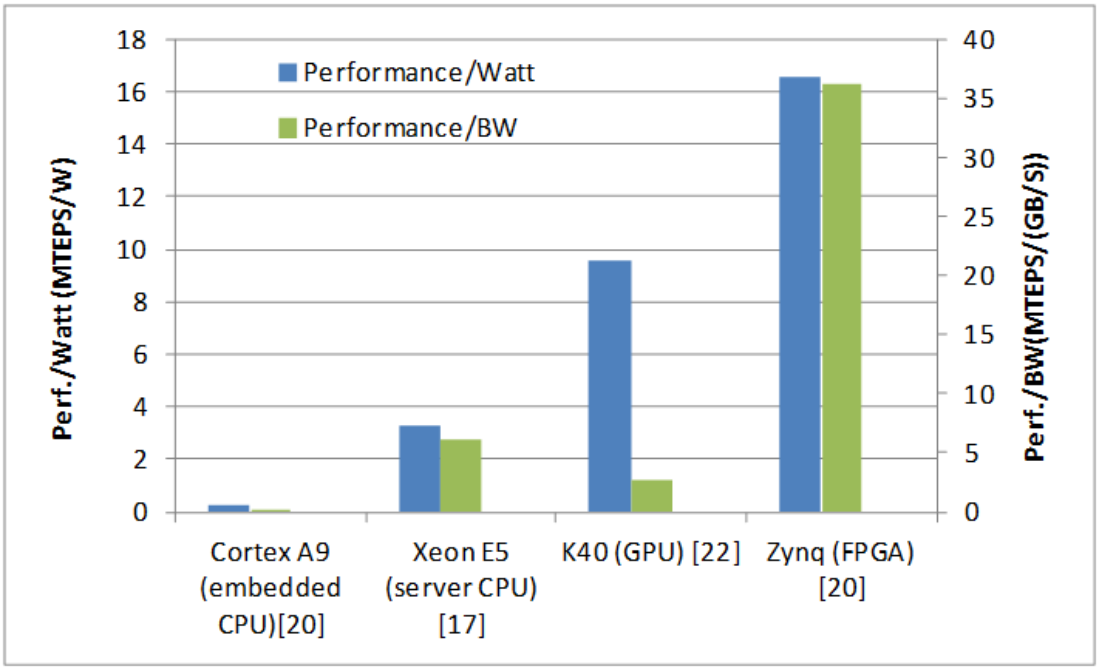


Figure 6. Performance of direction-optimized BFS on different platforms.

specific hardware design become apparent. Figure 5 show the energy-efficiency measurements in performance-per-watt for the baseline BFS algorithm on different platforms. The baseline FPGA work is second to only the embedded FPGA solution, which outperforms all competitors.

How FPGA-Based Graph Processing Can Benefit from Traditionally Overlooked Algorithmic Optimizations. After examining forward BFS, we will quickly look at an algorithmic variant called Direction-Optimized BFS (DOB) [31]. DOB is a new BFS algorithm aiming to solve the inefficiency in finding unvisited vertices in the later stage of forward BFS. DOB has two phases. The first phase of DOB is the same as that of the forward BFS, expanding frontier and performing relaxation based on the stored task vertices.

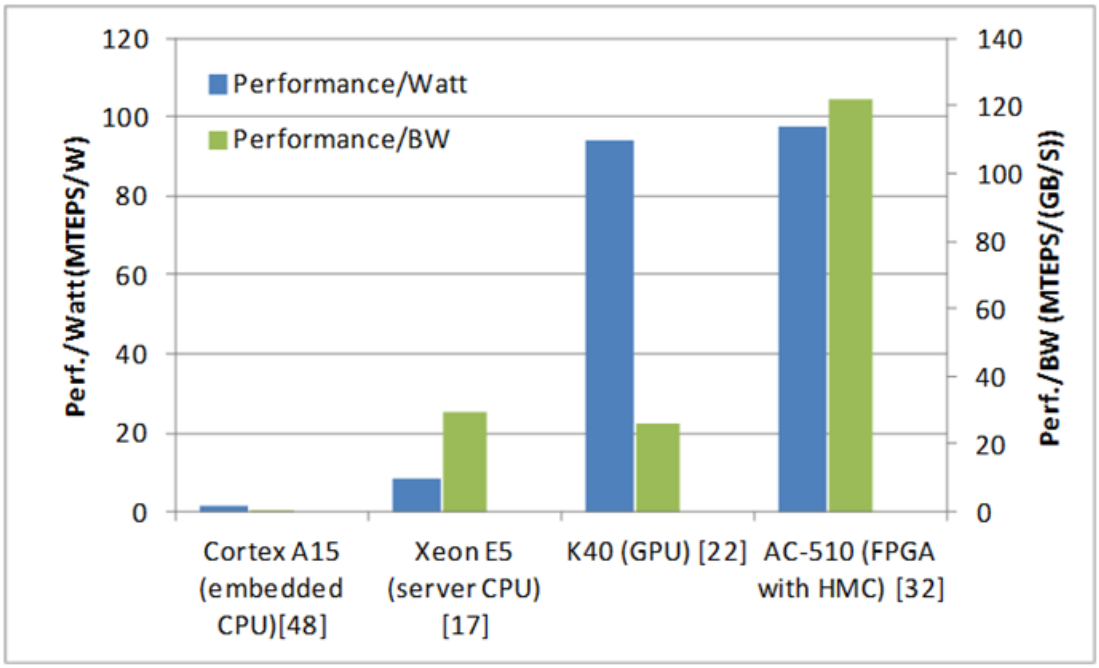


Figure 7. Energy efficiency of direction-optimized BFS on different platforms.

However, in the second phase, DOB performs relaxation backward by looking at each yet-visited vertex and verifying whether it is connected with another vertex that has been relaxed previously. DOB is a more efficient approach, as its developer found that the inner loop of forward BFS is ineffective in finding an relaxable neighbor in the second phase when most of the vertices in the input graph are already relaxed, which incurs large numbers of wasted vertex reads.

While DOB has been widely used in software-based BFS, it has not been applied to a FPGA accelerator until very recently [32]. The comparison of DOB on different platforms is shown in Figures 6 and 7, for performance and performance-per-watt, respectively. In both cases, the FPGA work dominates the results. In the DOB comparisons, the FPGA work is

built on a platform with the Hybrid Memory Cube technology, which provides a significantly higher memory bandwidth compared to most traditional FPGA systems. In combination with other platform advantages of application-specific processing on FPGAs, it helps the accelerator to compete with the GPU-based work [22] with twice the memory bandwidth and consumes twice the power.

3.5 Discussion

Graph Processing Works Well on FPGAs. From our survey of prior works for BFS, we found the performance of a BFS accelerator is highly sensitive to the memory bandwidth of the selected platform. The survey of DOB works showed that, when given a memory system with competitive bandwidth, the FPGA accelerator is capable of outperforming the state-of-the-art GPU solution. In terms of energy efficiency, FPGAs generally perform exceptionally well. For both forward BFS and DOB, FPGA-based works demonstrated excellent results in performance-per-watt.

Missed Opportunities for FPGA-Based Accelerators. While many works concentrate on the maximization of off-chip memory bandwidth, this is arguably not an ideal strategy for performance, as memory bandwidth is typically not the strong suit of most FPGA systems. While there exist a few high-end, extremely expensive platforms, such as Convey HC-2 and Xillinx Ultrascale+ with HBM that offers memory bandwidth comparable to GPUs, for main-stream FPGA products, the bandwidth is often fairly limited. We believe there are two opportunities that are commonly overlooked, which will allow them to unleash the

performance of FPGAs, even with a constrained memory sub-system. Those missed opportunities are as follows:

1. **Underutilization of On-Chip Storage.** One of the FPGA's greatest strength is its configurable on-chip storage components, BRAMs, which are often used for buffering fetched data for on-chip reuse in regular computations to reduce off-chip memory bandwidth consumption. However, in graph processing, due to the irregular data dependency, it is difficult to develop an effective buffering strategy through static analysis; hence, in many FPGA-based works, data buffering via BRAM is disregarded. We believe it is important opportunity for FPGA-based graph-processing solutions, as prior works on characterizing graph algorithms have shown that locality does exist in most real-world graphs [35]. By specializing the buffering scheme using application-specific knowledge, such as avoiding buffering data that are known not to be reused, to exploit the user-configurable on-chip buffers on FPGAs, there is a lot of potential for reducing the dependency on memory bandwidth.
2. **Slow Adaption of Algorithmic Optimization/Hardware Specialization.** During our literature survey, we found that there are a lot of algorithmic optimizations, such as input data pre-processing and worklist scheduling, that can significantly improve the performance of the application. While the applicability of this type of application-specific optimization is not limited to software-based solutions, its FPGA adaptation is often sluggish. The late adoption of DOB to FPGA accelerators is an example of this problem. While DOB has been the standard for the Graph500 software reference since 2011, the first FPGA-based DOB was not published until

2018. Needless to say, many other algorithmic optimizations never even have their chance to land on FPGAs.

In summary, we found that FPGA is a good platform for graph processing for energy efficiency. If performance is the main concern, current work requires a specialized high-bandwidth memory subsystem to fully exploit the available MLP. However, this feature is exclusive to certain high-end platforms and cannot be found in mainstream commercial systems.

To tackle this issue, we identify two observations that help to construct a competitive FPGA accelerator without requiring high off-chip memory bandwidth, which leads to our thesis work. Specifically, the first observation leads to the first part of this thesis, which optimizes data reuse for an FPGA graph traversal accelerator. The second observation leads to the second part of the thesis, which exploits CPU-FPGA collaboration on a shared-memory platform to realize scheduling optimizations that are traditionally infeasible to implement on FPGAs.

Part I. Optimizing a BFS Accelerator for a Low-Bandwidth FPGA Platform

CHAPTER 4. A TRADITIONAL ACCELERATOR FOR GRAPH TRAVERSAL: OUR BASELINE

Mapping the BFS in Algorithm 1 to hardware is not a straightforward process, as many design decisions are involved. Figure 8 presents a simple hardware implementation for BFS, which directly mirrors a sequential routine of Algorithm 1. It requires minimal design effort for development, as the accelerator consists of a single-state machine. However, this approach suffers poor performance due to system underutilization caused by the lack of memory latency hiding. In this state machine, the memory operations are always serialized, and the accelerator stalls until the ongoing read or write completes. For instance, when reading the outgoing edges from a task vertex, despite the fact that those edges are data-independent, this design is not capable of issuing parallel reads and causes frequent stalls. The length of the stall period is determined by the round-trip latency to the off-chip DRAM,

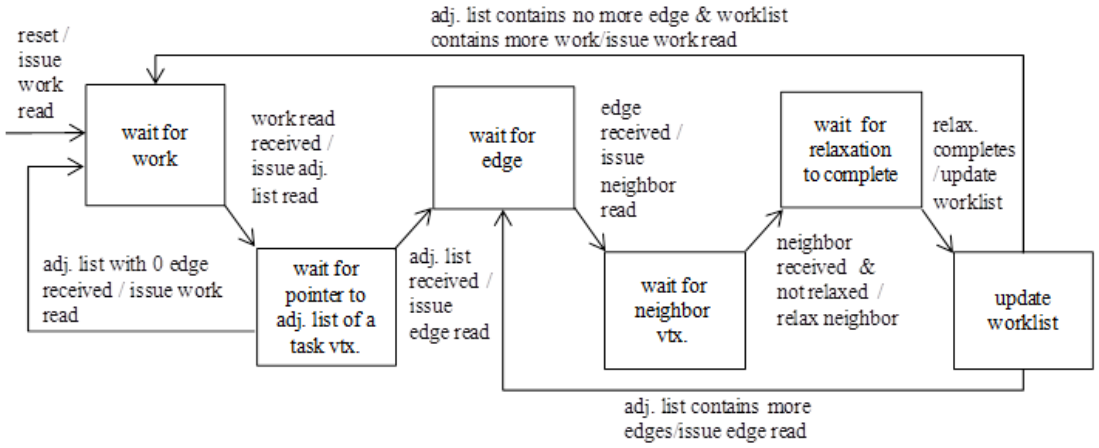


Figure 8. Sequential FSM for BFS.

which typically spans hundreds of clock cycles [36] [37] and thus severely limits the performance of the system.

As such, it is important to avoid the hardware stall by applying techniques for memory latency hiding. In general, there are two ways to mitigate the penalty for long memory latencies: 1) latency reduction, and 2) exploiting MLP. Reducing the latency of data accesses is a long-studied topic in the domain of computing. Techniques in this category include capturing on-chip data reuse through cache or scratchpad memory and introducing data prefetch. However, as discussed in Chapter 2, the irregular nature of graph processing prevents textbook-style standard static buffering schemes to work effectively. Hence, most developers primarily focus on maximizing MLP for their graph algorithm accelerators [11] [20] [32].

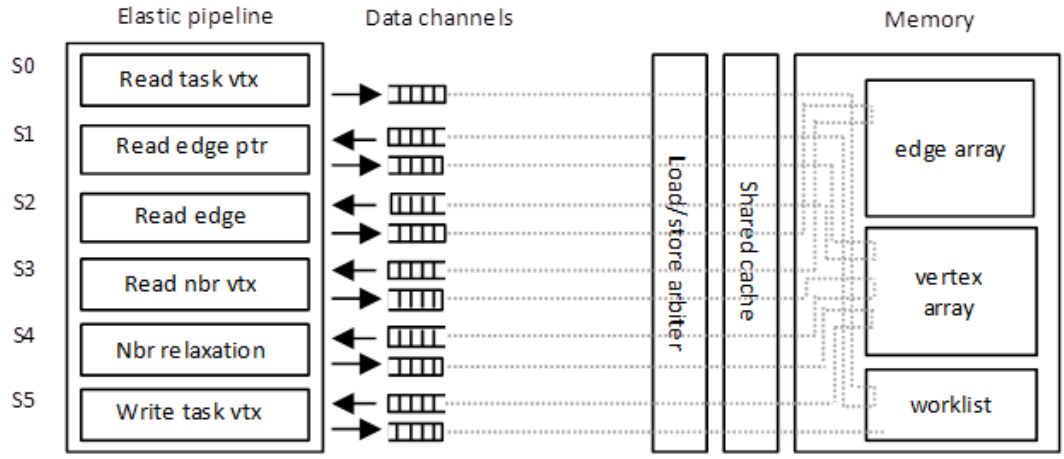


Figure 9. Our baseline BFS accelerator pipeline.

In terms of improving MLP, software solutions often exploit the independent memory operations at the task level by parallelizing across the outermost loop in the algorithm [17] [19]. In software this is typically done through multi-threading. While the same strategy can be applied to the presented hardware design by replicating the state machine for parallel-task processing, this approach is not always effective as it does not exploit the parallelism among relaxation of different neighbor nodes. In addition, this approach also needs a mechanism for guaranteeing the atomicity of accesses from different state machines, which tends to be complex and hard to scale.

To avoid the abovementioned issues, our accelerator extracts MLP through a different approach. Instead of focusing only the task-level parallelism at the outer loop, our

design uses an elastic pipeline to parallelize memory operations at different pipeline stages. The system diagram of our design is shown in Figure 9. In this figure, each of the six pipeline stages are responsible for issuing memory requests for the different types of data in BFS. Each stage continuously issues memory requests based on the responses of the memory transactions from the previous stage. In contrast to the sequential-state machine, which has only one state active at any point in time, all pipeline stages in this operate simultaneously. In between pipeline stages, we incorporate channel FIFOs for buffering memory requests and read results to reduce back pressure from later stages. This effectively hides back pressure from later stages and increases the likelihood that there are always pending memory requests waiting to be issued, ensuring high MLP. In a system like the Intel QPI-FPGA, many tens of memory requests could be outstanding; this is necessary to maximally hide memory latency and to saturate memory bandwidth. In a steady state, the full length of the pipeline would contain several BFS outer and inner loop iterations in flight.

To understand the detail of how this design works, let's examine how different pipeline stages map back to Algorithm 1. A single iteration of the BFS outer loop begins with stage S0 in Figure 9, which calculates the pointer to fetch the next node n from the circular buffer *Worklist[]*; the result of the fetch—the index of node n—is received by S1 to calculate a pointer to fetch *Offset[n]*. Both Offset[] and *Distance[]* hold one entry per node. In this solution, the data Offset[] and Distance[] are merged into a single array of per-node structs *{Offset,Distance}[]* such that S1 can fetch both *Offset[n]* and *Distance[n]* in one cache block load.

The rest of the steps, S2–S5, correspond to the BFS inner loop that performs relaxations. The inner-loop steps need to be repeated once for each neighbor of node n. Using Offset[], S2 calculates the pointer into *DestIdx[]* and *Weight[]*; here, again, we combined these two per-edge arrays as a single array of structs *{DestIdx,Weight}[]* for better spatial locality. S3 next uses the neighbor's index to fetch the neighbor's labels from Distance[]. After all this, S4 is finally ready to decide if the neighbor's distance needs to be updated. On a Intel HARP, because the AFU can only read and write memory in 64-byte cache blocks, S4 has to make the update by modifying the affected word in the cacheline fetched in S3 and writing the whole cache block back. Finally, the updated neighbor is appended to the worklist. To make efficient use of the cacheblock granularity writes, S5 usually accumulates several updates before writing back the cache block. The only exception to this policy is when Worklist[] in memory runs empty and is in a urgent need of a new work. In this case S5 must prematurely write a partially populated cacheblock.

