Using Workload Characterization to Guide High Performance Graph Processing

Mohamed W. Hassan

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Engineering

Peter M. Athanas, Chair

Yasser Y. Hanafy

Michael Hsiao

Paul Plassmann

Hoda Eldardiry

May 12, 2021

Blacksburg, Virginia

Keywords: Configurable Computing, Graph Processing, Heterogeneous Systems, High Performance Computing

Using Workload Characterization to Guide High Performance Graph Processing

Mohamed W. Hassan

(ABSTRACT)

Graph analytics represent an important application domain widely used in many fields such as web graphs, social networks, and Bayesian networks. The sheer size of the graph data sets combined with the irregular nature of the underlying problem pose a significant challenge for performance, scalability, and power efficiency of graph processing. With the exponential growth of the size of graph datasets, there is an ever-growing need for faster more power efficient graph solvers. The computational needs of graph processing can take advantage of the FPGAs' power efficiency and customizable architecture paired with CPUs' general purpose processing power and sophisticated cache policies. CPU-FPGA hybrid systems have the potential for supporting performant and scalable graph solvers if both devices can work coherently to make up for each other's deficits.

This study aims to optimize graph processing on heterogeneous systems through interdisciplinary research that would impact both ***the graph processing community***, and the ***FPGA/heterogeneous computing community***. On one hand, this research explores how to harness the computational power of FPGAs and how to cooperatively work in a CPU-FPGA hybrid system. On the other hand, graph applications have a data-driven execution profile; hence, this study explores how to take advantage of information about the graph input properties to optimize the performance of graph solvers.

The introduction of High Level Synthesis (HLS) tools allowed FPGAs to be accessible to the masses but they are yet to be performant and efficient, especially in the case of irregular graph applications. Therefore, this dissertation proposes automated frameworks to help

integrate FPGAs into mainstream computing. This is achieved by first exploring the optimization space of HLS-FPGA designs, then devising a domain-specific performance model that is used to build an automated framework to guide the optimization process. Moreover, the architectural strengths of both CPUs and FPGAs are exploited to maximize graph processing performance via an automated framework for workload distribution on the available hardware resources.

Using Workload Characterization to Guide High Performance Graph Processing

Mohamed W. Hassan

(GENERAL AUDIENCE ABSTRACT)

Graph processing is a very important application domain, which is emphasized by the fact that many real-world problems can be represented as graph applications. For instance, looking at the internet, web pages can be represented as the graph vertices while hyper links between them represent the edges. Analyzing these types of graphs is used for web search engines, ranking websites, and network analysis among other uses. However, graph processing is computationally demanding and very challenging to optimize. This is due to the irregular nature of graph problems, which can be characterized by frequent indirect memory accesses. Such a memory access pattern is dependent on the data input and impossible to predict, which renders CPUs' sophisticated caching policies useless to performance.

With the rise of heterogeneous computing that enabled using hardware accelerators, a new research area was born, attempting to maximize performance by utilizing the available hardware devices in a heterogeneous ecosystem. This dissertation aims to improve the efficiency of utilizing such heterogeneous systems when targeting graph applications. More specifically, this research focuses on the collaboration of CPUs and FPGAs (Field Programmable Gate Arrays) in a CPU-FPGA hybrid system. Innovative ideas are presented to exploit the strengths of each available device in such a heterogeneous system, as well as addressing some of the inherent challenges of graph processing. Automated frameworks are introduced to efficiently utilize the FPGA devices, in addition to distributing and scheduling the workload across multiple devices to maximize the performance of graph applications.

Dedication

I dedicate this dissertation to my mother "Rasmia" and my wife "Rana" whose words of encouragement pushed me forward all the way to the finish line. Their unwavering support was the pillar that helped me throughout this journey.

I also dedicate all my work to my late father, Prof. Wasfy Abouelmakarem, who inspired me to walk this path. His knowledge and wisdom motivated me throughout my entire life.

Acknowledgments

I am deeply grateful for my advisor, Prof. Peter Athanas, for giving me the chance to be a member of the Configurable Computing Machines (CCM) lab and for his unwavering support and belief in me. He provided me with valuable guidance and continuous support throughout my PhD. I would like to express my sincere gratitude to Prof. Yasser Hanafy for mentoring and supporting me by shining a light when tackling a new research area. I would also like to thank Prof. Wu-chun Feng for his assistance and insightful comments and suggestions that allowed me to contribute in multiple research areas. I would also like to thank Prof. Michael Hsiao, Prof. Paul Plassmann, and Prof. Hoda Eldardiry for their guidance that helped shape this dissertation.

I would like to offer my special thanks to Dr. Scott Pakin for his mentoring and guidance during and after my internship at Los Alamos National Lab. Introducing me to the field of quantum computing turned out to be one of the best research projects I have ever worked on. I would like to extend my sincere thanks to Dr. Andrew Schmidt who provided me with an amazing working environment that helped bring out the best in me. He also taught me a well-structured way to conduct my research which was exactly what I needed to push me over the top.

I would like to express my gratitude to the Intel Innovator program, especially Sujata Tiberwala, for granting me access to Intel's Devcloud cluster and accommodating my hardware and software requests to support my research. They provided me with state of the are hardware and the latest software that was very helpful in completing my research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many real-world problems can be presented as large-scale graphs with millions of vertices and billions of edges, such as social networks, biological interactions, and web graphs [37, 63]. Graph analytics is an area of study used to extract useful information from these huge datasets as depicted in Figure 1.1. For instance, in web graphs, web pages represent the graph vertices while hyper links between them represent the edges. Analyzing these types of graphs is used for web search engines, ranking websites, and network analysis among other uses [28]. The enormous size of these datasets compounded by the irregularity of the underlying problem exhibit substantial challenges to graph processing.

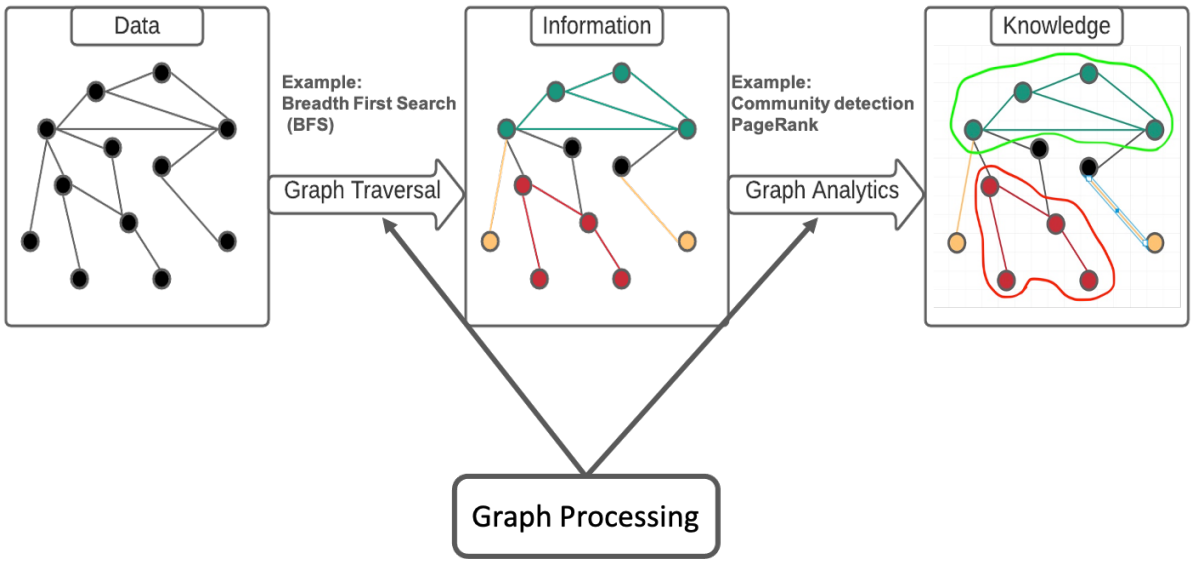


Figure 1.1: High level abstraction of the scope of graph processing

The importance of this application domain is highlighted by its usability to solve many problems but hampered by it underlying irregular nature. There is an abundance of research attempting to optimize graph processing via both, software optimizations [6, 20, 29, 44, 46, 48, 60, 63, 66, 67, 74] and hardware accelerators such as FPGAs [4, 5, 14, 15, 19, 21, 52, 53, 71, 75, 83]. However, efficiently utilizing such hardware accelerators with ease is still an open research problem especially when considering irregular applications like graph processing [4].