This high degree of pipeline concurrency is important for performance and energy efficiency, but it does not come without complications. Most notably, S3 reads a neighbor's predecessor to decide if it should be relaxed, and only if so, S4 writes back the new predecessor. There can be many tens of cycles between the read in S3 and when the write finally takes effect. This creates a read-after-write hazard window; during this period, if the old predecessor label is read instead of the newly improved one, which has not propagated to memory yet, by another node's relaxation step, this could lead to an erroneous update.

name	vertices	edges	type	description
Rand_cluster	512K	8M	Directed, unweighted	Synthetic graph with 16 communities. Each edge has a 25% probability of connecting to a random vertex from a different community and a 75% probability of connecting internally.
RMAT_sparse	512K	1M	Directed, unweighted	Sparse RMAT graph [] with default distribution parameters.
RMAT_dense	512K	8M	Directed, unweighted	Dense RMAT graph [] with default distribution parameters.
circuit_4	80k	307k	Directed, unweighted	Circuit simulation graph from UF Sparse Matrix Collection [].
us_roads	129k	165k	Directed, unweighted	Road network in USA from [].
cond-mat	40k	351k	Undirected, weighted	Collaboration network on the condensed matter archive [].
rgg_n_2_19	524k	6M	Undirected, unweighted	Synthetic RGG graph from [].
road_FLA	1m	2.7M	Directed, weighted	Road network in FLA from []
Linux_call	324k	1.2M	Directed, weighted	Call graph of the linux kernel from []
webbase-1M	1M	3.1M	Directed, weighted	Web connectivity matrix from a Nvidia technical report []
Patent_main	240K	560K	Directed, weighted	Citation network for NBER patents []

Figure 10. Datasets used in evaluations.

A naive solution to this problem is indiscriminately blocking all processing of new nodes in S3 until the previous node has cleared through S5. While this guarantees functional correctness, it approach would destroy pipeline parallelism. Hence, in our solution, we employ a scoreboard-like bookeeping structure to keep track of the outstanding neighbor nodes read in S3. The scoreboard would block only hazardous read attempts until the

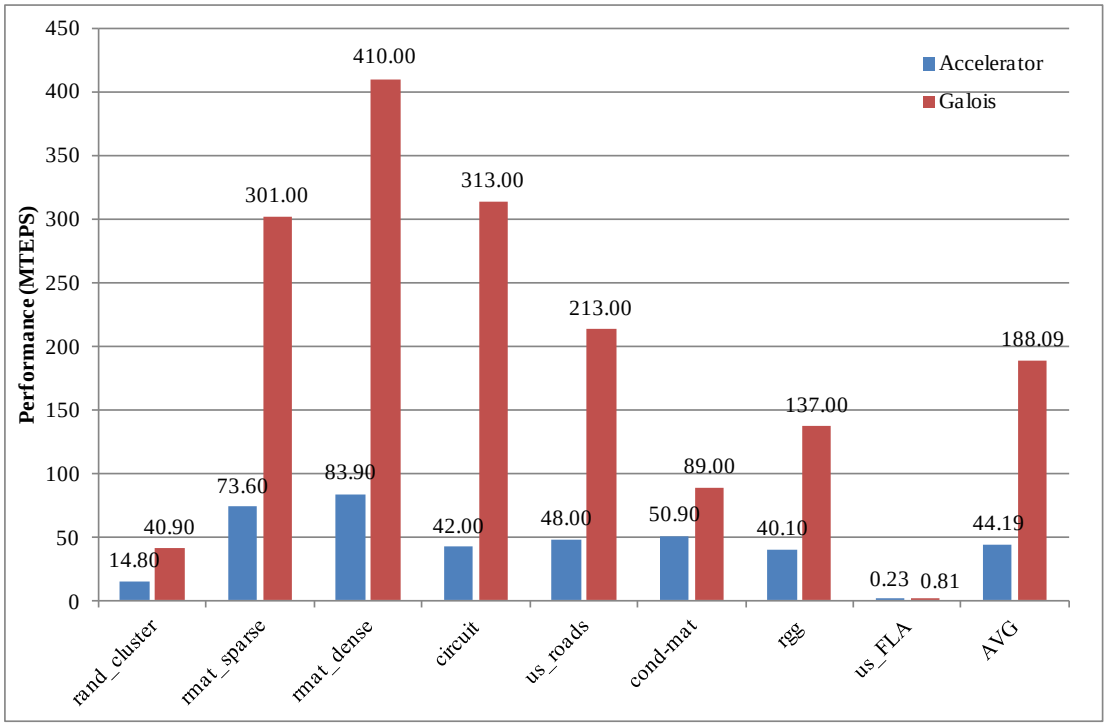


Figure 11. Performance of the baseline accelerator.

conflicting updates is dismissed in S4 (no update needed) or committed in S5 (after write completion).

The above represents a reasonable, albeit still basic, baseline accelerator of BFS. It reflects a traditional strategy to implement the BFS algorithm on an FPGA platform with the assumption that the input graph and the worklist is too large to fit on-chip and will have to be stored in the off-chip DRAM. In fact, the emphasis on maximizing concurrent outstanding memory requests closely resembles that of many recent works [20] [11].

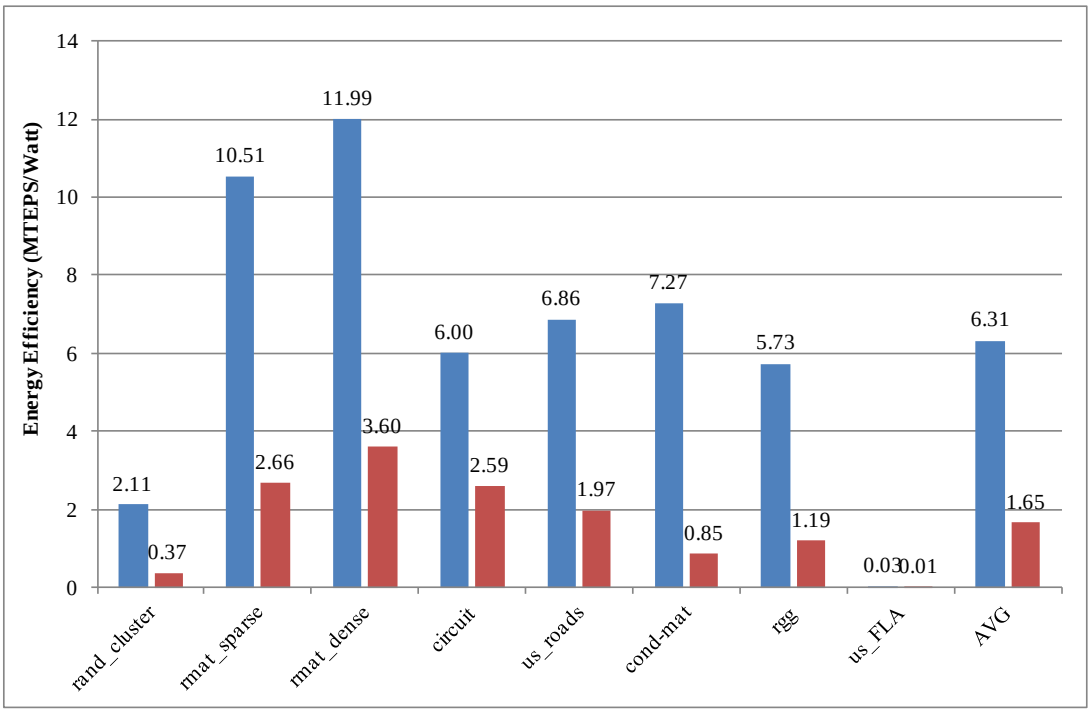


Figure 12. Energy efficiency of the baseline accelerator.

4.1 Performance Analysis

In this section, we perform an in-depth analysis to understand the performance and characteristics of the previously proposed baseline BFS accelerator. We will first identify its strengths and weaknesses, exposed in our experiments. Then, we propose new optimizations to tackle the discovered bottlenecks.

Methodology. We evaluated our baseline accelerator using an Intel HARP (shown in Figure 3). It features an Altera Stratix V FPGA and a host Intel Xeon E5-2680 V2 with 10 cores and hyper-threading integrated on a two-socket motherboard. The two devices are

provided with coherence access to a single memory channel. The Cache Coherent Interface (part of the QPI) maintains the data coherency between the last level cache (LLC) in the processor and a 64KB FPGA cache. A system protocol layer (SPL) is introduced to provide address translation and request reordering to user-defined FPGA designs with virtual addressing. A page table of 1,024 entries, each associated with a 2MB page (2GB in total), is implemented in SPL, which is loaded by the kernel driver. For this experiment, we are using only the FPGA to evaluate the presented elastic pipelines. The CPU is not used for the BFS operations. The graphs used in our experiments are summarized in Figure 10.

Results. The throughput performance of the accelerator is shown in Figure 11. On average, it delivers 44 million **traversed edges per second (MTEPS)** across our selected datasets. Its performance is roughly equal to the FPGA-only accelerator implemented on a Xlinx Zinq-7000 [20] with its memory bandwidth performance (4GB/sec) similar to HARP (6GB/sec).

In comparison to the BFS implementation in 20-thread Galois [17], a state-of-the-art software processing framework, running on the 10-core Xeon E5-2680 V2, our performance is, on average, 75% slower. This is not a surprise, as the hardware design relies solely on DRAM transaction throughput to gain performance, which is limited by the QPI interconnect and roughly 85% lower than that of the Xeon. However, in terms of efficiency (performance-per-watt), the baseline accelerator is, on average, 3.8X better than Galois which relies on multi-core processing and burns, on average, at 110 watt. The result of efficiency measurement is presented in Figure 12.

4.2 Bottleneck Analysis

In the previous section, we demonstrated that our baseline accelerator delivers competitive performance comparable to prior work on a system with similar DRAM bandwidth. However, it is still significantly slower than an optimized software implementation on a server-grade processor. To answer the main question of this thesis: if FPGA-based graph processing can compete favorably with the other platforms without relying on memory subsystems that are rarely found on standard commercial platforms, it is important to make sure that our design fully exploits the potential of the FPGA platform, and if not, to identify the bottlenecks. In this section, we will perform a detailed characterization study to answer these questions.

Methodology. Analyzing the system utilization and performance bottlenecks requires monitoring of the activities on difference components in our baseline accelerator. There is a limited scope of visibility to the operations on a physical hardware implementation without adding intrusive hardware performance scope or event loggers, which can introduce unintended effects on system timing and extra memory traffic. To avoid those issues, we use an in-house accelerator simulator, TLMsim, to pinpoint the bottlenecks. TLMsim is a C++-based in-house, cycle-accurate simulator for our elastic pipeline. With the channel buffer size correctly configured, it can accurately model different pipeline activities, while allowing the users to freely track signals and data on the simulated hardware. It models events in off-chip memory accesses, such as row-buffer hits/misses and memory interface stalls due to full Miss Status Handling Registers (MSHR).

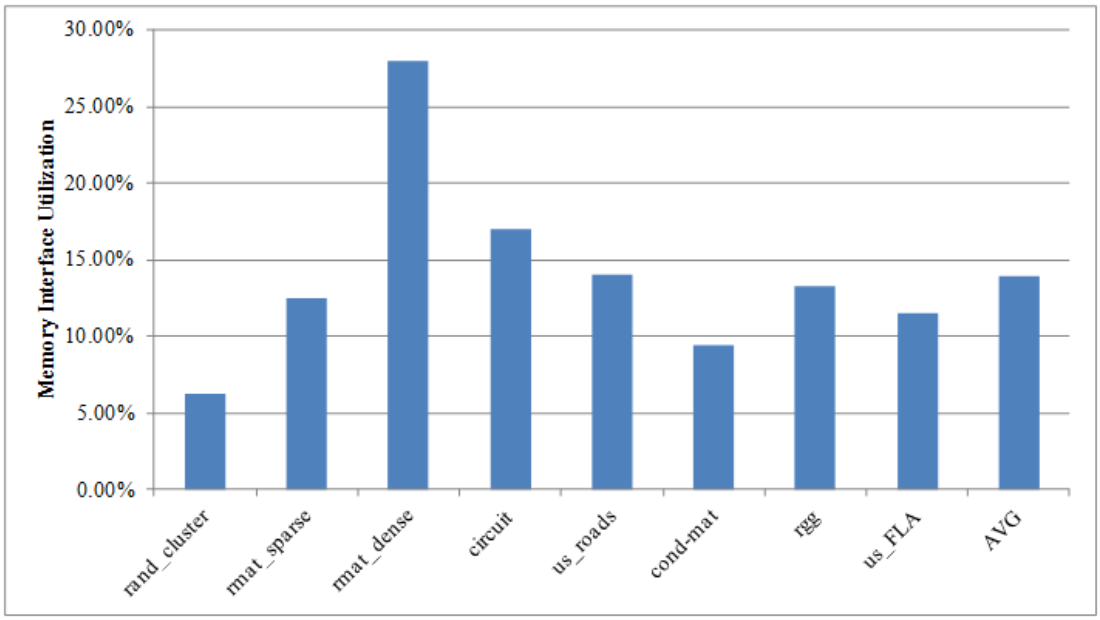


Figure 13. Utilization of the memory interface.

Memory Interface Utilization. As the main design focus of this accelerator is to maximize MLP, we first verify the accelerator's utilization of its memory interface. Across the datasets, we observe that the average length of worklist is longer than 500 on our accelerator, and the worklist is empty during less than 2% of the total execution cycles. This implies that the accelerator is unlikely to starve due to lack of tasks, and the pipeline should be constantly busy with pushing out memory accesses when the memory interface is available to accept more memory requests.

However, after profiling the activities at the memory interface, we noticed that the memory interface utilization is constantly lower than 30%, as shown in Figure 13. There are

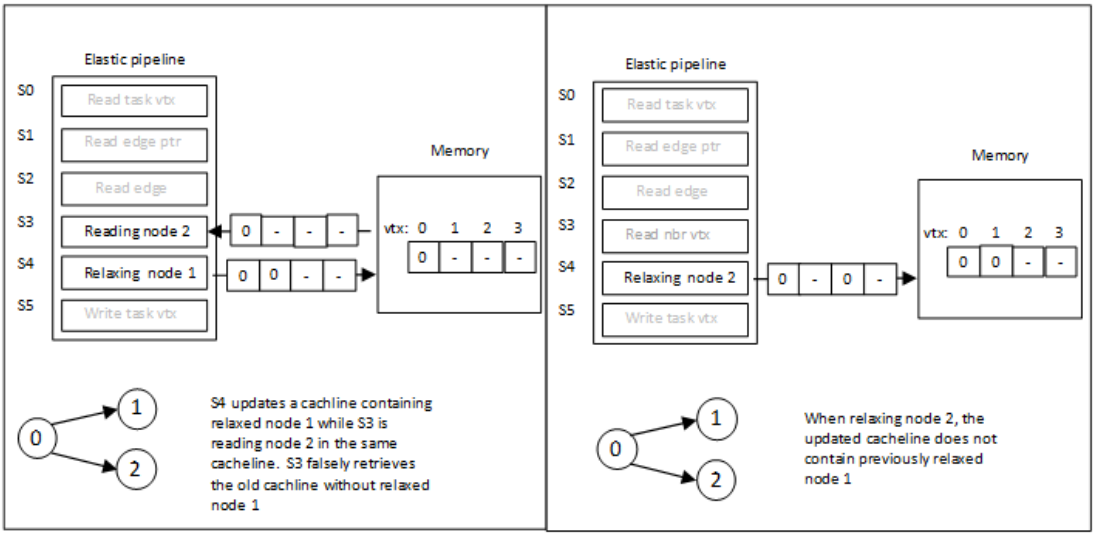


Figure 14. Illustration of hazards in our baseline accelerator.

two potential causes of this issue: 1) the accelerator had already saturated the maximum capacity of parallel memory operations that can be serviced by HARP, and additional requests cannot be accepted by the memory interface, or 2) the pipeline stalled due to data hazards. Scenario 1 is what we strive to achieve, as it implies that the accelerator is fully utilizing the memory sub-system and maximizing MLP. Unfortunately, our further study reveals that over 57% of the stalls are caused by the data hazard.

The hazard that causes the pipeline stall here is associated with a mechanism we employed in our data consistency model. As the there is a delay for each update to propagate to memory, when retrieving the label of a neighbor vertex in S3, if there is a non-retired write to the same address in S4 happening in parallel, the read can falsely retrieve the old cacheline value. This effect is illustrated in Figure 14 and results in a violation of data atomicity. To

maintain a consistent view of the vertex data, we employ a simple lookup table-based locking system to track the data dependency of accesses. The locking table maintains a record of addresses that are currently accessed by the later stages of the pipelines. If a new relaxation attempt tries to access any of those addresses, it would force serialization by blocking the new access until the earlier relaxation attempt completes and releases the lock, resulting in pipeline stalls. The results of our characterization study suggest that this hazard incurs over 65% of the stalls.

This symptom is not unique to our baseline accelerator. Prior works for CPU software [17] and FPGA [38] also encountered similar issues. For task-parallel design, this problem is commonly mitigated using an advanced scheduling technique [17] [38] [39] but cannot be fully avoided due to the complexity of the inter-thread/block communicating across multiple CPU cores/processing elements. In our case, the conflict is within the pipeline and can be completely resolved using the traditional techniques of data forwarding and write buffer that are commonly found in modern processors, which we will elaborate upon later in Chapter 5.

Off-Chip Access Reduction. Compared to optimizing MLP by increasing the number of concurrent memory operations, a more fundamental approach to tackle the latency problem is to reduce expensive off-chip accesses by improving data reuse. As previously pointed out, most commercial FPGA platforms are limited in their off-chip DRAM performances compared to CPU- or GPU-based systems; thus it is critical to efficiently utilize FPGA's highly customizable on-chip storage devices, including Flip-Flops and Block

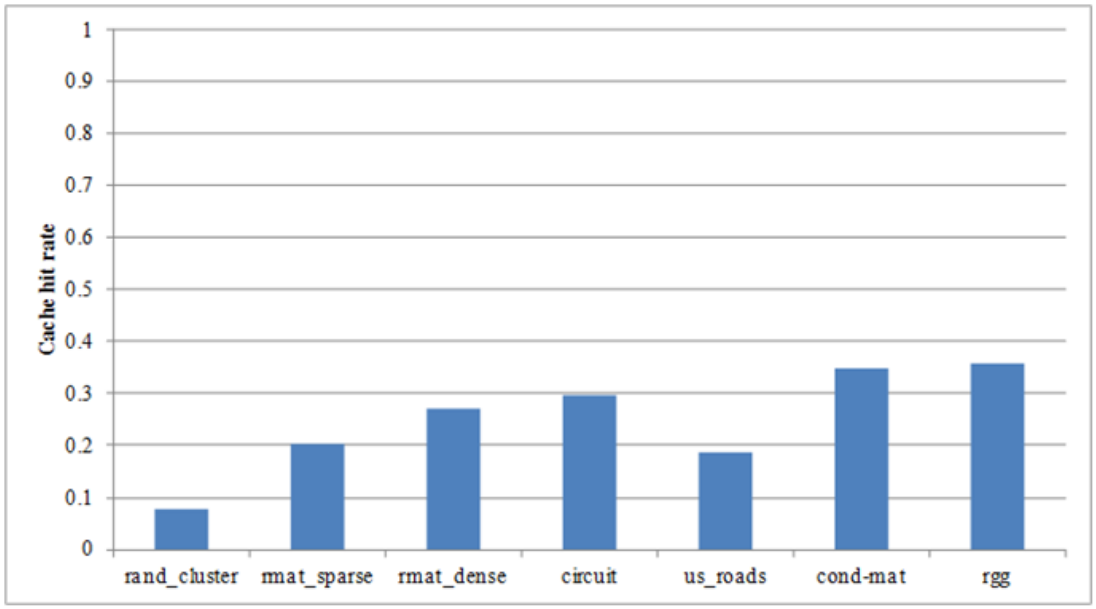


Figure 15. Cache hit rate of the unified 16KB CCI cache.

RAMS for buffering usable data. The platform we target in this experiment, Intel HARP, provides by default a 16KB cache in its data coherence interface with the CPU . We profiled the hit rate of this CCI cache for our BFS elastic pipeline across a set of selected graphs. The result, illustrated in Figure 15, shows that this default cache in ineffective in retaining reusable data, as the miss rate is consistently over 60% across all graphs.

After further investigation, we discovered that, most vertices are accessed at least twice in the graphs in our selected dataset; this implies that the misses are unlikely to be compulsory misses. Next, we performed a sweep of different cache capacities in the system configuration and measured the corresponding improvement in the cache miss rate, which suggests that those misses are unlikely to be capacity misses as we observed only limited improvement in the miss rate with a larger cache, as shown in Figure 16. The results imply

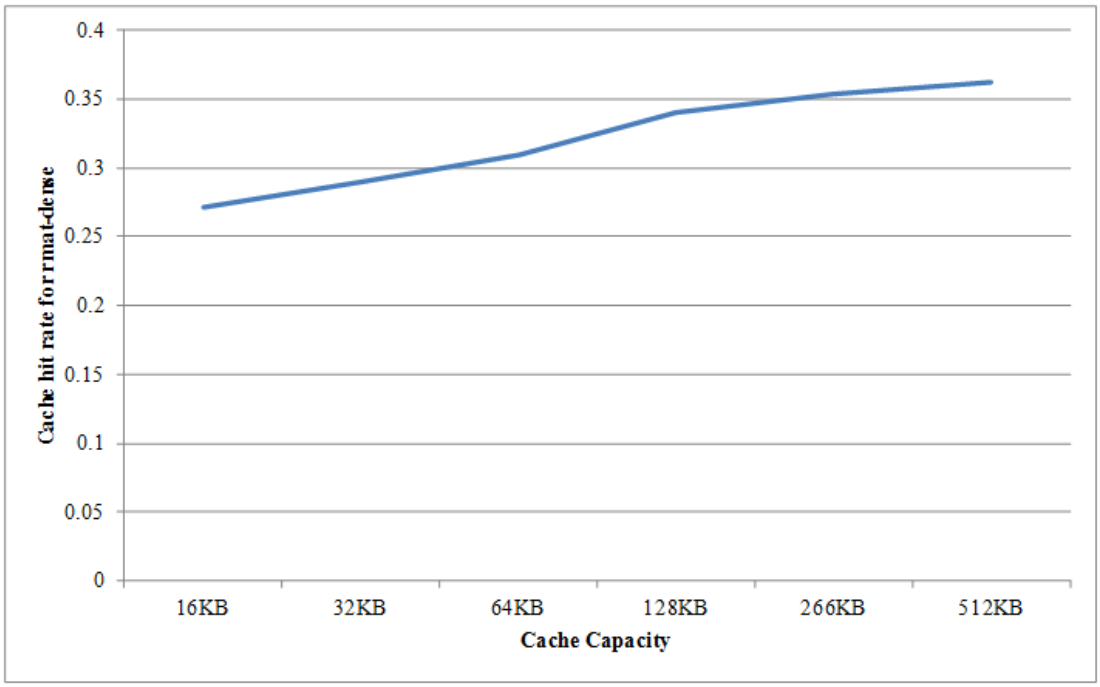


Figure 16. Hit rate for RMAT-sparse for different cache capacities.

that the CCI cache is likely dominated by conflict misses. After further investigation, we found that, on average, 30% of those misses are conflicts between edge and vertex accesses, and 45% are conflicts within vertex accesses. While increasing associativity can help reduce conflict misses, implementing a high-associative cache that meets the timing requirement for a fast-clocked accelerator can be a challenge. Later in Chapter 5, we will present a simple yet effective approach to mitigate this problem.

Inefficient DRAM Bandwidth Utilization. In addition to the low cache miss rate, we noticed a potential problem associated with the DRAM writes on a share-memory FPGA

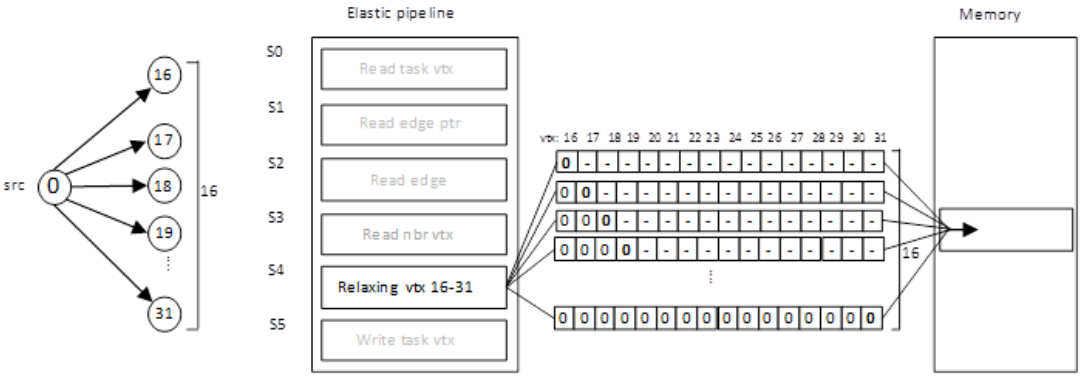


Figure 17. Illustration of the underutilization of write bandwidth during neighbor relaxation.

system. When a node relaxation succeeds, the baseline BFS accelerator issues an update to DRAM for the corresponding cacheline. The QPI interconnect automatically maintains the coherence of the node data at the LLC level on the CPU side. There are two drawbacks to this operation. First, each vertex update requires the writing of the full 64-byte cacheline to memory. The coherence protocol can potentially evict existing entry in the CPU LLC for accommodating the new modification. This can impact the performance if the user wants to parallelize BFS accelerations with other applications on the processor. The second drawback is the inefficiency in write bandwidth utilization. If there are 16 consecutive updates to the different vertices stored in the same cacheline, as illustrated in Figure 17, the accelerator will issue 16 separate 64-byte writes to the same address, while each write only effective changes a 4-byte word; thus, only 16/64=1/16 of the bandwidth is effectively utilized. This phenomenon wastes large amount of memory bandwidth and can potentially increase the latency for other memory requests from the accelerator or the CPU.

CHAPTER 5. OUR SOLUTIONS IN WORK 1

We have identified the bottlenecks that limit the performance of our baseline BFS accelerators, namely: 1) dependency stall in the neighbor-fetching stage; 2) poor on-chip data reuse; and 3) inefficient write bandwidth utilization. In this chapter, we perform in-depth analysis and propose solutions to tackle each of them.

5.1 Data Forwarding to Tackle Pipeline Stalls

In the last chapter, we discovered that the neighbor-fetch stage in our baseline accelerator encounters frequent stalls due to a dependency hazard. Specifically, this hazard is associated with the conflict in vertex data accesses in stage S3 and stage S4 in Figure 10, which results in large amounts of pipeline stalls in S3 as its read issuing has to wait until the dependent cacheline update from S4 to be committed in order to guarantee the consistency of data. Fortunately, we can eliminate the stalls by directly forwarding the modified cacheline from the outgoing channel buffer of stage S4 to stage S3 to prevent it from waiting for the update

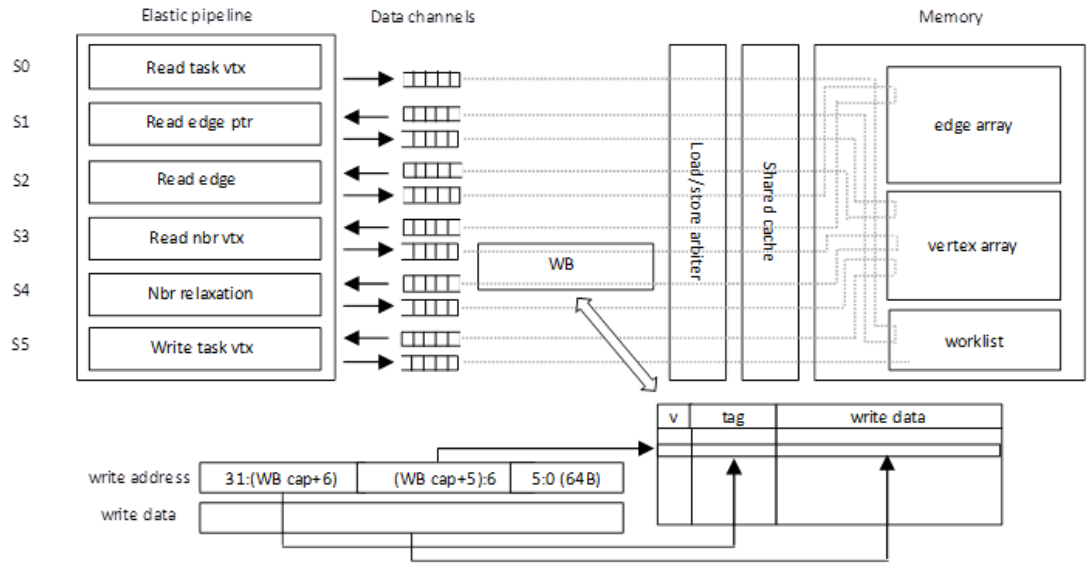


Figure 18. Block diagram of our accelerator with a write buffer.

propagation to memory.

Utilize a Write Buffer for Data Forwarding. The abovementioned data forwarding requires an additional storage structure called a **Write Buffer (WB)** to buffer the data from the write requests that have been issued but not yet committed. WB is a technique commonly used in CPUs for masking the latency of long memory writes. In our accelerator, WB is used in the scenario that a dependent read in S3 happens after the associated write has already been issued and exited the pipeline; thus, the needed data cannot be forwarded directly from stage S4. In this case, if the stored data in WB has a matching address with the read in S3, it can be passed over as the results of the read.

How Write Buffer Works. The base architecture of our WB design is shown in Figure 18. Structure-wise, WB essentially resembles a cache, in which each entry contains three fields: 1) the valid bit, 2) the 32-bit address tag of the store request, and 3) the 64-byte write data. Like a standard cache, the WB is indexed using the lower-order bit of the address. Before issuing a read for a vertex label, the S3 FSM performs a WB lookup to verify status of the corresponding address, and four possible statuses can be returned from WB:

1. WB entry is free, and the address of the entry does not match the address of the vertex to be loaded.

2. WB entry is free, but the address matches.

3. WB entry is locked, but the address mismatches.

4. WB entry is locked, and the address matches.

When a WB entry is locked, it means there is an active relaxation operation associated with the registered address currently occupying the entry. Otherwise, the entry is free to be allocated for a new read. Status 1 represents the scenario that the entry is free; hence, a new read can book the entry for an address that is different than what was stored in the entry. If the address pointed by the WB entry luckily matches that of a new request, no additional memory read needs to be issued, as they can be directly forwarded on-chip. However, there are two different cases in this scenario that affect where the data should be forwarded from. If the previous dependent transaction has completed and the data have already been written to WB (so the WB entry has been locked), represented as status 2; then,

the data should be forwarded from WB. If the WB entry is still locked, represented as status 4, then the dependent data are currently being processed down the pipeline and are not ready to be forwarded yet. This case requires that an updated cacheline be forwarded later from stage S4, when its processing completes. Finally, status 3 indicates the undesirable scenario in which the WB entry is locked for an ongoing memory request of a different memory address. This reflects a structural hazard rather than a true data dependency, as it is associated with the conflict of mapping memory operations of different addresses to the same WB entry; thus, the operation in S3 has to stall until the locking operation completes. This problem can be mitigated by increasing buffer capacity or associativity. Compared the baseline accelerator without WB, in which the hazards in case 2 and 4 would have stalled the pipeline, the introduction of WB prevents those stalls and helps to reduce memory traffic/access latency by retrieving the data directly on-chip.

5.2 Partitioned Buffer for Improving Reuse

Our study in Chapter 4 discovered that the generic 16-KB CCI cache performed inefficiently due to conflict misses. The first optimization we propose to tackle this problem is to avoid the inter-data type of conflict by partitioning and isolating the buffering of different data types. In general, the worklist is read and written sequentially in a streaming pattern, exhibiting great spatial locality. For each worklist cacheline brought from off-chip DRAM to the accelerator, it likely contains multiple task vertices and should be buffered to save future re-reads. To exploit this spatial locality, we allocate a small 64-byte, register-based cacheline buffer exclusively for the worklist data. For the edge-related accesses, the adjacency list, the

array where edges are stored in standard graph formats such as CSR [30], exhibits spatial locality, like the worklist. In addition, each edge is accessed strictly once in BFS, so there is no temporary locality and no need to allocate a large cache to save edge data for future reuse. Hence, we adopt the same strategy of using a single cacheline-size buffer.

By isolating the worklist and edge data from the cache, we allocate a cache dedicated for buffering only vertex data. The motivation behind a large, dedicated vertex cache can be summarized as follows:

1. Vertex accesses are, on average, responsible for more than 60% of the total number of memory traffic in our BFS accelerator; thus, it is logical to improve its performance by providing a large, dedicated cache only for vertex data.

2. In comparison to the other data types, there exists a lot more data reuse in vertex labels, as there tend to be multiple incoming edges to each vertex in a real-world graph; thus, cache data can be used multiple times.

3. Unlike the edge and worklist data, vertex accesses are mostly gather-scatter operations with very little spatial locality, and the technique of streaming buffer applied for worklist and edge data is not applicable. To capture the temporal reuses, cache is a logical design choice.

Integrating vertex cache and WB. Creating a large-capacity cache for exclusively buffering vertex data would consume a large number of FPGA BRAM, which is also used for the implementation of the WB for data forwarding, thus it makes sense to combine the two.

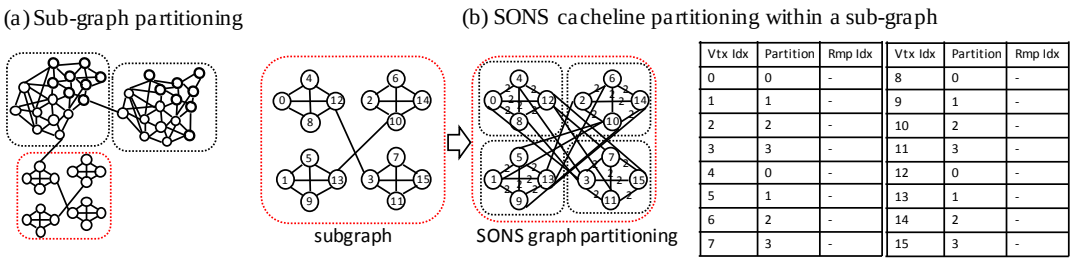
The function of the WB overlaps heavily with a vertex-exclusive cache, as both are temporary on-chip storage for vertex labels. The key difference is the entry replacement policy. Traditionally, WB updates its entry only when a memory write happens and invalidates its entry when the write is committed, whereas a cache is updated both for read and write, and the entries are not invalidated at the completion of writes. In contrast, our new buffer that merges WB and cache, which we will refer as Vertex Cache (VC) in the remaining of this document, follows the following modified replacement protocol.

1. An entry is updated for both non-dirty read and write.

2. Unlike WB, VC's entry is not invalidated after the completion of a write operation for potential future reuse.

As a side note, if the WB is to be used purely for hazard avoidance, the maximum number of entries the WB ever needs would be same as the number of possible outstanding writes the system can support. This size guarantees any non-completed write would have a buffer entry to accommodate its updated cacheline for forwarding purposes, hence stalls can be fully avoided. In the case of VC, we generally prefer higher capacities, as they help to reduce miss rates and improve data reuses.

5.3 Improving Locality Using Graph Partitioning-Based Preprocessing.



Vtx Idx	Partition	Rmp Idx
0	0	-
1	1	-
2	2	-
3	3	-
4	0	-
5	1	-
6	2	-
7	3	-

Vtx Idx	Partition	Rmp Idx
8	0	-
9	1	-
10	2	-
11	3	-
12	0	-
13	1	-
14	2	-
15	3	-

Figure 19. Khoram's hierarchical graph preprocessing.

The limited locality in its irregular, pointer-based memory accesses is the fundamental challenge to graph processing. Without locality, the inner loop in Algorithm 1 for neighbor relaxation can result in near-random accesses and poor cache performance. One common technique to tackle this problem in graph processing is to pre-process the input graphs based on the results of graph partitioning.