This dissertation aims to address and benefit two research areas, heterogeneous computing and graph processing. Heterogeneous computing, more specifically CPU-FPGA hybrid systems, are explored where the architectural strengths of each device are exploited. Moreover, FPGA design and optimization techniques are explored to optimize the usage of its underlying architecture. On the other hand, graph processing algorithms are analyzed to overcome some of its inherent challenges. Combining the improvements in these two research areas results in a significant overall enhancement in graph processing and CPU-FPGA hybrid systems efficiency as well.

1.1 Research Problems

Optimizing graph processing is a well-researched problem, whether it is using a generalized approach on CPUs and GPUs [22, 48, 68, 69, 74] or application specific using FPGAs [5, 57, 71, 75, 81, 84, 85]. While generalized approaches are programmable, they don't necessarily yield the best performance. On the other hand, application-specific solutions are restrictive to the targeted application only. Hence, with the rise of heterogeneous computing (more specifically, CPU-FPGA hybrid systems) and High Level Synthesis (HLS) tools, there is great potential to efficiently harness that growing computational power for irregular applications (such as graph processing). This section dissects the research challenges into two categories,

graph processing challenges, and hybrid-systems utilization challenges. Chapter 2 elaborates on the previous work and how to build on it to improve both the performance of graph applications and the efficiency of utilizing heterogeneous systems.

Graph processing challenges Graph algorithms are inherently difficult to process efficiently, especially when targeting large-scale real-world graphs. This application has a large memory footprint, coupled with poor locality, low compute to memory access ratio and irregular memory access pattern. As a result, system performance is usually bound by the throughput of the external DRAM bandwidth. Large scale graphs have exploded in size in recent years and extracting information from these graphs is increasingly challenging. The performance bottlenecks that hamper graph applications are diverse and variable, not only do they depend on the graph algorithm and the underlying hardware architecture, but the size and structure of the graph is also a critical variant. Graph processing challenges are summarized in the list below:

- Poor locality and irregular data access pattern
  - Unstructured relationships between graph vertices lead to poor locality, which incurs frequent high latency global memory access.
- Lack of scalability
  - Communication between partitions of large-scale graphs causes heavy traffic, which inhibits scalability.
- Heavy data conflicts
  - Vertices from different partitions may read/write the same vertex simultaneously, leading to heavy conflicts.

Heterogeneous processing challenges There is a growing trend in high-performance computing towards heterogeneous platforms, where 25% of the top 100 supercomputers are designed using heterogeneous model employing either GPUs or FPGAs or both [18]. The use of heterogeneous systems in HPC to enhance performance is growing and proved effective for data parallel applications with regular memory access patterns [23, 56]. However, these powerful systems are still yet to be efficient in the irregular application domain such as graph applications [5, 21, 26, 71, 74, 75].

This dissertation focuses on CPU-FPGA hybrid systems. The research focuses on two aspects, exploiting the strengths of both CPU and FPGA platforms, and the process of synthesizing efficient designs on the FPGA. FPGAs offer a massively parallel configurable architecture that could be customized and optimized for a target problem and usually provide superior power efficiency, yet it has its limitations. Developing an efficient FPGA-based hardware design requires knowledge and expertise about the underlying configurable architecture. While the use of High-Level Synthesis (HLS) tools significantly reduced the level of effort to utilize FPGAs, it does not necessarily yield an efficient and performant hardware design [26, 88]. The FPGA's performance and scalability problems are aggravated when targeting irregular applications.

1.2 Methodology

This research aims to expand the scope of graph processing frameworks to extract the best performance out of heterogeneous computing platforms when targeting graph applications domain. Hence, to tackle the challenges illustrated in Section 1.1, this dissertation lays out how to address some of the inherent challenges in graph processing in addition to exploiting the strengths of both CPUs and FPGAs.

Figure 1.2 lays out the dissertation overview. First, a workload distribution framework is devised to partition the workload on devices in a CPU-FPGA hybrid system. Second, FPGA-specific optimizations are explored for irregular graph-like applications to improve the performance of the synthesized hardware. Third, an FPGA Domain-specific performance model framework is constructed to model the performance and automatically explore the optimization space for graph applications.

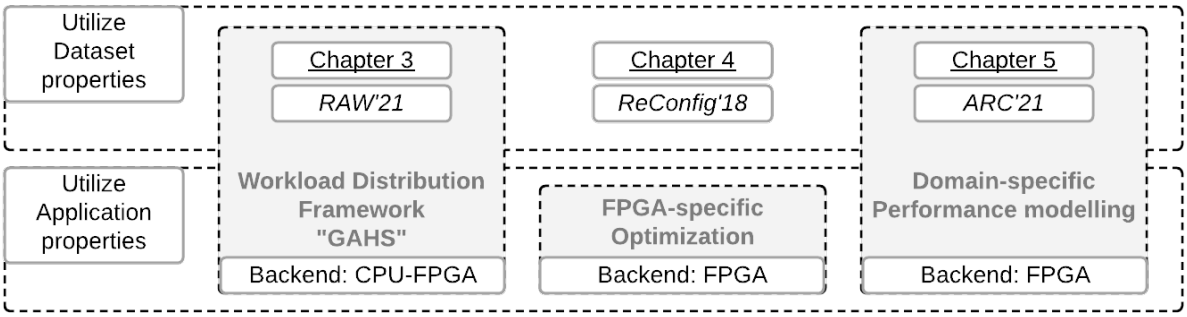


Figure 1.2: Dissertation overview.

The goal of this work is to optimize the performance of heterogeneous processing for graph applications. To harness the power of heterogeneous systems, one has to be able to exploit the architectural strengths of the available hardware devices. This dissertation focuses on CPU-FPGA hybrid systems, to optimize the performance of graph applications by exploiting knowledge of the graph-input dataset connectivity structure. However, exploiting the architectural strengths of FPGAs requires hardware skills, rendering it inaccessible to the masses. Hence the dissertation extends to explore how to efficiently utilize the FPGA's customizable architecture and optimize its performance, more specifically, when executing graph applications using HLS tools. Finally, research transitioned to performance modelling for HLS designs on FPGAs. Building on an established state of the art performance model [88], we fashioned a custom performance model specifically for graph-like applications synthesized for FPGA using HLS tools. Consequently, we were able to devise a framework that explores

the optimization space and recommends the best optimization strategy for a graph dataset.

Testing and experimentation of the frameworks was conducted on Intel's Devcloud platform. Intel acquired Altera [9] in 2016 and in their pursuit to democratize the use of heterogeneous systems and make it accessible to the masses, they constructed the Devcloud platform. It is an experimental cluster including many nodes, where each node is constructed as a CPU-FPGA hybrid system or a CPU-GPU hybrid system. During my research I have been granted access to the Devcloud platform to run my experiments to verify my research, which also serves them as a way of advertising for their beta software/hardware system. This allowed me to join their community, Intel Innovators, which is tasked with providing feedback of user experience of their system as well as participating in conferences and workshops to promote their heterogeneous ecosystem.

1.3 Contributions

This dissertation attempts to have a meaningful impact on both the graph analytics community and the heterogeneous/FPGA computing community. First, the graph processing community would benefit from improving the performance of graph analytics applications. Second, the FPGA community would benefit from research regarding FPGA specific optimization and domain-specific performance modelling which would ultimately serve the greater goal of including FPGAs into mainstream computing. The recent purchase of Xilinx by ARM [10], in addition to previous Altera acquisition by Intel [9], suggest trending towards tighter coupling of CPUs and FPGAs, perhaps on the same chip. The resulting collective impact of this research is well positioned to harness the computational power of a CPU-FPGA hybrid system, more specifically when targeting graph applications.