Prior Work: Hierarchical Partitioning-Based Vertex Reindexing. Graph partitioning and clustering algorithms are effective techniques for analyzing the topological structure of a graph. It finds closely connected vertices likely belonging to the same **community** and assigns them to the same partition. A community is a sub-graph that is densely connected internally. During the process of relaxation, due to the clustering nature, consecutive accesses are more likely to happen within the same community, and the cache can capture this locality if those topologically close vertices are mapped to physically adjacent locations in the memory space. In prior work from Khoram et al., the partitioning results have been used to remap vertex indices for better data locality [11]. The proposed vertex reindexing scheme re-assigns the vertices of the same partition with adjacent,

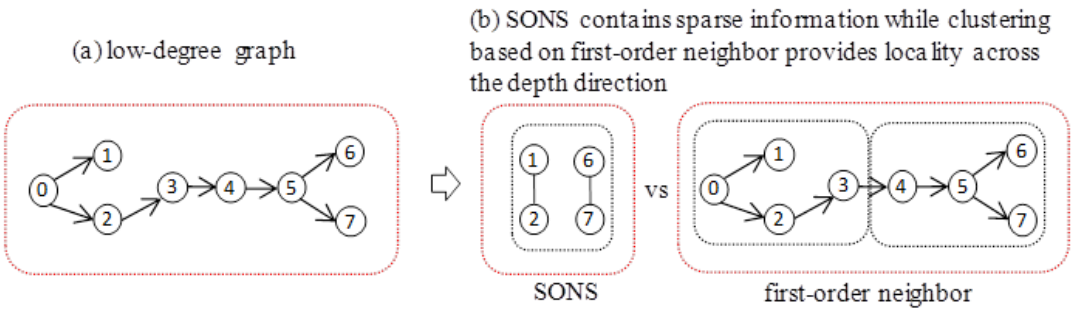


Figure 20. Illustration to show why a first-order neighbor is a better target for clustering for a low-degree graph in comparison to SONS.

continuous indices, increasing the probability that the vertices from the same cacheline are the outgoing neighbors from the nodes in a graph.

After understanding the high-level objective of Khoram's pre-processing optimization, let's dive deeper into the details of the implementation. The graph pre-processing composes two hierarchical steps, as illustrated in Figure 19. At the first level, the input graph is partitioned to sub-graphs sized to reflect the capacity of the on-chip storage. The goal is to identify communities with large numbers of internal connections and small enough to fit on-chip for reducing spilling to memory. All vertices from the same partition will then be assigned with consecutive indices. The second-level reindexing maximizes the spatial locality for adjacency list walking within each graph partition. This process involves two steps: 1) forming Second-Order-Neighbor Subgraphs (SONS), and 2) partitioning SONS to identify how to group vertices to the cacheline such that the spatial locality within each adjacency list is maximized.

SONS is a special auxiliary graph that connects second-order neighbors, or the vertices neighboring to a same parent vertex. When walking through the neighbors of a task vertex to determine if they should be relaxed, the memory traffic would be greatly reduced if most of the accesses hits the same cachelines. Partitioning SONS accomplishes this goal by mapping the vertices that are frequently second-order neighbors to the same cacheline partition for better locality.

The Limitation of SONS-Based Pre-Processing. While Khoram's vertex reindexing scheme is effective for graphs with large vertex degrees, meaning each vertex contains a large number of outgoing edges. For long diameter graphs, in which vertices tend to have low degrees, there would be little information to be captured by the SONS graph, as illustrated in the example shown in Figure 20. In this case, the SONS representations of these graphs are extremely sparse and thus cannot serve as useful hints for identifying locality-friendly cacheline groupings.

Our Improved Preprocessing Scheme: Adaptive Vertex Reindexing. Due to the low numbers of second-order neighbors in long-diameter graphs, we argue that a better approach to improve cacheline locality for low-degree graphs is to instead group first-order neighbors by partitioning the original graph. This strategy is usually less effective compared to the SONS-based solution for high-degree graphs, as it mixes a predecessor vertex with its neighbors during the cacheline layout, which introduces noises to the neighbor traversal (the inner loop of Algorithm 1) and can potentially lower the spatial locality. However, for low-degree graphs, the number of iterations for the inner loop is small, and the outer-loop

```
1:  sub_graph[]=partition(graph, sub_graph_size);
2:  for(int i=0;i<sub_graph.size;++i) do
3:      if sub_graph[].degree>alpha then
4:          sub_graph[i]=partition_SONS(sub_graph[i], cacheline_size);
5:      else then
6:          sub_graph[i]=partition_first_order(sub_graph[i], cacheline_size);
```

Algorithm 2. Algorithm of the proposed preprocessing optimization, DAVR.

operation dominates. In this scenario, graph traversal resembles link-list chasing, and the locality improves if the first-order neighbors are placed in the same cacheline as they would be accessed consecutively across the outer loop. This observation is illustrated in Figure 20.b and leads to our **Degree-Aware Vertex Reindexing (DAVR)** scheme.

DAVR is a graph pre-processing optimization built on top of the Khoram's hierarchical preprocessing optimization. While DAVR performs the same first-level sub-graph partitioning based on on-chip storage capacity, instead of focusing solely on the locality among the second-order neighbors in the second-level sub-graph pre-processing, it adaptively selects between first-order or second-order neighbors partitioning, according to the average vertex degree of the graph. The overall flow of DAVR is shown in Algorithm 2. The parameter *alpha* is used for selecting between first-order or second-order vertex reindexing. When the average degree of a sub-graph is lower than it, first-order reindexing is applied; otherwise, second-order reindexing is used. The value of alpha is currently empirically determined. From our experiments, we found the 1.3 to be a good value.

5.4 Memory Coalescing.

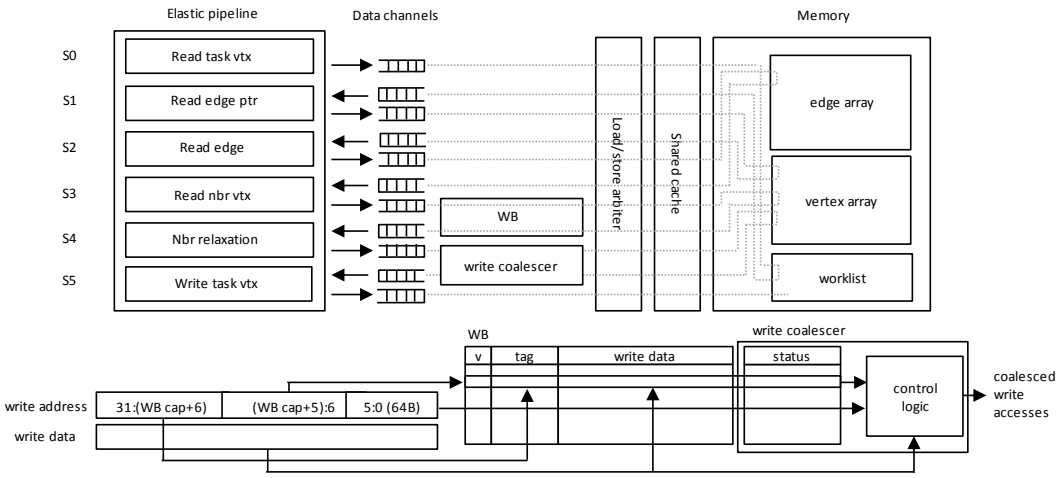


Figure 21. Block diagram of our accelerator with WB and write coalescing support.

The 4-byte update operation in the S4 stage of the baseline BFS accelerator is inefficient in bandwidth utilization, as the memory interface operates at 64-byte cacheline granularity. An approach to tackle this problem is to apply memory coalescing [40], which combines multiple sequential writes to the same address as a single write request. The coalesced cacheline contains multiple modified vertex indices instead of just one and thus improves the effective bandwidth utilization, as a higher fraction of the cacheline contains modified data. As the large number of vertex changes are now packed to fewer number of cacheline updates, coalescing effectively lowers the QPI interconnect traffic.

To support write coalescing, hardware changes to our baseline BFS accelerator is necessary. The architecture of the coalescing module is shown in Figure 21. It consists mainly of two components: the control logic and the status buffer. The status buffer is an add-on table to the WB that tags each entry with a status metadata. Just like the WB, it is indexed using the lower order bits of the address of a memory-write request. Each entry in

the status buffer represents the storage status of its corresponding address with two bits, and there are three possible statuses:

- The invalid state 0. This state represents that the entry is invalid, and there is no unfinished write in progress with the address.
- The "on-flight" state 1. This state represents that there is an unfinished write currently in progress.
- The "dirty" state 2. This state represents that there is more than one write currently in progress.

To understand how this design works, each entry in this status buffer essentially functions as a small-state machine for determining if new writes can coalesce with pending write requests. When a write request is issue from stage S4 of the BFS accelerator, the coalescing module will use the address to check its storage status. If it is in the invalid state, it implies that there is no outstanding write to the same address. The control logic will issue the write to memory normally and increment the state to 1, as there is now one memory request "on-flight." If it is in the on-flight state, there is currently one write to the address in progress. This new request is buffered in WB as a pending request and will be issued later when the competition signal for the previous write is received. Then the state is changed to "dirty," because the WB entry now contains a modified cacheline data that has not been written to memory yet. If a write happens when the state is dirty, it represents an opportunity of coalescing. The WB overwrites its entry with the newly modified cacheline that contains

all prior changes, which will be written to the memory when the ongoing write request commits. When the state machine confirms the completion response for a write, it dispatches a new write request for the coalesced cacheline stored in WB if the status is dirty. Finally, after the coalesced request is serviced, the state goes back to invalid and becomes ready for accepting write requests to a different address.

CHAPTER 6. Evaluation

In this section, we will evaluate the four proposed optimization techniques for our BFS accelerator—namely: 1) data forwarding, 2) partitioned on-chip data buffering, 3) write coalescing, and 4) our proposed graph pre-processing optimization AVR. The experiments were conducted using the Stratix V FPGA on an Intel HARP. The only exception is a cache capacity/associativity study in Section 6.1, which is difficult to executed on a physical HARP as the CCI cache is provided as encrypted IP from Intel, which prevents us from altering its configurations. Instead, we collected results for simulation-based experiments using our in-house TLMSIM simulator. We keep the same datasets from Chapter 4 for evaluation. We will first present separate case studies for each of the three proposed optimization techniques and then an aggregated analysis to understand how well they stack up.

6.1 Data Forwarding and Data Buffer Specialization

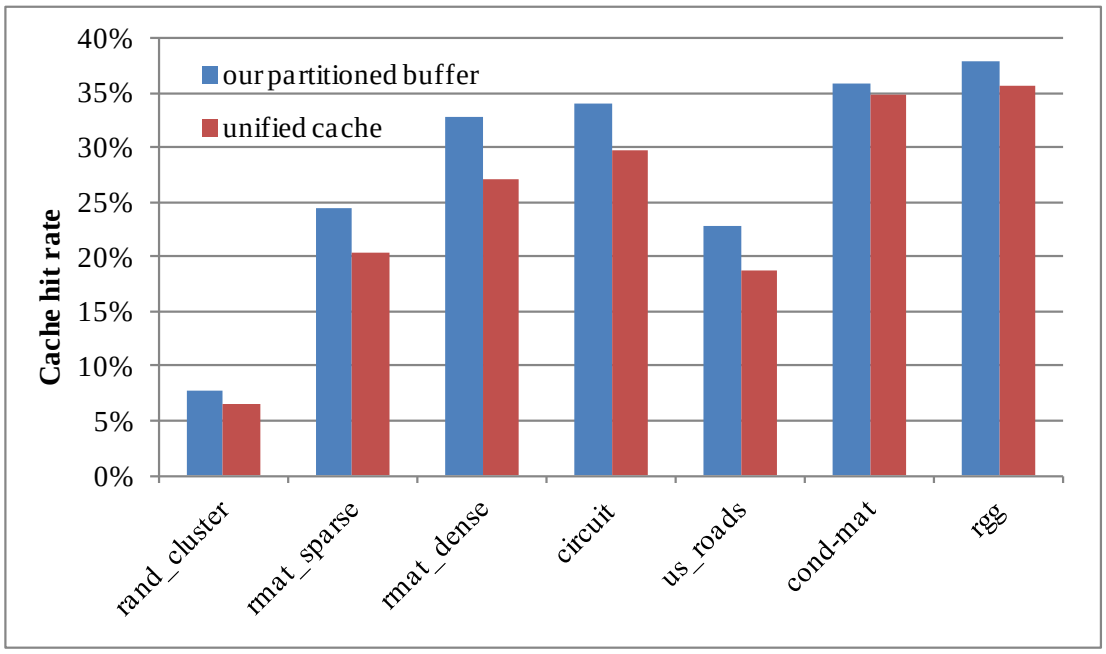


Figure 22. Comparison of cache hit rates of a unified cache and our partitioned buffers.

In this work, we propose a partitioned buffer scheme, which separates the buffering of worklist, edge and vertex data. To evaluate its effectiveness, we compare it with a unified buffer with the same capacity. In this experiment, we set the set-associativity of the unified cache and VC both to 4 and the total capacity to 64KB, which is the maximum setting supported by our accelerator without violating timing during the compilation flow. Figure 22 shows the results. Overall, our proposed partitioned buffering scheme consistently outperforms the standard approach of a unified cache by an average of 14% due to lower inter-data-type pollution and conflict misses. As for the effectiveness of data forwarding, it successfully eliminates 94% of the related stalls and exposes the new bottleneck of the memory system, as the accelerator now stalls mainly due to reaching the maximum parallel accesses supported by the interface.

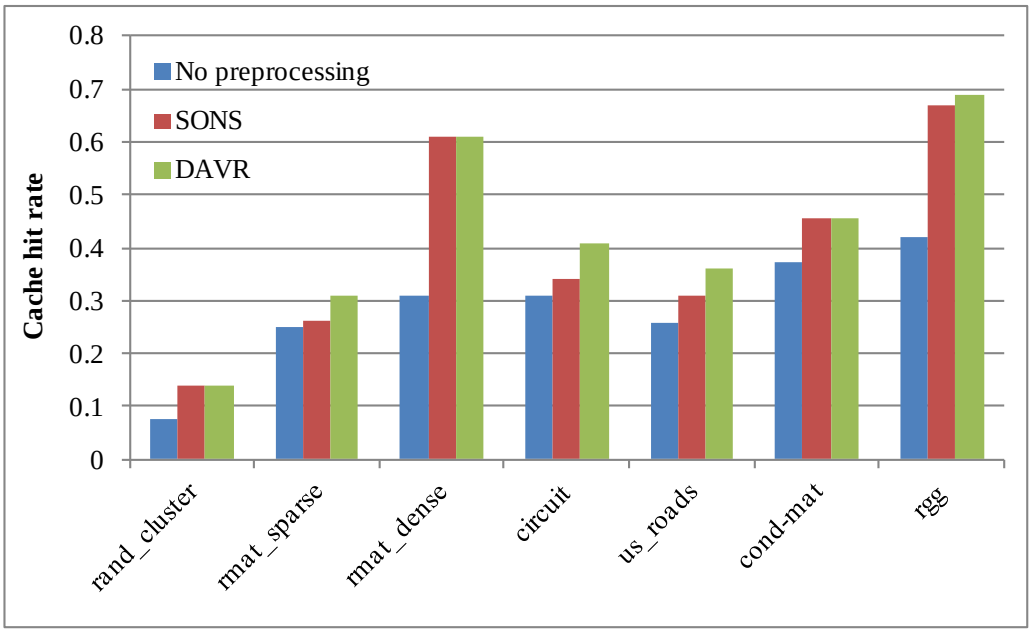


Figure 23. Hit rates of different preprocessing schemes on a 64KB cache.

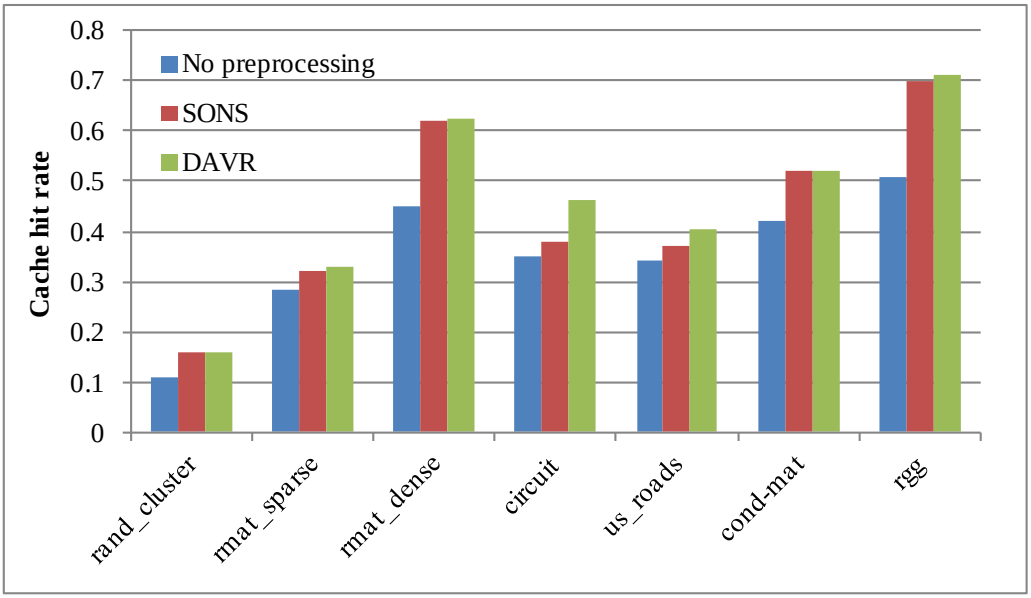


Figure 24. Hit rates of different preprocessing schemes on a 512KB cache.

6.2 Vertex Reindexing.

To evaluate our proposed adaptive graph pre-processing optimization, DAVR, we compared the cache miss rate of DAVR on our BFS accelerator with the SONS-based hierarchical vertex reindexing from [11] and the graphs without any pre-processing. We conducted two experiments: one with a large VC of 512KB, and the other with a small VC of 64KB. The set-associativity is again four in both cases. The results are shown in Figures 23 and 24. In the 64KB case, both reindexing schemes deliver a significantly improved cache hit rate compared to graphs without preprocessing. On average, SONS-based pre-processing improves the hit rate by 42%. Our proposed DAVR delivers an additional 18% improvement over SONS for the three low-degree graphs (rmat_sparse, circuit, us_roads) and 8% for all graphs.