To achieve the goal of high-performance scalable graph processing on heterogeneous platforms, multiple underlying steps have been taken. Hence, this study was conducted in four phases. In a first phase, graph datasets are evaluated on CPUs and FPGAs to capture the execution profile of such an irregular application. In a second phase, multiple aspects of irregular applications are analyzed such as memory access patterns, in addition to the execution profile, to formulate the basis of a workload distribution plan in order to exploit the strengths of each hardware device in a heterogeneous system. Up to 6.5× speedup was achieved compared to CPU-only and FPGA-only implementations, by simply distributing the workload on a CPU-FPGA hybrid system. In a third phase, due to the inefficiency of irregular applications execution on FPGAs, architecture specific optimizations were explored to accelerate sub-optimal performance of graph applications on FPGAs. In a fourth phase, a custom performance model is built to project the execution profile of graph applications on FPGAs and explore the optimization space. Compared to the architecture agnostic implementations, 3.4× speedup was achieved executing graph applications using the automated framework's optimization recommendations. The framework recommended the best optimization strategy in 90% of the test cases.

To that end, the contributions of this dissertation can be summarized as follows:

- Augmenting graph input characteristics into a workload distribution framework (GAHS: Graph Analytics on Hybrid Systems), which is described in detail in chapter 3. Since graph applications follow a data-driven computation style, each graph input has its unique impact on the execution profile. Hence, generalizing an optimization approach for all graph datasets will not yield the best performance. Therefore, GAHS Utilizes the graph features to decide the best workload distribution over the resources of a CPU-FPGA hybrid system.

- Formalizing a method to optimize irregular applications such as graph processing when targeting hardware-based accelerators (HLS-based FPGA designs). Candidate code patterns for optimizations were identified and optimization recipes were created and applied as detailed in chapter 4.

- Addressing the irregular nature of graph applications that makes performance modelling erroneous, a domain-specific performance model has been devised for HLS-based FPGA designs. General purpose performance models are usually accurate for compute bound applications (i.e., regular applications), but fails to capture the execution profile of memory bound applications (i.e., graph applications). Exploiting knowledge of the graph structure aided the model to capture the execution profile of such a data-driven application domain, as depicted in chapter 5.

- Exploring the optimization space of graph kernels on FPGAs, an automated framework has been devised assisted by the domain-specific performance model. The framework guides the best optimization strategy for FPGA designs based on knowledge of the graph input properties and hardware design specifications. Chapter 5 details how such an automated framework relieves FPGA developers of the time-consuming burden of manually synthesizing and testing various optimization strategies attempting to maximize performance.

1.4 Dissertation outline

The rest of this dissertation is organized as follows. Chapter 2 discusses background and related work. Chapter 3 describes a framework for workload partitioning and scheduling of graph analytics on CPU-FPGA hybrid systems exploiting the architectural strengths of both CPUs and FPGAs collectively. Chapter 4 addresses the efficiency and optimization of utilizing reconfigurable architectures to process irregular applications such as graph problems. Chapter 5 presents a domain-specific performance model and an automated framework to explore the optimization space for graph applications on FPGAs. Finally, Chapter 6 provides a summary of the contributions and outlines future research paths.

Chapter 2

Background and Related Work

While there are many ways computer scientists can attempt to accelerate the graph processing problem and scale it efficiently (such as optimizing software libraries, abstracting domain specific APIs, and using hardware accelerators), this work focuses on utilizing heterogeneous systems. This dissertation builds upon the work of graph processing, FPGA optimization, and heterogeneous computing domains. This research endeavor aims to enhance the performance and scalability of graph applications while exploiting the strengths of heterogeneous platforms, more specifically, CPU-FPGA hybrid systems.

This chapter first introduces background about graph processing in general in Section 2.1, later, chapters 3,4,5 elaborate in detail the background of each specific graph application used. Afterwards, Section 2.2 discusses previous research in the heterogeneous computing area concerning graph processing. Then, Section 2.3 presents previous work related to utilizing graph input properties to influence the runtime decision of graph processing. Later, Sections 2.4,2.5 detail previous efforts to optimize and model the performance of HLS based FPGA implementations. Finally, Section 2.6 summarizes the research attempts in each area and how it is built upon in this dissertation to achieve the overarching goal of enhancing graph processing on heterogeneous systems.

2.1 Graph Processing Background

This section describes the algorithmic properties of graph algorithms and their impact on the underlying hardware architecture. Many graph applications depend on sparse matrix vector operations as shown in Figure 2.1, which like most graph algorithms are typically characterized by their workload imbalance, unpredictable control flow, and irregular memory-access patterns. Such an application domain usually executes data-driven computations dictated by the connectivity structure of the graph, while performing a relatively small amount of computation, which makes execution time governed by the memory access latency.

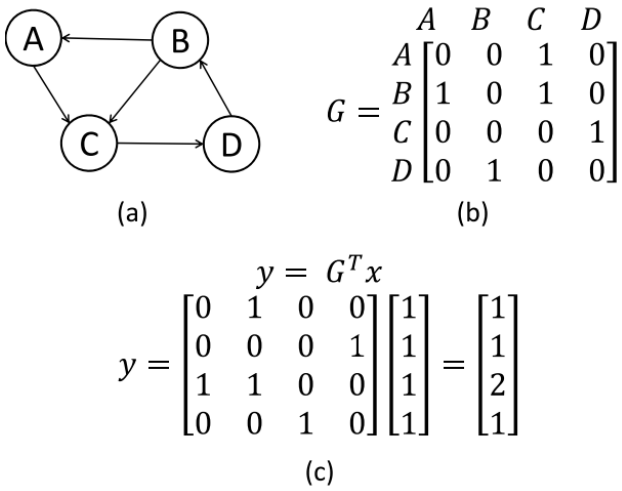


Figure 2.1: Graph (a) Logical representation (b) Adjacency matrix (c) In-degree calculation as SPMV (adopted from [69]).

Such pointer-based computation is unstructured and requires highly irregular fine-grained random memory accesses. This leads to poor temporal and spatial locality, which inhibits performance scalability on cache-based processors, especially when targeting real life large-scale graphs.

Properties of this application domain are listed below:

- *Data driven computation:* Most graph algorithms are iterative, where each iteration depends on the data processed (vertices and edges) in the previous iteration.
- *Irregular memory access:* Indirect, pointer-based memory references lead to random memory accesses with high latency.
- *Poor temporal and spatial locality:* Portions of a graph algorithm may be able to employ data reuse (high temporal locality), while other portions of the algorithm may have contiguous memory accesses (high spatial locality). Taking advantage of both locality types is challenging, while optimizing for one type and neglecting the other typically causes a performance hit.
- *Low compute to memory access ratio:* A significant amount of the graph workload is for traversing the graph while a much smaller amount is required for doing actual computation. This is the reason for characterizing this application domain as memory bound.

Graph applications are generally characterized by a low computation to communication ratio. A previous study by Ham et al. [22] showed that more than 90% of a graph algorithm's instruction (or execution time) is spent on traversing the graph, which translates to communication or global memory accesses. On the other hand, less than 10% of the instructions are for computation.

General-purpose processing (CPUs) is not ideal for such an execution profile. The fixed memory access granularity based on cache line sizes is rigid and wasteful in terms of memory bandwidth. This irregular application domain requires much finer granularity when accessing the global memory, where in the case of coarser grained CPU memory transaction,

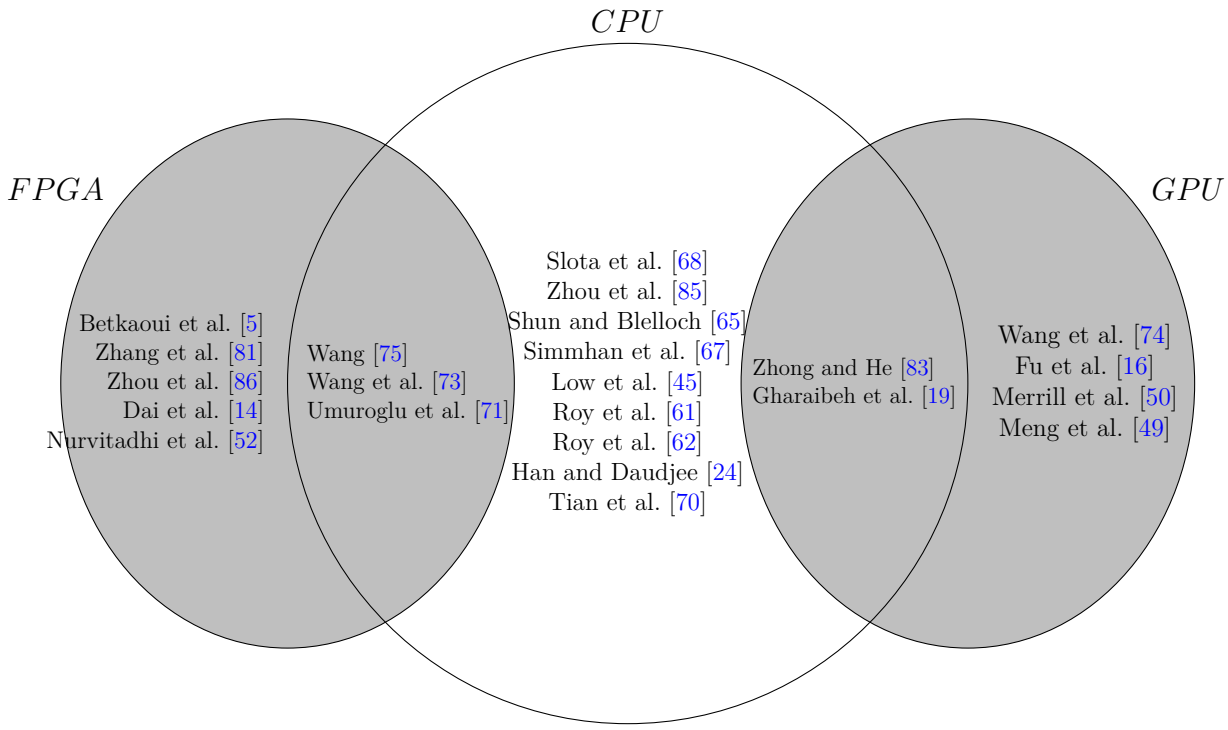


Figure 2.2: Overview of graph solvers: classification of hardware platforms

simply translates to wasted memory bandwidth. Hence, when executing such an irregular application, the little or no spatial and temporal locality of the application make it challenging to scale good performance on general-purpose CPUs. GPUs, on the other hand, offer massive parallelism for Single Program Multiple Data (SPMD) like applications. Graph applications with varying degrees of parallelism are a challenging target to harness the computational power of GPUs. While CPUs and GPUs are easier to program, FPGA designs are more flexible and power efficient. The data-driven execution profile of FPGAs saves a lot of the energy used by CPUs and GPUs to decode instructions and look up data in the cache, which usually consume the majority of power in instruction-based architectures. Table 2.1 shows an overview of the latest graph processing frameworks and their properties, while Figure 2.2 shows a visual representation of the hardware platform classification of graph solvers.

Graph processing framework	Year	System architecture	Hardware platform
[68] Solta et al.	(2015)	Distributed	CPU
[85] Zhou et al.	(2017)	Distributed	CPU
[65] Ligra: Shun, et al.	(2013)	Shared-memory	CPU
[67] GoFFish: Simmhan, et al.	(2014)	Distributed	CPU
[45] GraphLab: Low, et al.	(2010)	Shared-memory	CPU
[61] X-Stream: Roy, et al.	(2013)	Shared-memory	CPU
[62] Chaos: Roy, et al.	(2015)	Distributed	CPU
[24] Giraph: Apache Software Foundation	(2012)	Distributed	CPU
[70] Giraph++: Tian, et al.	(2013)	Hybrid	CPU
[73, 75] Wang, et al.	(2018)	Shared-memory	FPGA+CPU
[71] Umuroglu, et al.	(2015)	Shared-memory	FPGA+CPU
[5] Betkaoui, et al.	(2012)	Distributed	FPGA
[81] Zhang, et al.	(2017)	Shared-memory	FPGA
[86] Zhou, et al.	(2018)	Shared-memory	FPGA
[14] FPGP: Dai, et al.	(2016)	Shared-memory	FPGA
[83] Medusa: Zhong, et al.	(2013)	Distributed	GPU+CPU
[19] TOTEM: Gharaibeh, et al.	(2013)	Hybrid	GPU+CPU
[74] Gunrock: Wang et al.	(2016)	Distributed	GPU
[16] MapGraph: Fu, et al.	(2014)	Distributed	GPU
[50] Merrill, et al.	(2012)	Distributed	GPU
[48] Pregel: Malewicz, et al.	(2010)	Distributed	N/A
[69] GraphMat: Sundaram, et al.	(2015)	Shared-memory	N/A

Table 2.1: Overview of graph processing frameworks.

2.2 Heterogeneous Computing & Workload Partitioning

Various graph solvers make use of hardware accelerators to improve the performance of graph processing either using GPUs [16, 50, 74] or FPGAs [14, 25, 42, 52]. GPUs may appear poorly suited for graph processing, since massive parallelization with lack of adequate synchronization introduces load balancing and under-utilization problems. However, as stated by Merrill et al. [50], two key optimization techniques may alleviate these challenges, multi-threading and fine-grained synchronizations. Unlike the main GPU-based research, Zhong and He [83] and Gharaibeh et al. [19] leveraged the heterogeneity of CPU-GPU hybrid platform to

enhance the performance of graph processing. Zhong and He [83] introduced Medusa, a programming framework that exposes C/C++ APIs to make use of GPUs' massive parallel data structure. On the other hand, TOTEM, introduced by Gharaibeh et al. [19], distributes the workload on both GPUs and CPUs, where highly parallel workloads are assigned to GPUs, while small parallelism degree workloads are assigned to CPUs.

Many FPGA-based graph processing frameworks have been proposed, such as GraphGen [52], FPGP [14], GraphOps [53], ForeGraph [15], and others [5, 25, 42, 71, 75, 81]. However, the cooperative use of CPU-FPGA hybrid systems was not as extensively researched.

Research proposed by Umuroglu et al. [71] exploits a CPU-FPGA heterogeneous device to accelerate graph processing, specifically the Breadth-First Search (BFS) problem. They aim to maximize the memory bandwidth utilization and create a stall-free data path on the FPGA. The workload is partitioned to offload low parallelism iterations on the CPU while leaving the high parallelism iterations to the high-throughput FPGA. They achieve an average traversal speed of 172 million traversed edges per second (MTEPS).

Other work by Wang el al. in [75, 76] exploits the close coupling of CPU and FPGA in the shared memory architecture of the Heterogeneous Architecture Research Platform (HARP). The authors optimize data access by optimizing the on-chip data reuse. Moreover, they utilize the heterogeneity to implement Processor Assisted Scheduling (PAS) which offloads the scheduling tasks to the CPU and leaves the high-throughput data crunching to the FPGA. While employing hardware and pre-processing optimizations, they achieved a maximum performance of approximately 280 MTEPS for sparse graphs. While this work strives to optimize the custom FPGA design to maximize performance, it makes their design strictly application-specific.

2.3 Graph-Input Aware Research

This section highlights the impact of the graph input on the execution profile of a graph processing framework. Programmable graph processing frameworks such as Gunrock [74] tend to integrate graph primitives to augment a general-purpose library. However, as stated by Meng et al. [49], these programmable approaches may fail to deliver optimized performance due to sensitivity to the input graph instance in addition to variations in the execution profile of different graph algorithms. For example, the best hand-tuned BFS implementation using Gunrock library achieves 47 GTEPS on a power-law graph, while executing a road-net graph performance drops to 44 MTEPS. The optimizations applied to graph traversal applications such as BFS, are not applicable to the denser workload of centrality measure applications such as PageRank.