This study demonstrates vertex reindexing to be an effective method to increase the locality in a graph. As our preprocessing solution DAVR outperforms the state-of-the-art SONS approach, this confirms our theory that exploiting the variation of the average vertex degree in different graphs is the key to improving locality for long-diameter graphs.

6.3 Write Coalescing

Write coalescing is a hardware optimization for reducing the number of the memory transactions from vertex relaxations. To quantify its effectiveness, we need to first define a metric for quantifying write reduction. We define the number of memory writes to update vertex data without coalescing in the stage 5 of the baseline pipeline as *Ub* and the one with coalescing as *Uc*, and the *reduction ratio* is defined as *(Ub-Uc)/U*, which is equivalent to the

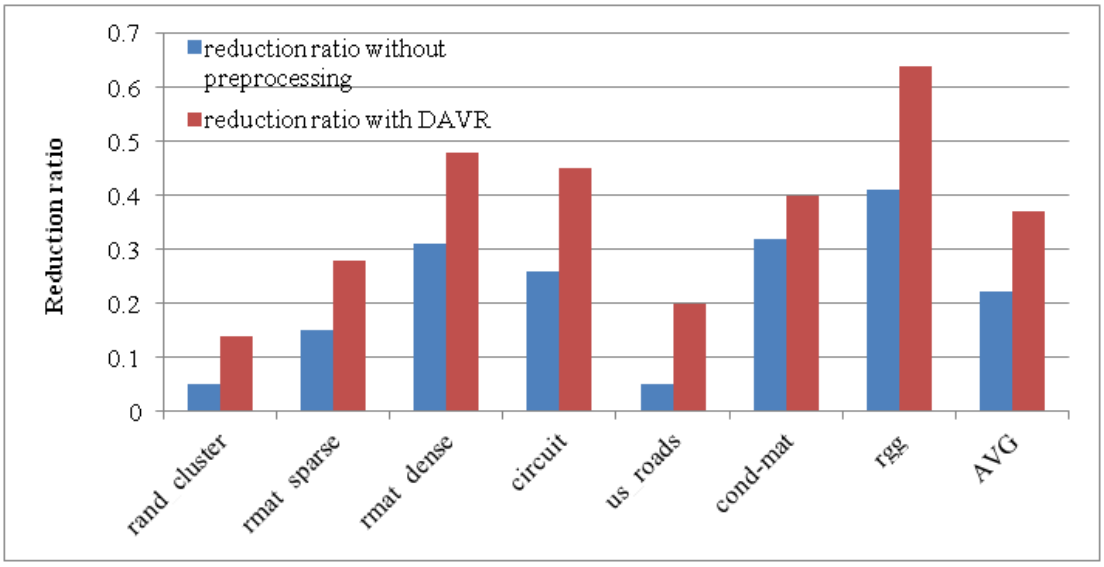


Figure 25. Write traffic reduction brought by memory coalescing.

fraction of memory writes that are eliminated by coalescing. The coalescing rates of our BFS accelerator with 512KB four-way associative VC is shown in Figure 25. Without vertex remapping applied, the reduction ratio is, on average, 22%. With the proposed DAVR graph processing, we observed a significantly improved reduction ratio of 37%, on average.

6.4 Overall Performance Improvement

Performance with proposed Hardware Optimization. In this section, we evaluate the application processing throughput for the proposed optimizations. The experiments are conducted with the 512KB, four-way associative VC. Figure 26 shows the processing throughput improvements brought by cascading vertex-caching and write coalescing. On

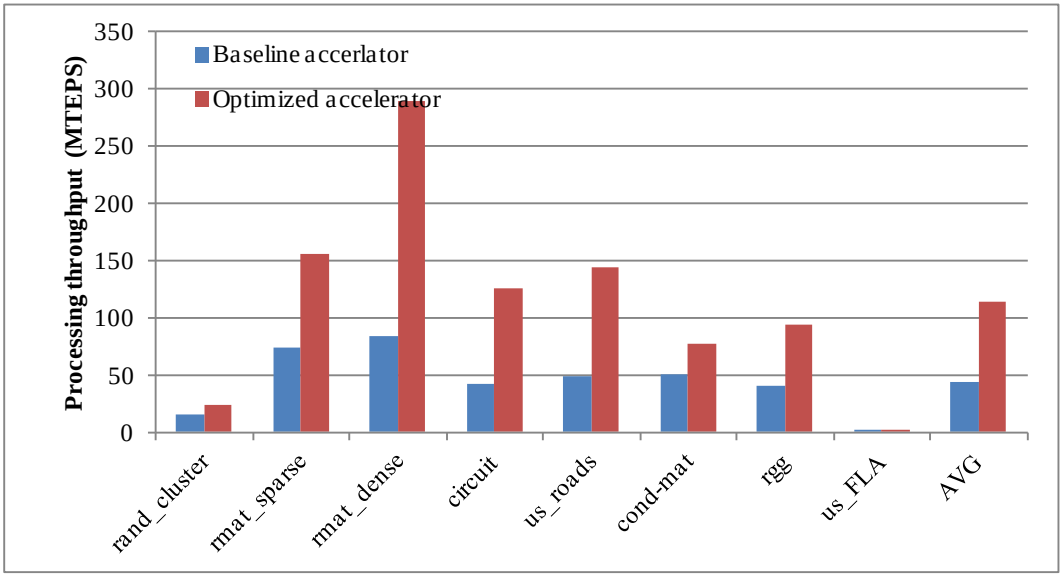


Figure 26. Processing throughput of the optimized accelerator.

average, the optimizations deliver an average 2.6X improvement over the baseline accelerator. We then applied DAVR to pre-processes the input graphs and repeat the experiments. The results are shown in Figure 27. On average, this configuration delivers a 4.13X speedup over compared to the baseline accelerator with non-processed graphs.

Comparison with Software-Based Solution. Finally, we compared the performance of our implementations with Galois [17], a highly optimized irregular processing framework for CPU-based systems. Galois provided multiple BFS implementations. We used the default "barrier" option, as it generally produced the best performance and configured the number of threads to be 20, the maximum supported by our system. The experiments were conducted on the CPU host of an Intel HARP, which features a Xeon E5-2680 V2 processor at 2.8 GHz and 128GB DDR3 memory. Figure 28 shows the

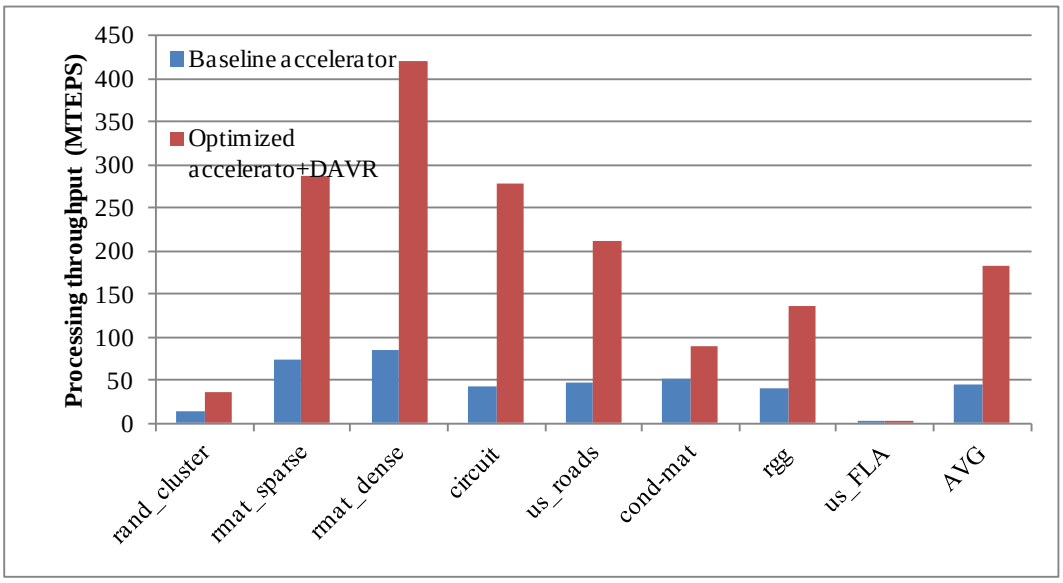


Figure 27. Processing throughput of the optimized accelerator with graph preprocessing.

performance comparison of Galois and our BFS accelerator with all proposed optimization applied (512 KB, four-way associative VC, write coalescing and DAVR preprocessed graphs). Overall, our accelerator performed comparably with Galois for RMAT-16, USA-east and RGG, with the average throughput difference to be less than 7%. For Cond-mat, our accelerator was outperformed by Galois by 67% due to a high miss rate in its vertex cache. In terms of power consumption, Galois consumes over 100 watts consistently across the four datasets, while our accelerator operates at a power budget of 8.3 watts, which makes it the better option for energy efficiency or performance per watt.

Discussion. Despite the Xeon having a higher-bandwidth, lower-latency memory system compared to the FPGA, the software BFS implementation only achieves roughly the same performance as the accelerator, which implies memory resource underutilization for the

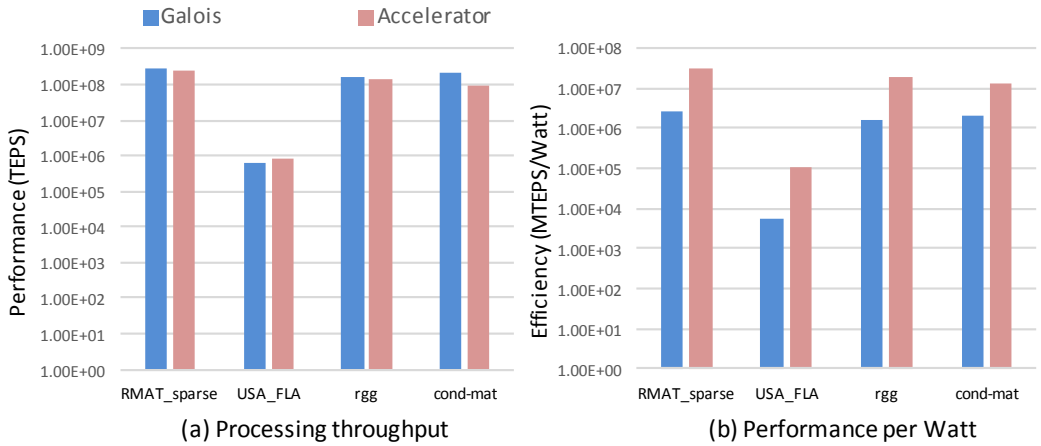


Figure 28. Comparison with Galois.

processor. Upon closer examination, we discovered that the processor fails to extract parallel memory reads from the instructions, as it issued only one load every 7.8 clock cycle on average. This implies severe underutilization of the CPU's memory interface, which can theoretically accept two loads per clocks cycle in each core. For our 10 core Xeon, this translates to 20 loads per clock cycle. In comparison, the baseline accelerator is more efficient at this task. It issues one load every 1.3 clock cycle. Given that the memory interface accepts only one read per clock cycle, this translates to a 90% utilization.

The problem of memory interface underutilization is not unique to Galois, as we also observed the same problem on another BFS implementation [19]. Many factors can contribute to this bottleneck. One probable cause is the CPU's inability to extract parallel loads within the inner-loop body for neighbor relaxations (lines 7 to 14 in Algorithm 1) as there exists dependency among the memory instructions.

Part II. Processor-Assisted Scheduling for Irregular Applications on Shared Memory FPGAs

CHAPTER 7. WORKLIST SCHEDULING REVISITED

Worklist-Based Graph Algorithms. Graph analytics are powerful tools for determining the relationships among connected components in a graph. While differing in function, many graph analytic algorithms can be abstracted as a worklist and an iterative processing routine [17]. The worklist is a data-structure that stores a set of pending tasks, often implemented as a queue; the processing routine iteratively dequeues and complete those tasks until the worklist is empty. With the freedom to define the task and the processing routine, this worklist-based model can be generally applied to represent different computations. As a graph computation progresses, new tasks are generated in an order determined by the topology of the input graph and the execution phases, which tend to be highly irregular. While this irregularity imposes a great challenge for extracting parallelism and exploiting data locality, it presents an opportunity for worklist scheduling as the naturally occuring

insertion order of tasks is often not the ideal processing order.

Worklist Scheduling. For many graph algorithms, the tasks stored in the worklist can be scheduled to be processed in any order without affecting its functional correctness. Examples include Breadth First Search (BFS) and Single Source Shortest Path (SSSP). Even though the standard first in, first out (FIFO) queue is frequently used as a worklist for in-order processing because of its low implementation complexity, there exists better processing schedules that can improve performance by exploiting algorithm-specific information.

A classic example is the Dijkstra SSSP algorithm [5]. SSSP determines the shortest paths from a source vertex to the other vertices in a graph by iteratively expanding the frontier of search starting from the source. In each iteration, instead of processing the tasks in the worklist by their insertion order, the Dijkstra algorithm prioritizes the task vertex with the shortest path for frontier expansion. This scheduling optimization can reduce the number of tasks and processing time by an order of magnitude. In fact, this type of priority scheduling is very common in software-based graph computations. State-of-the-art graph processing framework such as Galois [17], GraphMat [19] and GunRock [22], all provide built-in support for task prioritization.

7.1 Worklist Scheduling on FPGAs

Graph computations are fundamentally memory-bounded, which imposes a challenge for software-based implementations to fully utilize the large, fast clocked cores in modern

processors. On the other hand, this provides an opportunity for FPGA-based solutions, as FPGA's fabric can be fully customized to target only the needed computations for achieving great performance and efficiency. However, FPGA-based accelerators rarely exploit the optimization of worklist scheduling. To examine the difficulty in FPGA-based scheduling, in this work, we conducted two case studies for worklist-based graph computations. The first case study focuses on the work-reduction scheduling of Dijkstra SSSP. The second case study targets a new locality-aware scheduling for BFS that we proposed, which improves on-chip data reuse by prioritizing the processing of the vertices in close topological proximity to one another. We facilitate the BFS accelerator developed in the first part of the thesis as the baseline design for conducting our study of worklist scheduling, which delivers comparable performance to a state-of-the-art software implementation [12] at much lower power and can be easily extended to support SSSP. We then added support for priority scheduling to the baseline accelerators and discovered that, in addition to the implementation complexity, this approach consumes a large amount of on-chip BRAM for caching the priority-queue. Without this cache, the long-latency accesses to the priority queue stored in memory overshadow the benefit of scheduling and degrade the overall performance. As the baseline accelerator also needs BRAM for caching vertex data, this results in a serious conflict for the limited FPGA resources.

7.2 Our Solution

To tackle this problem, we propose Processor Assisted Scheduling (PAS) on a shared-memory FPGA platform. A shared-memory system, such as Intel HARP [6], offers coherence

accesses to a shared off-chip memory for the FPGA and the host processor through a high-performance interconnect like Intel's QPI. This enables dynamic, fine-grain data-sharing and communication between the CPU and the FPGA. PAS harnesses the power of a shared-memory system by using FPGA for the standard graph processing routine and dynamically offloading the scheduling task to the host processor. The software scheduler exploits the sophisticated cache system and lower memory latency of the processor to deliver fast scheduling service, which helps the accelerator to obtain the benefit of worklist scheduling without consuming precious BRAM resources.

In our evaluation, PAS allows SSSP and BFS to obtain over 90% and 80% of the potential scheduling benefit on an Intel HARP, respectively. In addition, as the software scheduler spends most of the processing cycles waiting for new tasks to be produced by the accelerator or the priority-queue accesses to complete, this lightweight routine places a negligible load on the CPU and interferes minimally with other concurrent applications, making PAS ideal for servers. By allowing the processor and the accelerator to work on what they are good at, the collaborative processing of PAS achieves better performance and energy efficiency than each can individually deliver.

7.3 Our Contributions

In this second part of thesis, we made two major contributions:

- We developed, implemented and evaluated PAS on an Intel HARP. The evaluation results suggest that, with a very small communication overhead between the

processor and the FPGA accelerator, PAS allows the accelerator to obtain the majority of the performance benefit from worklist scheduling without consuming extra FPGA resource. Furthermore, we executed multiple standard CPU benchmarks along with PAS, and observe nearly no interference from PAS as it causes no observable performance degradation to those benchmarks.

- As part of PAS, we developed a new worklist scheduling for BFS, which combines graph-preprocessing and dynamic scheduling to prioritizes the processing of vertices that are likely in near topological neighborhood to improve the utilization of cache and overall performance.

7.4 Use Case of Worklist Scheduling: Reducing Tasks for SSSP.

The Inefficiency of In-Order Processing. SSSP is a graph traversal algorithm for discovering the lengths of the shortest paths from a source vertex to the remainder of the graph. A worklist-based implementation of SSSP supporting graphs in Compressed Sparse Row (CSR) format is shown in Algorithm 3. In this algorithm, the label of a vertex represents the length of its shortest path from the source. During initialization, all vertices are set to infinity, as the shortest paths have not been discovered yet, and then the worklist enqueues the source. SSSP then iteratively processes the tasks, or vertices with newly discovered shortest paths, from the worklist by attempting to "relax" their neighbors (line 4-16). Relaxation is the process of exploring the path spanning form the source to the task vertex

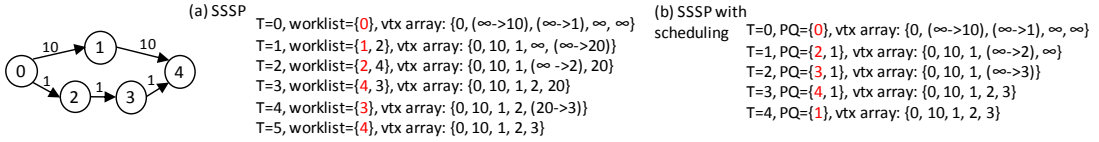


Figure 29. SSSP scheduling to reduce number of tasks/relaxations from 6 in (a) to 5 in (b). The dequeued task of each iteration

and then to its neighbor. If this path is shorter than the previous path assigned to the neighbor, the relaxation succeeds and a new shortest path is found. The process of relaxation involves two steps: 1. computing the length of the new path by summing the path length of the task vertex and the length of the edge to its neighbor, 2. updating the neighbor if the new path is shorter than what it is currently assigned with. Relaxed vertices become the new frontier of exploration and are inserted to the worklist as tasks for future relaxation. There are two important observations to be made here. First, while this implementation assumes an in-order FIFO for the worklist, the stored task vertices can actually be processed in an arbitrary order without violating the algorithmic correctness. Second, later relaxations which hops over more short edges can lead to a shorter path compared to an earlier relaxation which hops over fewer long edges. The example in Figure 29.a illustrates this scenario, where the upper path with 2 hops actually is actually than the lower path with 3 hops. This phenomenon creates an inefficiency because all vertex updates before the final relaxation are later overwritten and should be avoided at the first place as they incur unnecessary but expensive writes to memory.