GSWITCH, proposed in [49], is a pattern-based algorithmic auto-tuning system. It includes a set of algorithmic patterns that assemble variants of the algorithm and dynamically switches between optimization strategies with negligible overhead. This framework abstracts a graph algorithm as a set of algorithmic patterns characterized by quantitative parameters. Other work [79] proposes an input aware auto-tuning framework for parallel sparse matrix-matrix multiplication called IA-SpGEMM. This is strongly related to graph applications since the two application domains share many of characteristics. IA-SpGEMM automatically decides the best sparse format for arbitrary sparse matrices to enhance the performance based on sparse features from the input including statistics of the matrix sparsity, the ratio of non-zero elements and its variance. This work is limited to deciding the best sparse format to be used. While both [49, 79] work was tested on CPUs and GPUs, none took advantage of the heterogeneity of the system.

2.4 HLS Design and Optimization on FPGAs

Czajkowski et al. [12] were the first to propose Altera's OpenCL compiler demonstrating four well-known applications: Monte Carlo Black-Scholes (MCBS), matrix multiplication (SGEMM), finite difference (FD), and particle simulation (Particles). To enhance the performance of dynamic programming on FPGAs, Settle [64] introduced OpenCL pipes [64], which improves the performance by 1.5× and 9.9× in comparison to the GPU and CPU implementations, respectively, while providing energy savings of up to 26-fold. Both endeavors achieved high utilization of FPGA resources with low clock frequency (less than 200 MHz). Moreover, the FPGA-specific implementations differed from their GPU-based counterparts. While the GPU implementation used SIMD-like parallelism, the FPGA implementation adopted a MIMD-like execution where each thread executed a distinct operation on a set of data items.

More recent research efforts proposed by Zohouri et al. [88] evaluated the performance of six regular benchmarks from the Rodinia suite using the Altera OpenCL SDK on a Stratix V FPGA. The original OpenCL implementations followed the bulk synchronous parallel (BSP) execution model, targeting GPU-like architectures with massive multi-threaded execution. Unfortunately, this approach can degrade FPGA performance due to barrier synchronization points that dictate flushing the pipeline, effectively halving the pipeline throughput. The authors in [88] reached the conclusion that FPGA-specific optimizations must be applied to the OpenCL kernels to yield efficient, high-performance hardware designs. In particular, they outlined five main FPGA-specific optimization techniques: compute unit replication, vectorization (or "SIMD-iztion"), loop unrolling, shift registers, and sliding windows. These optimizations improved the performance by up to two orders of magnitude compared to the BSP OpenCL kernels and achieved 3.4× better power efficiency when compared to the

NVIDIA K20c GPU.

The work published by Zohouri et al. [88] was extended in [55] to evaluate the performance of three different design methodologies for FPGAs: general-purpose manycore system (30 Nios II soft-core processors), FSM-based architecture using LegUp HLS tool (MIMD architecture with focus on lower latency), and Intel's FPGA SDK for OpenCL (deeply-pipelined architectures with focus on higher throughput). The experiments showed that the FSM and soft-core implementations have scalability issues that are mainly related to cache conflicts and capacity misses. This issue was partially solved using a multi-banked cache design. However, the OpenCL implementations still outperformed both approaches across all the applications with up to two orders-of-magnitude speedup.

Other work [40, 72] used the OpenDwarfs benchmark suite to evaluate the performance of the OpenCL programming model on a Stratix V FPGA using the Altera OpenCL SDK. Kernels from regular application domains were tested, such as N-body methods, structured grids, unstructured grids, and dense linear algebra. Unlike Rodinia, the OpenDwarfs suite provides architecture-agnostic OpenCL kernels rather than GPU-specific (i.e., GPU-biased) implementations. These kernels were used as the baseline for comparison on the CPU, GPU, Intel MIC, and FPGA architectures. The authors explored FPGA-specific optimization techniques that exploit different parallelism levels as well as minimizing data movement across the memory hierarchy. It was also reported that the architecture-agnostic OpenCL kernels yielded inefficient hardware designs, which further suggests the need for FPGA-specific optimizations.

Finally, XSBench, a proxy application for Monte Carlo simulation, was used in [47] to evaluate the performance of OpenCL applications with irregular memory accesses on FPGAs on an Intel Arria 10 FPGA platform. The authors applied three different optimizations and evaluated their effect on performance. A fused multiply-add unit was integrated into the

design. The BRAMs were used to implement a constant cache along with data pre-fetching and packing techniques. The final optimization technique used vector data types and stored them in private memory. Applying these optimizations delivered a 50% improvement in energy efficiency, while sacrificing 35% of the performance compared to an Intel Xeon CPU with eight cores.

2.5 HLS Performance Modelling and Design Space Exploration

As demonstrated in multiple publications [7, 17, 39, 55, 77], the expected performance for HLS design is highly unpredictable. Also, HLS tools require FPGA-specific optimizations more often than not in order to yield efficient hardware architectures [26, 40, 47]. One of the earliest works on modelling performance for HLS design was presented in [77]. Static and dynamic analyses were used to build an analytical performance model for the key architectural features of FPGAs under the OpenCL programming model. This tool can predict the performance of OpenCL kernels with different combinations of FPGA-specific optimizations. This greatly helps in guiding the code-tuning process for performance purposes. However, this approach depends on collecting information by performing static analysis of the LLVM code, dynamic profiling of the OpenCL application execution on GPUs, then feeding the information into the analytical model. This amount of information is not readily available for users and is not easily reproducible for new applications.

Zohouri et al. [88] presented one of the latest works in the performance of OpenCL kernels on a Stratix V FPGA. They proposed in this work an analytical performance model that captures the baseline performance of HLS designs. The approach presented is sound but will

only be feasible in the case of compute bound applications. However, in the case of irregular memory access applications such as memory bound graph applications, the model fails to depict the execution profile accurately. Moreover, the exploration of optimization techniques was wildly off par. This is mainly due to the simplistic memory access model presented in their work.

Other interesting work proposed by Da Silva et al. [13], explores the idea of extending the roofline model [78] to the HLS design space. This work utilizes the resource consumption and the parameters used in the HLS tools, to maximize the performance and the resource utilization within the area of the FPGA. This work considers the resource utilization on the FPGA to maximize area usage by replicating the Processing Entities (PEs), and in turn maximize performance. However, in addition to the limited bandwidth of global memory access on FPGAs, complications may arise due to the irregular memory access patterns of graph applications. Moreover, this work only recognizes the peak performance attainable on the FPGA design, which is not realistic considering the irregularity of the graph applications domain.

Another effective design space exploration work was proposed in [82], where they recursively quantify the loop latency through an analysis on the LLVM-IR level. However, a major drawback that would make this work not well suited for graph applications, is that their analysis depends on having static loop bounds. This work is efficient in the case of regular memory access applications, where the loop bounds are usually static and global memory can be accessed through fixed stride accesses. However, unfortunately this is not the case for graph applications. Graph applications have an inner loop which is dynamically bound depending on the data input which makes this work an un-viable candidate for design space.

2.6 Discussion

In summary, previous studies by Meng et al. [49], Xie et al. [79] showed the impact utilizing graph input properties to influence runtime decisions in graph processing and sparse matrix operations. Moreover, it has been established how powerful heterogeneous systems can be if its computational potential can be efficiently harnessed. Hence, this dissertation explores how to exploit the strength of each device in a CPU-FPGA hybrid system supported by the knowledge gained from graph input analysis, which facilitated the formulation of an automated framework for workload distribution and scheduling on both CPUs and FPGAs. Chapter 3 elaborates about this topic in detail.

Focusing on FPGA implementations, previous studies showed the need for applying FPGA-specific optimizations to OpenCL kernels to generate efficient custom hardware accelerators. However, the expected performance and efficiency is highly dependent on the characteristics of the target application as well as the graph input properties. Moreover, generic OpenCL kernels (following the BSP execution models), achieve high performance on GPUs, but generate extremely inefficient FPGA designs. While previous work generally focused on regular OpenCL kernels, this dissertation attacks the problem of optimizing the more challenging problem of *irregular* applications such as graph processing. The importance of this particular study is highlighted by the development of new benchmarks, such as CHO [51], developed for the sole purpose of providing unoptimized OpenCL kernels to test the FPGA capabilities using OpenCL compilers. Further details about the FPGA-specific optimization are examined in Chapter 4.

Modelling the performance of such FPGA designs is challenging especially in the case of irregular applications. Previous research relied on dynamic profiling [77], simplified memory access modelling [88], and oversimplified assumptions [82]. However, this dissertation tackles

the performance modelling for irregular graph applications by augmenting the innovative idea of using information from the dataset input. Building on the performance model proposed in [88] and augmenting it with knowledge from graph-input aware research, we were able to capture the execution profile of irregular memory accesses. This addressed the problem of simplified memory access modelling without the need for extra information gathering using dynamic profiling which is discussed in detail later in Chapter 5. This work extends the use of graph input properties which helped in formulating an automated framework for exploring the optimization space.

Chapter 3

Workload Distribution on CPU-FPGA Hybrid Systems

This chapter focuses on a high-level view of system design, in particular heterogeneous computing. The focus in this chapter is shifted towards a CPU-FPGA hybrid system, where the architectural properties of each device are exploited to enhance the performance of graph applications processing. This chapter examines two hypotheses. First, a variety of properties of a set of graph instances are extracted and examined to determine if they can be used to predict the analytical execution profile on different hardware devices. Second, can this information be used to partition the workload and properly schedule it on a hybrid CPU-FPGA system. This work is intended to enhance the performance and scalability of graph processing by exploiting the heterogeneity of CPU-FPGA hybrid systems [28].

This chapter details a framework called Graph Analytics on Hybrid Systems (GAHS) for workload partitioning and scheduling of graph analytics on hybrid systems. The decision-making process of the framework is configured to be guided by data input properties. The goal of this work is to expand the design space exploration to include both CPUs and FPGA. Moreover, GAHS not only depends on the characteristics of the application and the available hardware resources but also includes data input properties.

3.1 Motivation & Goal

Many real-world problems can be presented as large-scale graphs with millions of vertices and billions of edges, such as social networks, biological interactions, and web graphs [37, 63]. Graph analytics is an area of study used to extract useful information from these huge data-sets. For instance, in web graphs, web pages represent the graph vertices while hyperlinks between them represent the edges. Analyzing these types of graphs (in this case, usually using the PageRank algorithm [54]) are used for web search engines, ranking websites, and network analysis among other uses. This emphasizes the motivation for enhancing the performance and scalability of graph applications.

3.1.1 Challenges

Performing graph analytics to extract useful information from these ever-growing graphs is a complex and challenging task. The sheer size of these graph data-sets combined with the irregular nature of the underlying problem pose a significant challenge for performance, scalability, and power efficiency [4, 26]. Large real-world graphs are challenging to process efficiently, not only due to their large memory footprint, but most graph algorithms entail irregular data-dependent memory access patterns with low compute to memory access ratio. To complicate matters further, these data-sets tend to be scale-free (following the power-law degree distribution), which makes load balancing and access locality harder to achieve. Analyzing and extracting useful information from this big data domain is challenging, which emphasizes the important role of high-performance scalable graph processing.

Previous research attempted to overcome the challenges of graph processing using the growing trend in heterogeneous high-performance computing. The use of hybrid systems to enhance the performance of applications is growing and proved effective for data-parallel

applications with regular memory access patterns [23, 56]. However, these powerful systems are still yet to be efficient in the irregular applications domain such as graph applications [5, 21, 26, 71, 74, 75]. On the other hand, there is little prior work that explores optimizing the graph processing environment using the features of the input graph instance while targeting a hybrid ecosystem.

3.1.2 Approach

Observing a hybrid ecosystem of hardware including CPUs and FPGAs, each platform excels in a specific aspect. Generally, CPUs are easily programmable and can take advantage of instruction-level parallelism and sophisticated cache policies. On the other hand, FPGAs offer superior power efficiency and customizable hardware architecture to fit an application's needs. Hence, this chapter presents a graph analytics framework (GAHS) that exploits the strengths of each platform through efficient workload partitioning on a CPU-FPGA hybrid ecosystem. Moreover, GAHS auto-tunes the workload distribution based on graph input features (since it is a data-driven application). Not only that, but GAHS also learns as it goes. Augmenting GAHS's library with more execution profiles helps in fine-tuning the workload distribution decision making process.

3.1.3 Contributions

Using multiple workload distribution designs and graph input features, GAHS framework was able to achieve significant performance improvement compared to CPU-only, FPGA-only implementations, and state-of-the-art hybrid FPGA solvers. Results show that up to 6.5× speedup can be attained over a CPU-only or FPGA-only implementations through proper distribution of the workload. Moreover, GAHS achieves an average of 18× speedup

compared to state-of-the-art hybrid CPU-FPGA solvers. The contributions of this work are listed below:

- Analyzing and profiling the execution of multiple graph types with a wide range of properties.
- Devising a novel framework that uses graph input features to decide the best workload distribution over the resources of a CPU-FPGA hybrid system.
- Constructing a self-learning technique, where the framework fine-tunes its decision-making process as it ingests more graphs via a feedback loop.

3.2 Background

This section explains the background of the PageRank application, the graph data-set, and the workload distribution configurations used. This is intended to introduce the problem scope and design space. PageRank application is used as a case study to showcase GAHS, so a detailed explanation of the algorithm and its kernel implementation is presented in Section 3.2.1. Then we elaborate on the graph data-set and the feature-set used to characterize graph properties. Finally, we demonstrate our workload distribution configurations.

3.2.1 PageRank Algorithm

PageRank (PR) is an algorithm used by the Google search engine to rank web pages [54]. Initially, PR assigns an equal PR value of the reciprocal of the number of vertices to all the vertices of the graph. Afterward, a series of *superstep* iterations are executed that terminate either after a user-defined iteration count or an automated convergence check condition. In

each iteration, each vertex donates a share of its PR value to its neighbors along its outgoing edges. The number of outgoing edges of each vertex is used to divide the PR value equally among the recipient vertices. After the summation operation of each vertex calculating the total PR value donated to it, a damping factor is applied to the final PR value. The damping factor is a value set between 0 and 1, to which the default is 0.85. For this work, the default damping factor is used and the number of iterations of PageRank is set to 20 iterations.

The implementation in this chapter as in Chapter 4 is based on the SPMV-based PageRank algorithm from [8]. The kernels breakdown is shown in Table 3.1. The graph input is imported into a *compressed sparse row* (CSR) data structure and a CSR initialization kernel is invoked. The CSR initialization kernel is responsible for updating the sparse matrix that represents the graph to be compatible with SPMV calculations. Originally the sparse matrix included only the connectivity structure of the graph; however, after invoking the CSR initialization kernel, it is updated to consider the portion of PR value a vertex will receive from its neighbors. This allows a standard SPMV kernel to perform the PR value update, where the SPMV's dense vector represents the previous iteration's PR values for the vertices without the need for atomic addition (which is required in the original algorithm). The added overhead of including the CSR initialization kernel is insignificant relative to the overhead of the atomic addition operation required by the original PageRank algorithm [8].

Kernel code-name	**K1**	**K2**	**K3**	**K4**
Kernel name	init_buffer()	CSR_initialize()	SPMV()	PR_update()
Description	Initializes PR buffers	Constructs CSR matrix from the graph	Performs SPMV calculations	Updates PR array with the damping factor

Table 3.1: Kernels of the SPMV-based PageRank algorithm and the relevant kernel naming code used throughout this chapter.