Dijkstra-Based Scheduling. To tackle this problem, the worklist scheduler can prioritize the processing of the task vertices assigned with short path labels. As the path to their neighbors are likely also short, this helps to block future relaxation attempts and thus

```
1:    FIFO_worklist.enqueue(source);
2:    for(int i=0;i<num_vtx;++i)vtx_array[i].predecessor=-1;
3:    while !FIFO_worklist.empty()  do
4:     int task_vtx=FIFO_worklist.dequeue();
5:     int curr_edge_offset=vtx_array[task_vtx].edge_offset;
6:     int num_edge=vtx_array[task_vtx].num_edge;
7:     for num_edge to 0 do
8:      int curr_nbr=edge_array[curr_edge_offset];
9:      int nbr_vtx=vtx_array[curr_nbr];
10:     if vtx_array[nbr_vtx].predecessor==-1 then
11:      vtx_array[nbr_vtx].predecessor=task_vtx;
12:      FIFO_worklist.enqueue(nbr_vtx);
13:     end if
14:    end for
15:   end while
```

Algorithm 3. Baseline SSSP algorithm.

reducing the number of memory writes. Since there would be fewer successful relaxations, the number of tasks pushed to the worklist and the number of iterations needed for convergence are also reduced. To further reduce the number of tasks, the scheduler can even remove duplicated inserts of the same vertex in the worklist and perform relaxations only based on the shortest distance label. This technique is the core of the Dijkstra algorithm, which can be implemented by simply replacing the queue-based worklist in Algorithm 3 (FIFO_worklist in line 3 and 17) with a MIN priority queue. Figure 29.b shows an example of how priority scheduling reduces tasks.

7.5 Use Case of Worklist Scheduling: Improving Cache Performance for BFS

Like SSSP, BFS is another graph traversal algorithm for finding the shortest paths. However, BFS operates on non-weighted graphs with constant-length edges, thus the task reduction scheduling from the Dijkstra SSSP cannot be applied. In addition, in the particular implementation of Algorithm 1, vertices are labeled with the indices of their predecessors in their shortest paths instead of the path lengths. Although the Dijkstra optimization is not applicable, assuming hardware caches are used for BFS processing, worklist scheduling can be used to improve data locality and cache performance by prioritizing the tasks that are likely to reuse cached data during their relaxation.

Locality-Aware Scheduling. To introduce locality-awareness to the worklist scheduler, we need an efficient heuristic to identify tasks that are likely to reuse cache data.

One possible approach is to prioritize the task vertices with the largest number of cached neighbors, which was proposed in [2]. While this is an intuitive method, there are three main drawbacks: 1. There exists a delay between the scheduling of a task and the relaxation of its neighbors. As the content of the cache can change constantly, it is possible that the neighbors of the scheduled vertex get evicted during this delay. 2. The scheduler needs direct access to the content of the cache, which requires changes to the cache architecture. 3. The operation of finding the overlapped set between of the cached data and the neighbors of task vertices is complicated.

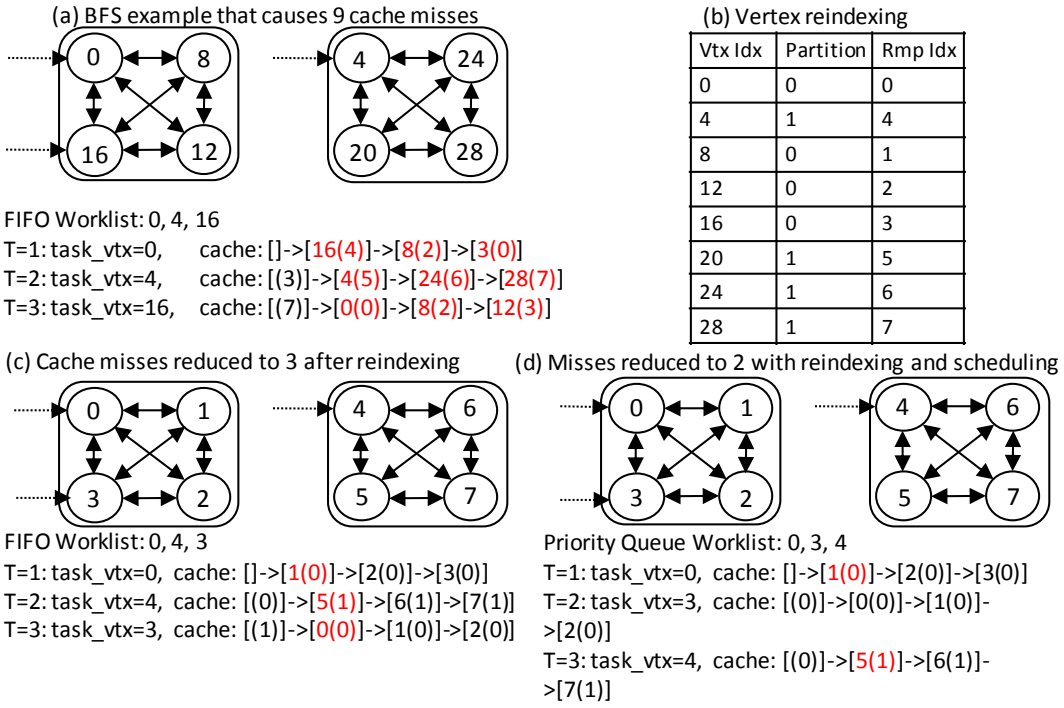


Figure 30. An example of locality-aware scheduling for BFS, assuming the cache stores only 1 cacheline and each cacheline stores 4 vertex labels. Cache misses are highlighted in red.

To avoid the above-mentioned issues, we decide to focus on a different scheduling strategy, which exploits locality within a "community" by dispatching the tasks from the same community in consecutive order. A community is a sub-graph that are densely connected internally. During the process of relaxation, due the clustering nature of a community, processing task vertices of the same community consecutively likely results in better temporal locality. As many real-world networks have been shown to contain community structures [13], this community-based scheduling can be generally applied.

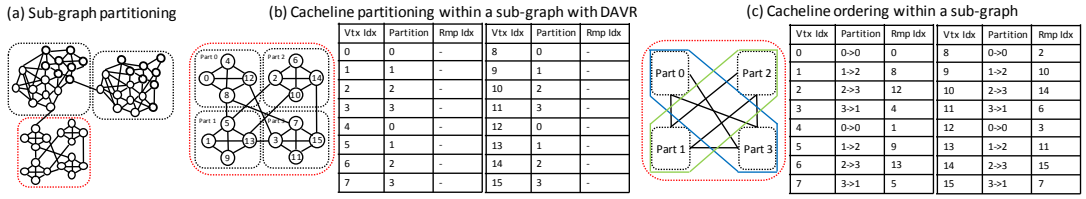
Graph Partitioning and locality-Aware Vertex Indexing. To enable

```
1:   PQ_worklist.enqueue(source);
2:   for(int i=0;i<num_vtx;++i)vtx_array[i].predecessor=-1;
3:   while !PQ_worklist.empty() do
4:    int task_vtx=priority_queue_worklist.dequeue();
5:    int curr_edge_offset=vtx_array[task_vtx].edge_offset;
6:    int num_edge=vtx_array[task_vtx].num_edge;
7:    for num_edge to 0 do
8:     int curr_nbr=edge_array[curr_edge_offset];
9:     int nbr_vtx=vtx_array[curr_nbr];
10:    if vtx_array[nbr_vtx].predecessor==-1 then
11:     vtx_array[nbr_vtx].predecessor=task_vtx;
12:     worklist_temp.enqueue(nbr_vtx);
13:    end if
14:   end for
15:   if PQ_worklist.head same level as worklist_temp.head
      || PQ_worklist.empty() then
16:    insert entries in worklist_temp to PQ_worklist;
17:   end if
18:  end while
```

Algorithm 4. BFS with locality-aware scheduling.

community-based scheduling, we need an efficient method to identify the task vertices belonging to the same community, which can be achieved by applying standard graph

partitioning. Graph partitioning algorithms are effective techniques for analyzing the topological structure of a graph. It finds closely connected vertices and assigns them to the same partition. The partitioning results have been used to remap vertex indices for better data locality [11]. This is done by reassigning the vertices of the same partition with adjacent, continuous indices. This optimization provides spatial locality for the vertices that are likely



Vtx Idx	Partition	Rmp Idx
0	0	-
1	1	-
2	2	-
3	3	-
4	0	-
5	1	-
6	2	-
7	3	-

Vtx Idx	Partition	Rmp Idx
8	0	-
9	1	-
10	2	-
11	3	-
12	0	-
13	1	-
14	2	-
15	3	-

Vtx Idx	Partition	Rmp Idx
0	0->0	0
1	1->2	8
2	2->3	12
3	3->1	4
4	0->0	1
5	1->2	9
6	2->3	13
7	3->1	5

Vtx Idx	Partition	Rmp Idx
8	0->0	2
9	1->2	10
10	2->3	14
11	3->1	6
12	0->0	3
13	1->2	11
14	2->3	15
15	3->1	7

Figure 31. Proposed hierarchical vertex reindexing scheme. First-level sub-graph partitioning shown in (a). Second-level cacheline partitioning shown in (b). (c) shows second-level cacheline ordering remaps vertices from partitions 0 and 3 / 1 and 2 in a

accessed together in the same graph partition. The small example in Figure 30.a, b and c shows how indexing vertices based on their partitions can reduce cache misses. For our purpose of community identification, the remapped indices provide useful hints. After reindexing, the vertices with near indices are more likely to belong to the same community. The scheduler can exploit this observation and prioritize the task vertices with near indices to improve cache performance. In fact, the scheduler can be implemented as a simple priority queue to provide the needed ordering, as shown in Algorithm 4. Figure 30.d illustrates how priority scheduling can be applied to further improve performance of BFS with a re-indexed graph.

We modified version of DAVR to incorporate locality information into the indexing of vertices. An example of our graph pre-processing flow of is shown in Figure 31. At the first-level, we partition the graph to sub-graphs sized to reflect the cache capacity of the system (512 KB in our experiment). All vertices from the same partition will then be assigned with consecutive indices. The same partition will then be assigned with consecutive indices. The second level reindexing happens within partition and involves two steps. In the first step, we applied the adaptive reindexing from the phase 2 of DAVR, which maximizes

the cacheline sharing by adaptive choosing between SONS-based indexing and first-order neighbor based indexing (explained in Chapter 5), as shown in Figure 31.b. In a Intel HARP, each 64-Byte cacheline contains 16 4-Byte indices, thus the SONS partitioning granularity is set to 16.

In the second step, to introduce finer granularity hints for scheduling within each sub-graph partition, we hierarchically apply binary partitioning to order the cachelines based on their connectivity, as shown in Figure 31.c. This step maps topologically adjacent vertices within each sub-graph to physically adjacent cachelines in memory. It provides an incentive for scheduling at cacheline level instead of sub-graph level, as memory performance can be improved by exploiting finer-grain spatial locality. For all of our experiments, we use METIS [10], an open-source graph partitioning tool, to implement the hierarchical reindexing.

7.6 Worklist Scheduling for Other Applications

The application of worklist scheduling is not limited to the above-mentioned examples. Although explained in the context of BFS, the proposed locality-aware scheduling can be generally applied to other graph traversal algorithms. For the same purpose of improving locality, it has been shown that, with a modified processor in which the worklist scheduler can directly access the content of its cache, scheduling effectively improves cache performance for applications including PageRank and Collaborative Filtering, without graph preprocessing [2].

Another use case of priority scheduling is time-stamp sorting. Simulation applications such as Discrete Event Simulation [9] and Asynchronous Variational Integrators [8] from the Galois benchmarks use priority queues to efficiently find tasks with the smallest time-stamp instead of manually managing the ordering of task processing. Furthermore, the fact that major graph processing frameworks such as GraphLab [14] and Galois all feature native support for task prioritization explains its importance and general applicability.

CHAPTER 8. FPGA-ONLY BASELINE PERFORMANCE WITH AND WITHOUT SCHEDULING

In this chapter, we will perform three sets of baseline studies for FPGA only environments. We first explain how to extend the BFS accelerator proposed in part 1 of this thesis to support SSSP. After explaining the evaluation setups in Section 8.2, in Section 8.3, we assess the theoretical benefit of worklist scheduling by assuming a zero-overhead scheduler in simulation. In Section 8.4, we study the practicality of adding a FPGA-based scheduler next to the accelerator.

8.1 Traditional FPGA-Only Accelerators for Graph Traversals

In this section, we will present the baseline FPGA-only BFS and SSSP accelerators that we

will use as the baselines for our later discussion about worklist scheduling. For the study of locality aware scheduling for BFS, we use our optimized accelerator from Part 1 of this thesis as the baseline, with its VC configured to be 512 KB and 4-way associative. For the work reduction scheduling of SSSP, we modified the basic architecture of the proposed BFS accelerator to support Bellmon-Ford SSSP.

The architecture of the proposed BFS accelerator can be extended to support SSSP. Due to the similarity between the two algorithms, only minor changes to the BFS accelerator have to be made. Those changes are:

- Instead of labeling each vertex with the index of its predecessor in the shortest path like BFS, SSSP updates the label of each vertex with the shortest path discovered. To compute the length of a path to its neighbors, we need to know the distance assigned to a task vertex (Line 6). While the distance label can be fetched after the being de-queued from the worklist, the distribution of indices of task vertices are generally irregular and results in low memory performance. To tackle this problem, our accelerator copies the distance data to be stored along with their associated task vertices in the worklist. This allows distance labels to be accessed sequentially along the task vertices.

- An integer adder is added in the SSSP accelerator for computing the length of the newly discovered path to the neighbors of a task vertices, which is not needed in BFS as the vertices are labeled with predecessor indices instead of path lengths.

- In the fifth pipeline stage for determining if a relaxation should happen, instead of

simply checking a vertex has been updated previous, SSSP needs to perform a "less than" comparison to make sure that the length of the new path is shorter than what has been previously assigned to a vertex.

8.2 Evaluation Setup

In following sections, we will present multiple experiments for FGPA-only graph processing and worklist scheduling. The experiments are conducted on top of a shared-memory FPGA platform and a hardware simulator. Except for the studies in section 4.3 and 4.4, the evaluations were based on RTL implementations of the proposed accelerator for an Intel HARP.

Intel HARP features a shared-memory, cache-coherent integration of a Stratix V FPGA and a Xeon E5-2680 v2 processor at 2.8GHz. Its Stratix FPGA has a maximum off-chip memory bandwidth of 7GB/second (bounded by QPI) [3] and 5MB of on-chip dual-ported BRAM. Its Xeon CPU, on the other hand, has a peak memory bandwidth of 49GB/second and 25 MB of 3-level on-chip cache. The Xeon CPU, on the other hand, has a peak memory bandwidth of 49GB/second.

For the other experiments that are difficult to realize on a physical platform, including a limit study presented in section 4.3 and analysis for integrating FPGA-based scheduling with the accelerator in section 4.4, we used a custom in-house simulator. The simulator features cycle-accurate modeling of the accelerator's pipeline stages and off-chip memory system including caches and DRAM.

Our baseline accelerators were implemented using Verilog. We used the same set of networks from previous experiments, shown in Figure 10. The input datasets are all pre-processed with the proposed locality-aware reindexing scheme.

8.3 Limit Study for Worklist Scheduling

Before we start crafting a physical solution for worklist scheduling, it is important to perform a preliminary analysis to understand the potential benefit worklist can bring. For this purpose, We conducted a simulation-based limit study for the locality-aware scheduling for BFS and worklist reduction scheduling for SSSP using our simulator that models our baseline accelerator. We swapped the default FIFO-based worklist with a zero-overhead priority queue which supports instantaneous item insertion and extraction. By ignoring priority scheduling overheads such as the latency of insertion, the performance improvement brought by scheduling represents its maximum limit.

BFS. We first evaluated locality-aware scheduling for BFS with the non-scheduling accelerator as the baseline. We compared their cache hit rate and computation throughput, and the results are shown in Figure 32.a . We observed a peak 2.73x improvement in cache hit rate and 2.09x improvement in performance. The minimum improvement 4% with 2% for cache hit rate and throughput, respectively. This wide difference is caused by the topological differences of the graphs. The graphs with clearly-separable, densely interconnected community structures are more likely to benefit from the scheduling. On average, the cache hit-rate is improved by 35% and the execution time is improved by 28.6%.