3.2.2 Graph Data-set and Features

The network data repository [59] is used as the source of real-world graphs used in this work. To ensure the generality of different types of graphs for the framework's learning process, we use a set of graphs with a wide range of properties as shown in Table 3.2. Although only a sample of the graphs is shown, 25 unweighted directed graphs were used spanning different types of graph data-sets including web graphs, social networks graphs, email graphs, citation networks, ecology networks, and road networks. This sample is used to evaluate the initial parameters of the framework's decision tree. Graph analytics applications (including PR application) follow a data-driven execution profile, which emphasizes its high sensitivity to the properties of the input graph. Since the goal is to capture a graph's sparsity and connectivity structure, we chose the features shown in Table 3.2, which are explained below.

- CC (Clustering Coefficient): is a measure of the tendency of graph vertices to cluster together.
- D (Diameter): is the greatest distance between any pair of vertices in the graph.
- DEG (average degree): is the average number of edges for graph vertices.
- SCC (Strongly Connected Component): is a maximal strongly connected subgraph.

This feature-set has been selected based on graph parameters used in [49, 79] for input-aware execution, augmented with additional parameters that capture the graph connectivity

Type	Graph	Vertices	Edges	CC	D	DEG	SCC
Road network	road-road-usa	23947347	28854313	0.0176	10	2	0
Collaboration network	ca-hollywood-2009	1069126	56306654	0.7664	4	105	0.000001
Web graph	web-Stanford	281903	2312498	0.5976	10	16	0.533985
Social network	socfb-Georgetown15	9414	425639	0.225	3	90	0.000106
Ecology network	eco-foodweb-baydry	128	2138	0.3346	2	33	0.804688

Table 3.2: Sample of the data-set used for initializing GAHS.

structure. The workload per iteration in the PR application depends on the number of neighbors for each graph node, hence, the selected feature-set has a direct impact on the amount of computation and memory requests in each iteration.

The features and statistics of the data-set have been collected using SNAP library functions [43]. We use this feature-set to capture the structure of a graph instance. This structure directly affects the execution profile of the graph application in multiple ways. For instance, it affects the data parallelism granularity and memory transactions granularity. While CPUs take advantage of sophisticated cache policies, the fixed memory access granularity (which is based on the size of a cache line) is not well suited for graph processing. Graph applications may require much finer granularity when accessing the global memory, which in the case of coarser-grained CPU memory transactions simply translate to wasted memory bandwidth.

The graph structure also affects the pipeline parallelism efficiency, which is important for FPGA implementations. The data-driven execution model of FPGAs is well suited for the graph application domain. However, the graph structure may impose stalls in the data path pipeline (waiting for global memory access), significant performance degradation is observed. Hence, when executing such an irregular application with limited spatial and temporal locality, we aim to exploit the strengths of both general-purpose CPUs and application-specific FPGAs.

3.2.3 Workload Distribution Configurations

The execution of the PageRank algorithm was profiled for both CPU execution and FPGA execution to come up with reasonable partitioning of the workload over the available resources. The workload distributions chosen for this work are based on the execution profile of multiple graph inputs. Four design configurations were used to partition the workload

between the CPU and FPGA in addition to CPU-only and FPGA-only configurations. In these cases (CPU-only and FPGA-only) the workload is not distributed over multiple devices, rather offloaded to one device only. Depicted in Table 3.3 are the different workload partitioning designs with the distribution of kernels among CPUs and FPGAs. Offloading kernels to the appropriate hardware resource in a hybrid system shows significant performance improvement as discussed later in Section 3.4.

Configuration	**Workload distribution**			
	K1	**K2**	**K3**	**K4**
Config-CPU	CPU	CPU	CPU	CPU
Config-FPGA	FPGA	FPGA	FPGA	FPGA
Config-1	FPGA	CPU	FPGA	FPGA
Config-2	FPGA	CPU	CPU	FPGA
Config-3	FPGA	CPU	CPU	CPU
Config-4	CPU	CPU	FPGA	FPGA

Table 3.3: The distribution of kernel computations over the hybrid system for the different workload distribution configurations.

During the initial stages of building the framework, it was observed that the "CSR_initialize" kernel denoted as "K2" is significantly slower when executed on the FPGA compared to the CPU implementation. The reason is the many irregular global memory accesses required by that kernel that are not amortized over the very lightweight computation of the kernel. This influenced the decision when designing the workload distribution configurations to always offload that kernel to the CPU (except in the FPGA-only configuration). This emphasizes the importance of sharing the workload across the available hardware resources in the hybrid system.

3.3 Graph Processing on Hybrid Systems Using GAHS

The goal of GAHS framework is to make the best use of the available resources in a hybrid system. The general structure of GAHS is shown in Figure 3.1, which shows the breakdown of the components of the framework. GAHS is divided into two major parts, a back-end, and a front-end. The back-end constructs the analysis and learning process while the front-end recommends workload distributions for new incoming graph instances. GAHS is first initialized using the execution profile of a graph data-set with a wide range of properties. Then, it correlates relations between the execution profile of a graph and the different workload distribution configurations. After that initial start, GAHS can start recommending workload distribution configurations to new graphs. It also makes use of a feedback loop that uses the new graphs' execution profile to refine its analysis and recommendation process.

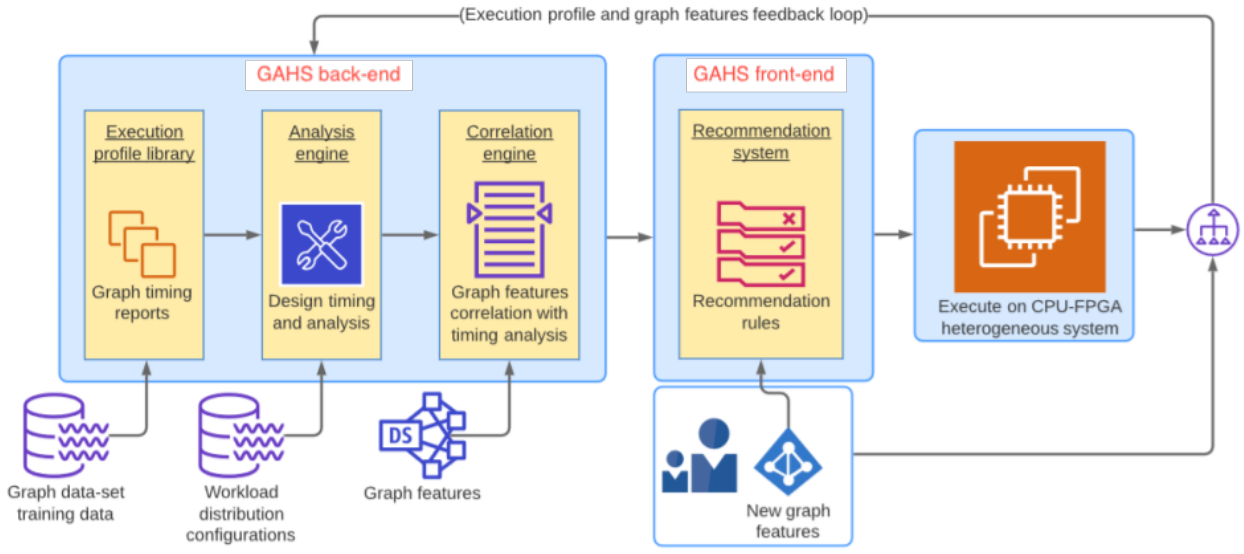


Figure 3.1: GAHS framework general structure and its associated components.

3.3.1 Execution Profile Library

The first module of GAHS's back-end is responsible for quantifying the execution profile of the application. This module is responsible for maintaining data structures and storing the initialization graphs' execution profiles in addition to execution profiles acquired from a learning feedback loop. A custom data structure was developed in this module to store all the relevant information of a graph's execution profile, including graph names, timing reports, and indexing pointers for easy information retrieval.