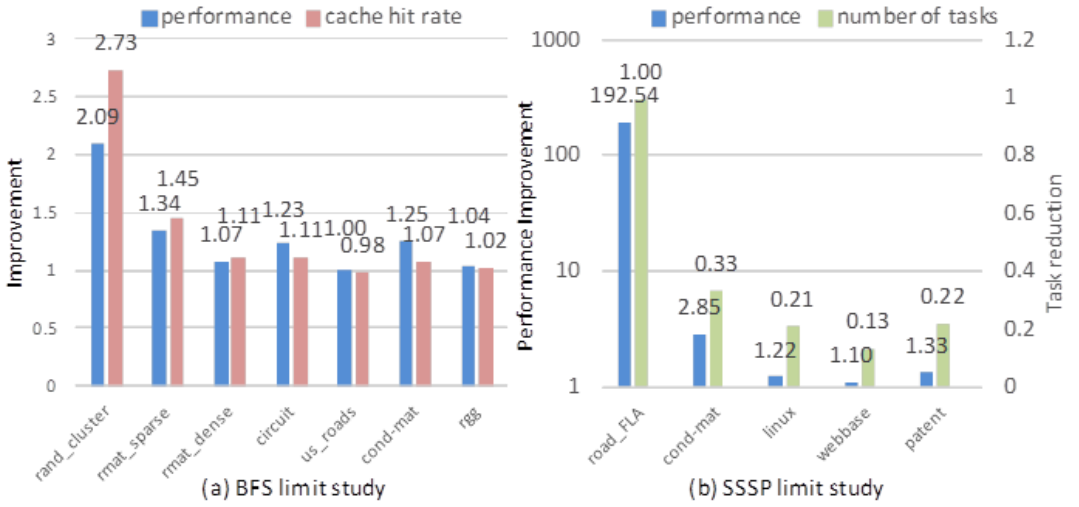


Figure 32. Comparison of the baseline BFS accelerator with the Galois software BFS on HARP.

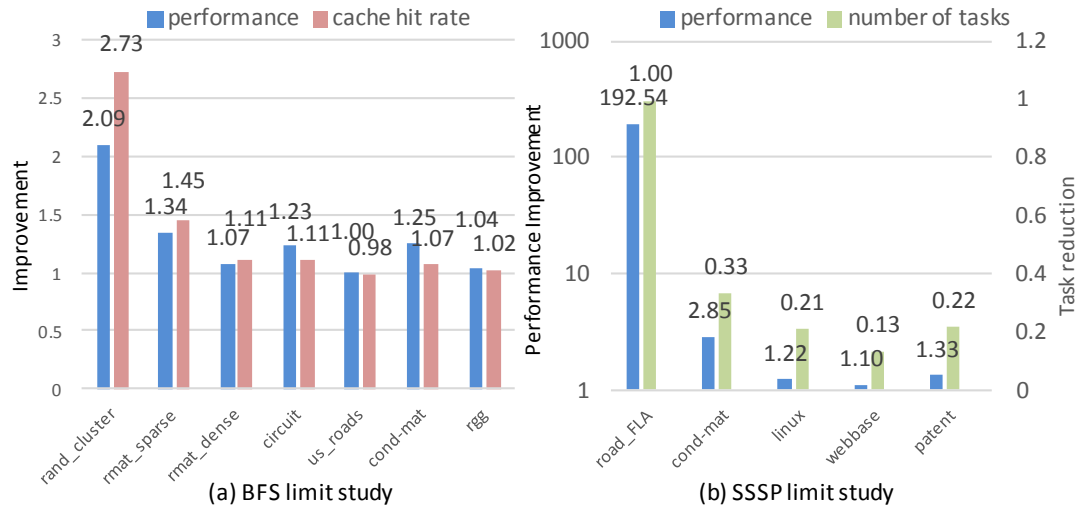


Figure 33. Limit studies for the potential performance improvement of worklist scheduling.

SSSP. For SSSP, the scheduler prioritizes task vertices with smaller distance

labels. The reduction for number of active vertices pushed to the worklist is shown in Figure 32.b. The work reduction shows a wide variation from 13% to over 99% depending on the distribution of edge lengths and graph connectivity. In general, the graphs with wide variation in edge lengths and high vertex degrees tend to have larger opportunity. As the fewer of worklist inserts implies shorter processing time, the performance improvement is strongly correlated to the number of reduced works.

8.4 Adding Hardware Priority Queue to the Baseline FPGA-Only Accelerator

The limit study in section 4.2 reveals promising potential in worklist scheduling for improving performance as if the priority-queue worklist has no performance and resource overhead. To understand this overhead for a hardware priority queue implementation, we modified our simulator to model the BRAM-tree, a binary-heap-based priority queue architecture from [7].

The original BRAM tree requires the entire priority-queue to be implemented using on-chip BRAM, which limits the capacity and prevents it from supporting the processing of large graphs. To resolve this issue, we modified the design to support the spilling of higher levels of the binary heap to off-chip DRAM when the tree size exceeds BRAM's storage capacity. This design essentially uses BRAM as cache to buffer the items at the lower levels of a tree, thus its performance would be dependent of the size of the BRAM allocated for caching, as spilling to off-chip DRAM incurs long latency.

We evaluated this design with different priority-queue cache sizes using the RMAT_sparse dataset for BFS. The result is shown in Figure 33.a. The key observation is that a large amount of BRAM (enough to buffer 50% of the vertices) is needed as the heap cache to offset the DRAM spilling overhead and breaks even in performance with the baseline BFS accelerator without scheduling. With less than 1MB heap cache, the scheduling overhead of accessing the priority-queue in memory actually lowers performance compared to the baseline. Similar results were observed when running SSSP with the Linux_call graph, as shown in Figure 33.b. This creates a problem in resource contention as the baseline accelerator also need large amount of on-chip storage to deliver competitive performance. More BRAM used for heap buffering implies less on-chip storage for aching graph vertices. This can impact the performance of the accelerator negatively, as the vertex cache hit rate lowers due to smaller capacity.

CHAPTER 9. PROCESSOR-ASSISTED SCHEDULING ON SHARED-MEMORY FPGA PLATFORMS

Recently, platforms offering coherent memory access for its the host processor and FPGA fabric have become available. Intel-Altera HARP is one such example targeting the server market. HARP integrates the two devices using its QPI interconnect, which enables the possibility of fine-grain, heterogeneous collaboration. To get around the problem of BRAM contention for integrating worklist scheduling shown in the last section, in this work, we exploit this opportunity of collaboration for worklist scheduling. Specifically, our proposed processing paradigm, Processor-Assisted Scheduling (PAS), dynamically offloads the task of

worklist scheduling to the host processor.

9.1 Motivations for PAS

There have been a few prior attempts to exploit the close integration of the processor and the FPGA for collaborative graph processing, e.g., [20] [24]. In both works, the main graph processing routines are mapped to the processor. This is very different from our approach.

In section 4.2, we learned that the state-of-the-art software implementation of a graph algorithm under-utilizes processor resources while consuming a large amount of power as it relies on multithreading, multi-core processing to extract MLP. In contrast to prior works, our proposed solution, PAS, avoid this problem by mapping the main routine of relaxation to an efficient FPGA accelerator, which delivers performance comparable to the processor while consuming much lower power. Furthermore, PAS allows the accelerator to enjoy the benefit of worklist scheduling without having a priority-queue on the FPGA-fabric, which contents for BRAM resources with the accelerator's cache. PAS offloads priority scheduling to a light-weight software scheduler that consumes a negligible load on the processor. By assigning the processor and the FPGA with the tasks they are good at, PAS allows them to deliver better performance together than each can achieve alone.

To summarize, the benefits of PAS are as follows:

(1) No BRAM contention. PAS does not integrate priority queue with the FPGA accelerator, which requires large number of on-chip buffering of latency hiding. Thus accelerator has full control over the utilization of BRAM for graph data caching.

(2) Lower memory latency. Modern processors have large cache with advance features for minimizing misses. For Intel HARP, the 25MB smart cache of its Xeon is significantly larger than its 6MB BRAM of its Stratix V FPGA, which can help the memory latency sensitive heap operations on the large in-DRAM priority queue. On an Intel HARP, the host processor has significant lower memory latency compared to its FPGA, as the memory requests does not have to traverse through the QPI interconnect like the FPGA does [3]. This provides better priority-queue performance when traversing the non-cached levels of the tree.

(3) Negligible processor load. The worklist scheduling routine often stays idle as the insertion of new tasks happens only once in a while. If we were to implement the scheduler using software, this property implies minimal interference to the other applications. We will demonstrate this property later in section 6, which makes PAS an ideal technique for servers.

(4) Ease of implementation. PAS requires only minimal changes to the baseline accelerators, and implementing the scheduling routine as software is fundamentally simpler in comparison to designing it in RTL. In addition to the specific scheduling we implemented, it becomes possible to readily try from software other equally light-weight scheduling alternative [16]. Furthermore, a small amount of priority inversion in scheduling will not impact the correctness of the program while still preserving most of the scheduling benefit. This helps us to keep the design of the scheduler simple without the need to worrying about always maintaining the perfect order of tasks.

9.2 Implementation of PAS

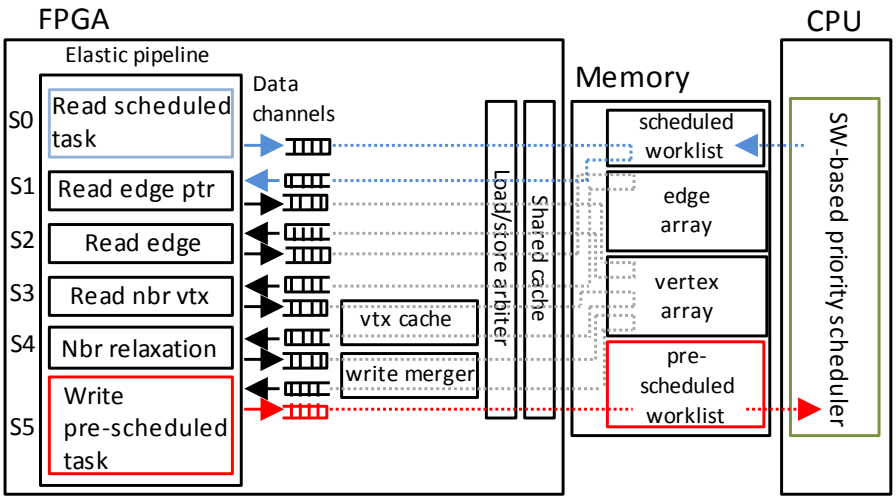


Figure 34. Block diagram of Processor-Assisted Scheduling. Changes to the baseline accelerator are highlighted in colors.

To offload the task of worklist scheduling to the processor, we made several changes to the worklist writing (s5) and reading stages(s0) of the base BFS and SSSP accelerators. The new system architecture is shown in Figure 38. We split the worklist to two data structures: a pre-scheduled queue and a scheduled queue, and the software priority scheduler is shown in Algorithm 5.

The pre-scheduled queue is a circular FIFO with the accelerator being the producer and the software scheduler being the consumer. When new works are generated because of successful relaxation, they are inserted to the head position of pre-schedule queue sequentially, just like the original worklist. The software scheduler periodically probes into the tail position of the queue to verify if new works have arrived by checking a valid flag tagged to each cacheline in the pre-schedule queue. This tag is labelled by the accelerator

```
1:   while accelerator is still running do
2:    sleep(PROBE_PERIOD);
3:    WL_header header=presch_WL[0];
4:    if header.valid then
5:     insert(presch_WL[1], header.numTasks, PQ);
6:     presch_WL=presch_WL+CLINE_SIZE;
7:    end if
8:    if (header.acc_idle()||(sch_timer=SCH_PERIOD))&&(!PQ.empty())
      then
9:     insert(PQ, MIN(PQ.size(), CLINE_SIZE-1), sch_WL);
10:    sch_WL=sch_WL+CLINE_SIZE;
11:    sch_timer=0;
12:   else
13:    sch_timer=sch_timer+1;
14:   end if
15:  end while
```

Algorithm 5. Software priority scheduler in PAS.

when writing the new works to memory. When valid works are received, the software scheduler insert those new tasks in to a priority queue, which allows efficient identification of high priority tasks.

The scheduled queue is another circular FIFO with the software scheduler being producer and the accelerator being the consumer. When the scheduler predicts that the accelerator need new work to keep its pipeline busy, the high priority works will be extracted from the priority queue and then inserted to the scheduled queue, awaiting to be processed by the accelerator.

The timing of writing to the scheduled queue is important to performance. Writing

too frequently causes constant draining of the priority queue without letting high-priority tasks to "bypass" to the front, which defeats the purpose of scheduling. Writing too infrequently can potentially leave the accelerator in starvation for tasks when the pipeline is idle. To tackle this problem, we introduce awareness to the idleness of the accelerator on top of a simple periodic scheduling protocol. In this protocol, the software schedules new tasks either periodically or when the accelerator signals its idleness. In the case of Intel HARP, which writes to memory at 64- Byte granularity, an idleness flag is tagged to every non-scheduled work cacheline by the accelerator. The 1-bit idleness flag is set when the edge pointer fetching stage (S2) for active vertices remains idle for multiple cycles due to lack of work, implying it needs more scheduled tasks from the software scheduler.

9.3 Evaluation: Performance

We evaluated PAS on a physical Intel HARP for both BFS and SSSP (not simulation). As PAS introduces extra memory access overhead for heterogeneous communication between the processor and the accelerator, we decided to first assess if PAS remains practical on HARP by comparing it with the result in the no-overhead ideal case obtained using simulation in section 8.3. For BFS, this comparison is shown in Figure 11.a. On average, PAS improves cache hit rate by 29% and throughput improvement by 20%. In comparison to the limit study without scheduling overhead, the differences of the improvements are only around 5%, implying PAS is capable of retaining most of the theoretical benefit of scheduling even with the overhead from a physical platform.

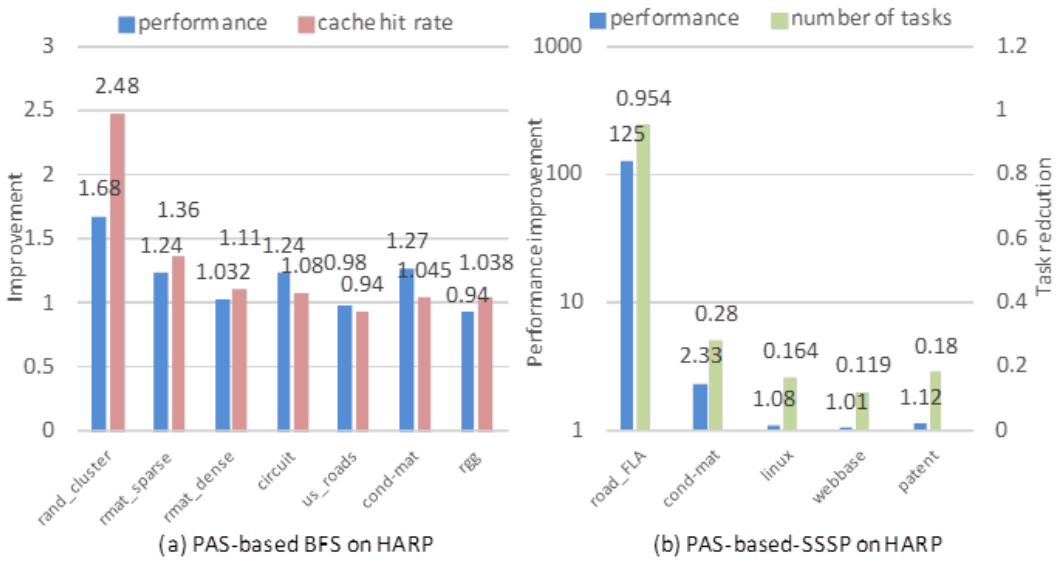


Figure 35. Results for evaluation of PAS.

For SSSP, the result is similar to BFS, as shown in Figure 11.b. We observed the overhead for SSSP to be slightly higher, this is likely due to the slightly more complex scheduling routine, which is based on the distance labels of vertices instead of just the indices of vertices like BFS. Overall, the difference between the limit study and the physical implementation of PAS-based SSSP is around 15%.

While at a glance, the scheduling overhead might be surprisingly small as the scheduler's probing to the unscheduled worklist in memory might appear expensive, there is a simple explanation. The QPI interconnect of Intel HARP maintains coherency at the CPU's last-level cache (LLC). This means when the accelerator updates the pre-scheduled queue, the new tasks would be brought to the last level cache directly for the processor to retrieve. This eliminates the long-latency off-chip memory accesses during task probing by exploiting

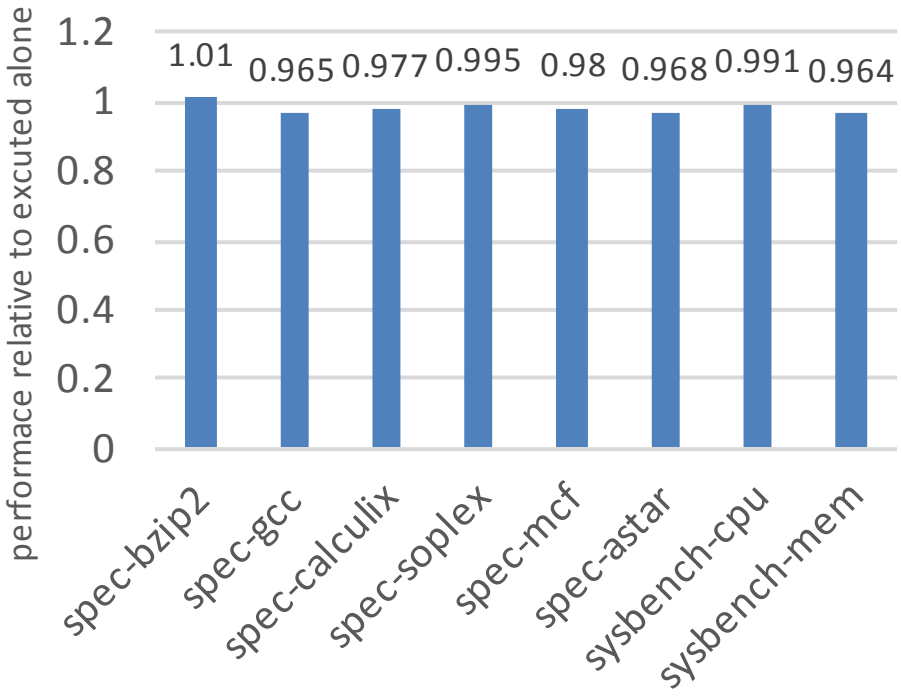


Figure 36. Performance of standard CPU benchmarks running in parallel with PAS relative to running alone.

a special form of locality between the accelerator and the scheduler.

9.4 Evaluation: Scheduler Load

To understand the software scheduler's processor resource consumption, we characterized it using Intel PCM [23] and observed extremely low 5% core utilization and near-idle 15.7 Watt power consumption. We then assessed the scheduler's potential interference to other concurrent programs on the processor by dispatching PAS in parallel with as set of selected SPEC2006 [4] and Sysbench [1] benchmarks. All test cases are configured to 20 threads to

fully exploit HARP's 10-core hyper-threading-enabled Xeon. The results, shown in Figure 40, indicate that the PAS scheduler causes no observable interference to the benchmarks. This characteristic implies PAS can be a good processing paradigm for servers, as the light-weight scheduler is unlikely to impact the services to the other server clients.

9.5 Summary of PAS

In this second part of this thesis, we presented our exploration of the worklist-scheduling optimization for graph algorithms on a shared-memory FPGA platform, which led to Processor-Assisted Scheduling (PAS), a new heterogeneous processing scheme that allows the processor to dynamically assist the FPGA accelerator with finding the high-priority tasks. In particular, we first developed an accelerator for BFS and SSSP. With this as the baseline, we analyzed the effectiveness of integrating priority-queue-based scheduling on FPGA and discovered its heavy consumption of BRAM and potential resource contention with the baseline accelerator.

To solve this problem, we developed PAS, which exploits the close integration of the processor and the accelerator on a shared-memory platform by offloading the task of worklist scheduling to the processor. In our evaluation, we case-studied locality-aware scheduling for BFS, a new scheduling heuristic for improving cache performance that we proposed, and the task reduction scheduling of the Dijkstra-based SSSP. Benefited from the processor's superior memory sub-system, the software scheduler delivers timely scheduling service for the accelerator, as the evaluation results show that PAS is able to obtain over 90% and 80%

of the peak scheduling performance benefit for BFS and SSSP, respectively. Additionally, the light-weight scheduler consumes minimal processor resources and does not interfere other concurrent applications. This approach allows the two devices to focus on the tasks they are good at and collaborative.

CHAPTER 10. CONCLUDING REMARKS

To re-iterate, the thesis of this dissertation is FPGA-based graph processing is capable of delivering not only outstanding energy efficiency but also competitive performance on a shared-memory system that lacks off-chip memory bandwidth. The key to achieve this is, as oppose to pursing solely on maximizing parallel off-chip memory accesses like processor-based solutions do, focusing on specializing the solution design to exploit the hardware features that are unique to the FPGA platform. This allows us to realize the optimizations for graph processing that were previously overlooked in the hardware domain.

Specifically This work demonstrates this principle with two works: 1. maximizing on-chip data reuse and pipeline utilization for an FPGA-only accelerator by incorporating application-specific knowledge, and 2. exploiting the close CPU-processor integration on a shared-memory platform to realize worklist scheduling, which is an important algorithmic optimization that can effectively improve data locality for BFS and reduce the number of

works for SSSP but has been traditionally ineffective to implement on FPGAs due to design complexity and heavy BRAM consumption.

Reducing Dependency on Off-Chip Memory Performance. In the first part of this thesis, the work focuses on maximizing the performance of the a BFS FPGA accelerator that follows the traditional principle to maximizing MLP. First, we discovered that the processing pipeline is severely underutilized. We found that the underutilization is caused primarily by the stall due to dependency in data accesses in different pipeline stages, which can be fully resolved by using data forwarding from a write buffer. Next, we observed that prior accelerator designs often lacks features to enable effective on-chip data reuse as the irregularity of graph processing complicates the design of data buffering. We believe this is performance-critical as main-stream off-the-shelf FPGA systems often lack off-chip bandwidth but feature abundant on-chip storage devices. To tackle this problem, we propose a partitioned data caching scheme that isolates the buffering of different data types. In addition, we developed an enhanced data preprocessing optimization, DAVR, to improve the locality of the vertex array. In combination, they significantly improved the cache hit rate compared to the standard approach of using a unified cache. Finally, we identified the problem of inefficient write bandwidth utilization and proposed a write request coalescing optimization that combines multiple vertex updates to a single transaction to improve the utilization of write memory bandwidth. With all optimization incorporated, the accelerator was capable of delivering comparable performance to a state-of-the-art software implementation on a high-end CPU with superior performance while consuming merely 7% of the power.

Exploiting Complex Algorithmic Optimization through FPGA-CPU Collaboration. In the second part of the study, we explored worklist scheduling, a widely used optimization for various worklist-based graph algorithms, on a shared-memory CPU-FPGA platform. As this is rarely applied to past accelerator designs, we first looked into the reason. We discovered that, in addition to implementation complexity, implementing a priority queue used for scheduling on FPGA consumes a large amount of on-chip storage, causing resource contention with an accelerator that uses BRAM for data caching. To tackle this problem, we proposed PAS for share-memory CPU-FPGA systems, which exploits the close integration of the two devices to dynamically offload the scheduling task to a software scheduler running on the processor. In our evaluation, we case-studied locality-aware scheduling for BFS, a new scheduling heuristic for improving cache performance that we proposed, and the task reduction scheduling of the Dijkstra-based SSSP. Benefited from the processor's superior memory sub-system, the software scheduler delivers timely scheduling service for the accelerator, as the evaluation results show that PAS is able to obtain over 90% and 80% of the peak scheduling performance benefit for BFS and SSSP, respectively. Additionally, the light-weight scheduler consumes minimal processor load and interfere minimally with other concurrent applications.

10.1 Limitations

We discuss some of the limitations of the current work and present possible solutions in the future work.

1. Preprocessing Overhead. In the proposed graph preprocessing optimizing DAVR, the time spent for partitioning the input data can be orders of magnitude longer than the execution of the application. While this cost can be potentially amortized over different applications, applying such optimization for a single application can be counterproductive.

2. Lack Support of Morph Algorithms. Morph algorithms are a class of graph computations that modifies the structure of the input graphs. Examples include mesh refinement [44] and Survey Propagation [45]. Many of the proposed optimizations might not be applicable to morph algorithms due to various reasons. For instance, DAVR remaps vertices with the assumption of a fixed graph topology. The modification to the connectivity of vertices breaks this assumption and compromises its effectiveness. Morph algorithms append vertices to the input graph, but none of the commercial platforms available today support dynamic memory allocation on the FPGA fabric, which makes accelerator implementation difficult as it requires predicting the extra memory space needed for storing the extra vertices and edges before launching the accelerator.

3. Assumption of Asymmetry in Memory Performance and Close CPU-FPGA Integration. PAS is designed based on the assumption that the processor of the shared-memory system features a superior memory subsystem compared to the FPGA fabric to justify the overhead of scheduling offloading. While this is true for most existing platforms [6] [41]. For platforms of which

the processor has no advantage in memory performance [42], the effectiveness of PAS is unclear.

10.2 Future Work

Processor Assists for Other Algorithmic Optimizations. In this thesis, we explored graph processing on a shared-memory CPU-FPGA platform and discovered an effective model for a synergic collaboration between the two devices. While we focused on a worklist scheduling, there are other important algorithmic optimizations that can potentially benefit from this type of heterogeneous collaboration. One example is that Direction-Optimized processing for BFS (DOB) [31]. DOB delivers better performance by switching between forward and backward BFS based on the number of visited vertices in the input graph. DOB was traditionally overly complicated to be ported to FPGA. The first attempt to FPGA-based DOB appeared in 2018 [32]. While it delivers impressive performance improvement over the baseline BFS, the proposed solution uses dedicated hardware for maintaining a list of unvisited vertices, which consumes FPGA resources and additional off-chip memory bandwidth. Our proposed heterogeneous scheme can be applied to take over the task of managing this list of unvisited vertices just like it helps BFS and SSSP to schedule the worklist. This can lower implementation complexity and potentially improve performance as the tracking of unvisited vertices is now done by the processor and consumes no precious CPU memory resources.

PAS for Other Scheduling Protocols. While the proposed heterogeneous strategy to priority scheduling is generally usable across different applications, in the thesis, we focus on case studies two relatively-simple algorithms, BFS and SSSP, as the proxy for demonstrating its effectiveness. Another direction for future work is to extend the coverage to support other types worklist scheduling optimizations. For instance, it would be interesting to test PAS with more complicated applications that need scheduling e.g., DES, EBS. And our proposed locality-aware scheduling protocol can be further tested with other worklist-based applications to verify its effectiveness in improving locality beyond graph traversals.

Generalizing PAS for Other Shared Memory Platforms. While we used only an Intel HARP for evaluating PAS, there are actually several other shared-memory systems which can be used for implementing PAS. Examples include Xilinx Zynq, IBM CAPI and the new Intel HARP 2. As many of them offers different hardware features that are not found on HARP, it would be interesting to investigate how to incorporate these features to further improve PAS. For instance, Zynq offers a dedicated AXI based channel for direct signaling between the processor and the FPGA fabric, which offers the opportunity to eliminate the software scheduler's iterative memory probing for finding out if new tasks have become available to be scheduled.

Abstracting PAS for Better Usability. Another interesting direction of research would be developing a general abstraction of PAS. While many applications can benefit from scheduling tasks in their worklists, their difference in algorithmic characteristics can

affect the tuning of the scheduler. For instance, the performance of the Dijkstra SSSP is often insensitive to occasional priority inversion. This fact can be exploited to reduce the amount of communication between the scheduler and the accelerator for platforms that have limited processor-FPGA bandwidth. A domain-specific framework can hide the complicated details of hardware tuning by exposing only a "knob" that allows the users to specify the application's sensitivity to priority inversion and automatically optimizes the frequency of scheduling based on the input sensitivity.

References

[1] 2013. Sysbench. Retrieved October 1, 2018 from https://wiki.gentoo.org/wiki/Sysbench.

[2] Nathan Beckmann and Daniel Sanchez. 2017. Cache-guided Scheduling Exploiting Caches to Maximize Locality in Graph Processing. In 1st International Workshop on Architectures for Graph Processing.

[3] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In Proceedings of the 53rd Annual Design Automation Conference (DAC '16). ACM, New York, NY, USA, Article 109, 6 pages. https://doi.org/10.1145/2897937.2897972

[4] Standard Performance Evaluation Corporation. 2006. SPEC CPU 2006. Retrieved October 1, 2018 from https://www.spec.org/cpu2006/.

[5] E.W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. Numer. Math. 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[6] PK Gupta. 2015. Xeon+FPGA Platform for the Data Center. Retrieved October 1, 2018 from https://www.archive.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[7] M. Huang, K. Lim, and J. Cong. 2014. A scalable, high-performance customized priority queue. In 2014 24th International Conference on Field Programmable Logic and Applications (FPL). 1–4. https://doi.org/10.1109/FPL.2014.6927413

[8] Universtiy of Texas at Austin ISS. 2013. Asynchronous Variational Integrator. Retrieved October 1, 2018 from http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/asynchronous_variational_integrators

[9] Universtiy of Texas at Austin ISS. 2013. Descrete Event Simulator. Retrieved October 1, 2018 from http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/discrete_event_simulation

[10] George Karypis and Vipin Kumar. 1997. METISaĂŤA Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices. (01 1997).

[11] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. 2018. Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18). ACM, New York, NY, USA, 239–248. https://doi.org/10.1145/3174243.3174260

[12] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. 2009. Lonestar: A suite of parallel irregular programs. In 2009 IEEE International Symposium on Performance Analysis of Systems and Software. 65–76. https://doi.org/10.1109/ISPASS.2009.4919639

[13] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2008.Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. CoRR abs/0810.1355 (2008). arXiv:0810.1355 http://arxiv.org/abs/0810.1355

[14] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A New Framework for Parallel Machine Learning. CoRR abs/1006.4990 (2010). http://arxiv.org/abs/1006.4990

[15] Batul J. Mirza, Benjamin J. Keller, and Naren Ramakrishnan. 2003. Studying Recommendation Algorithms by Graph Analysis. J. Intell. Inf. Syst. 20, 2 (March 2003), 131–160. https://doi.org/10.1023/A:1021819901281

[16] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In The 51st Annual IEEE/ACM International Symposium on Microarchitecture.

[17] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Mendez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. SIGPLAN Not. 46, 6 (June 2011), 12–25. https://doi.org/10.1145/1993316.1993501

[18] Ali Shokoufandeh and Sven Dickinson. 2002. Theoretical Aspects of Computer Science. Springer-Verlag New York, Inc., New York, NY, USA, Chapter Graphtheoretical Methods in Computer Vision, 148–174. http://dl.acm.org/citation.cfm?id=644608.644614

[19] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. Proc. VLDB Endow. 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[20] Y. Umuroglu, D. Morrison, and M. Jahre. 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). 1–8. https: //doi.org/10.1109/FPL.2015.7293939

[21] T. Wang, Y. Chen, Z. Zhang, T. Xu, L. Jin, P. Hui, B. Deng, and X. Li. 2011. Understanding Graph Sampling Algorithms for Social Network Analysis. In 2011 31st International Conference on Distributed Computing Systems Workshops.123–128. https://doi.org/10.1109/ICDCSW.2011.34

[22] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-performance Graph Processing Library on the GPU.

SIGPLAN Not. 51, 8, Article 11 (Feb. 2016), 12 pages. https://doi.org/10.1145/3016078.2851145

[23] Thomas Willhalm, Roman Dementiev, and Patrick Fay. 2012. Intel Performance Counter Monitor - A better way to measure CPU utilization. Retrieved October 1, 2018 from https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[24] S. Zhou and V. K. Prasanna. 2017. Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform. In 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). 137–144. https://doi.org/10.1109/SBAC-PAD.2017.25

[25] James Demmel. 1999. CS 267 Applications of Parallel Computers Lecture 2: Memory Hierarchies and Optimizing Matrix Multiplication. Retrieved October 1, 2018 from http://web.cs.ucdavis.edu/~bai/ECS231/optmatmul.pdf

[26] Xilinx, Inc.. 2016. 7 Series FPGA Overview. Retrieved October 1, 2018 from http://www.eng.ucy.ac.cy/theocharides/Courses/ECE408/7%20Series%20FPGA%20Overview.pdf

[27] Altera Inc.. 2006. FPGA Architecture White Paper. Retrieved October 1, 2018 from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf

[28] Intel Inc.. 2008. Intel QuickPath Architecture. Retrieved October 1, 2018 from https://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf

[29] S. Zhou, V. Prasanna, "Accelerating graph analytics on CPU-FPGA heterogeneous platform", *Proc. IEEE Int. Symp. Comput. Architecture High-Perform. Comput.*, 2017.

[30] Intel Inc.. 2008. Intel QuickPath Architecture. Retrieved October 1, 2018 from https://www.intel.com/pressroom/archive/reference/whitepaper_QuickPath.pdf

[31] Basel Bani-Ismail, Ghassan Kanaan. 2012. Comparing Different Sparse Matrix Storage Structures as Index Structure for Arabic Text Collection. In International Journal of Information Retrieval Research. Volume 2 Issue 2. 52-67.

[32] Jialiang Zhang , Jing Li, Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform, Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25-27, 2018, Monterey, CALIFORNIA, USA [doi>10.1145/3174243.3174245]

[33] Micron Inc.. 2011. Micron ACS Tech Brief: Achieve High-Performance Computing in Three Easy Steps. Retrieved October 1, 2018 from https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/ac-series-hpc-modules/ac-510

[34] Micron Inc.. 2017. The Convey HC-2 ™ Computer Architectural Overview. Retrieved October 1, 2018 from https://www.micron.com/~/media/documents/products/white-paper/wp_convey_hc2_architectual_overview.pdf

[35] Scott Beamer. Understanding and Improving Graph Algorithm Performance. Retrieved October 1, 2018 from http://www.scottbeamer.net/pubs/beamer-thesis.pdf

[36] Young-kyu Choi , Jason Cong , Zhenman Fang , Yuchen Hao , Glenn Reinman , Peng Wei, A quantitative analysis on microarchitectures of modern CPU-FPGA platforms, Proceedings of the 53rd Annual Design Automation Conference, p.1-6, June 05-09, 2016, Austin, Texas [doi>10.1145/2897937.2897972]

[37] Joe Chang 2018. Memory Latency II. Retrieved October 1, 2018 from http://www.qdpma.com/ServerSystems/MemoryLatencyII.html

[38] Maysam Lavasani , Hari Angepat , Derek Chiou, An FPGA-based In-Line Accelerator for Memcached, IEEE Computer Architecture Letters, v.13 n.2, p.57-60, July 2014 [doi>10.1109/L-CA.2013.17]

[39] Xiaoyu Ma , Dan Zhang , Derek Chiou, FPGA-Accelerated Transactional Execution of Graph Workloads, Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 22-24, 2017, Monterey, California, USA [doi>10.1145/3020078.3021743]

[40] Nvidia Inc. 2009. NVIDIA OpenCL Best Practices Guide Version 1.0. Retrieved October 1, 2018 from

https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf

[41] IBM Inc. 2018. IBM Power System AC922 Introduction and Technical Overview. Retrieved October 1, 2018 from https://www.redbooks.ibm.com/redpapers/pdfs/redp5472.pdf

[42] Xilinx Inc. 2018. Zynq-7000 SoC Data Sheet: Overview. Retrieved October 1, 2018 from https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf

[43] Luis Carlos Maria Remis. 2016. Breadth-First Search for Social Network Graphs on Heteerogeneous Platforms. Retrieved October 1, 2018 from https://core.ac.uk/download/pdf/158313623.pdf

[44] Universtiy of Texas at Austin ISS. 2013. Delaunay Mesh Refinement. Retrieved October 1, 2018 from http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/delaunay_mesh_refinement

[45] A. Braunstein , M. Mézard , R. Zecchina, Survey propagation: An algorithm for satisfiability, Random Structures & Algorithms, v.27 n.2, p.201-226, September 2005