3.3.2 Analysis Engine

The analysis engine examines the properties of a graph instance along with its execution profile. This is a major component of the framework designed to find the best configuration for a graph instance with the objective of minimizing the execution time. This module is aware of the different design configurations identified earlier in Section 3.2.3. The information from the execution profile library is extracted and combined with the appropriate workload distribution design configuration. At this point, the analysis engine has the execution profile of each kernel along with information about the hardware resources used to execute each kernel. Next, the analysis subroutine is initiated to analyze the application's execution profile breakdown.

The execution profile is broken down into three major parts, host-device communication, kernel execution time, and workload distribution overhead. Although both the CPU and FPGA are considered computational devices, in this instance the 24-core CPU is referred to as the host while the FPGA is referred to as the device. The host-device communication pertains to loading initial data to the device and unloading the final result to the host. This is mainly affected by the memory bandwidth of a hardware resource. The kernel execution

time is purely dependent on the efficiency of the hardware resource used when executing a specific kernel. For example, high data parallelism and fine-grained kernel computations are more suited to FPGA execution, while irregular cache-friendly kernels are better suited for CPU execution.

The workload distribution overhead pertains to the overhead incurred by moving data from one hardware device to the other. It is calculated as a ratio of the communication time relative to the performance gain of offloading computation to another device. This overhead is mitigated by parallel execution on more than one hardware device; however, its effect is not negligible. The analysis engine calculates the relative overhead coefficient that results from using a specific configuration with a specific graph input. This relative overhead coefficient is used later in the recommendation system.

3.3.3 Correlation Engine

The correlation engine starts by drawing relations between the analyzed information from the analysis engine and the graph input features. It is responsible for drawing links between graph input characteristics and hardware architecture abstractions for each kernel. The set of features identified previously in Section 3.2.2 are used to set the parameters that construct a scoring system. Thresholds for these parameters are set by the correlation engine using empirical evaluation of the initialization data-set. However, the more execution profiles learned by the framework, the more fine-tuned these thresholds are. The scoring system is explained later in Section 3.3.4, but the parameter thresholds are set at this stage. This module includes an iterative method that sets the parameter thresholds based on graph features linking it to the best workload distribution configuration. Absolute graph features identified in Section 3.2.2 are used in addition to a combined feature ($\frac{V}{E}$), which is a measure

of the sparsity of the graph.

- (V, E): The graph size is used to evaluate whether the graph fits in the cache memory of the CPU.

- (CC): Higher clustering coefficients translate to higher data reuse between different vertices. Properly designed hardware implementations can exploit data reuse to significantly enhance performance on FPGAs.

- (DEG): The degree of a vertex determines the number of its neighbours, which directly translates to the number of memory requests required to perform computation for that vertex. FPGAs offer flexible granularity for memory requests making use of burst coalescing memory access up to a maximum threshold (defined by the hardware specification). On the other hand, CPU cache has a fixed granularity, where if it doesn't match that of the graph instance may lead to many cache misses.

- $(\frac{V}{E})$: Graph sparsity directly impacts the number of stall cycles required to resolve global memory requests.

After the initial learning phase, the parameters thresholds are relayed to the recommendation system to configure the scoring system. The correlation engine is rerun each time a new graph is processed to calculate the new threshold. However, the overhead impact of rerunning that module is insignificant. The effect of the new graph feature values on the threshold is weighted giving the history of learned statistics a heavier weight. The weight factor is related to the number of learned graphs versus new incoming graph profiles. The new threshold value is the weighted average over all the previously and newly learned graphs.

3.3.4 Recommendation System

The front-end of the framework is responsible for deciding the workload distribution for a new graph input. GAHS's front-end starts by ingesting the graph's features to decide the best workload distribution. The recommendation of using a specific design configuration is dependent on a scoring system. Graph features are used to calculate the score of every design configuration available, based on the preset thresholds of the correlation engine. Only a sample of the scoring system's decision tree pseudocode is previewed in Listing 3.1 to showcase its methodology; however, some evaluations take precedence before resorting to the scoring system. The graph size is first evaluated using (V, E) features to determine if its working set is small enough to fit in the cache memory of the CPU. In that case, `Config-CPU` is the design configuration of choice. Otherwise, the recommendation system is employed to decide which workload distribution is most suited for the graph input instance. Thresholds for the scoring system are heuristically set by the correlation engine, which was initialized using a data-set with a wide spectrum of properties.

Listing 3.1: Sample of the scoring system (preview).

```
if(graph.CC > CC_threshold)
    config-2-score ++;
else
    config-3-score ++;
    config-4-score ++;

if(graph.DEG < DEG_threshold1)
    config-4-score ++;
elseif(graph.DEG < DEG_threshold2)
    config-3-score ++;
else
    config-FPGA-score ++;
```

3.3.5 Fine-Tuning Feedback

A graph instance is executed on the hybrid system using the workload distribution of the design configuration suggested by GAHS. The execution profile along with the graph features are relayed to GAHS's back-end for the feedback learning process. As the framework learns new graphs and reruns the back-end part of the framework, the correlation engine updates the recommendation system with the new feature thresholds to reconfigure the scoring system. Hence, the system becomes more accurate in predicting the best design choice. Initially it needs a collection of graphs with a wide range of properties to jump-start the framework. 25 graphs were used to initialize the framework and achieved an initial prediction accuracy of 89%.

3.4 Results & Evaluation

This section explores how GAHS is capable of enhancing the performance of the PageRank application on a hybrid platform. It starts by discussing the hardware resources used in this work, then steps into the details of performance evaluation and a deep analysis of the framework's results.

3.4.1 Hardware Platform

The computational platform used is the Intel Devcloud [36]. The computational node includes an Intel Arria 10 GX FPGA (Intel PAC Platform) and an Intel Xeon Gold 6128 CPU running at 3.40GHz. The FPGA board has two 4-GB DDR-4 memory banks, 1150-K logic elements, 54,260-Kb of M20K on-chip BRAM memory, and attaches to the host through a PCIe interface. The FPGA compiled designs run at an average frequency of 275 MHz. On the other hand, the CPU includes 24 cores, 32-KB L1 cache, and is running Ubuntu 18.04.

3.4.2 Testing Data-Set

100 real-world graphs are used for testing the framework from the network data repository [58, 59] with graph sizes up to 100-M edges. A sample of the dataset is shown in Table 3.4 due to the very large size of the data-set. Table 3.4 also depicts a graph ID for each graph to link to the performance plots. We chose to use the graph ID in the plots as the graph names are too long to include in the figures. The last column to the right in Table 3.4 shows GAHS's workload distribution recommendation for each graph input.

Graph ID	Graph	Vertices	Edges	CC	D	DEG	SCC	GAHS's choice
G1	ca-coauthors-dblp	540486	15245730	0.8019	7	56	0.000002	(Config-2)
G2	cit-DBLP	12591	49744	0.1169	5	7	0.019061	(Config-1)
G3	cit-patent	3774768	16518948	0.0757	9	9	0	(Config-FPGA)
G4	email-enron-large	33696	180812	0.5092	5	11	0.00003	(Config-4)
G5	road-asia-osm	11950757	12711604	0.0006	1377	2	0	(Config-3)
G6	road-roadNet-CA	1957027	2760389	0.0465	442	3	0.000001	(Config-2)
G7	rt-retweet-crawl	1112702	2278853	0.0187	6	4	0.000001	(Config-4)
G8	socfb-A-anon	3097165	23667395	0.097	6	15	0	(Config-2)
G9	socfb-Duke14	9885	506438	0.2455	3	103	0.000101	(Config-FPGA)
G10	socfb-uci-uni	58790782	92208196	0.043	9	3	0	(Config-2)
G11	web-BerkStan	685230	7600596	0.5967	10	22	0.488678	(Config-2)
G12	web-google	1299	2774	0.3483	8	4	0.00077	(Config-3)
G13	web-it-2004	509338	7178414	0.816	8	28	0.000002	(Config-2)
G14	web-wikipedia2009	1864433	4507316	0.1596	9	5	0.000001	(Config-2)

Table 3.4: Sample of the testing data-set used for evaluating GAHS framework.

3.4.3 Total Execution Time Analysis

GAHS's workload distribution recommendations achieve up to 6.5× speedup compared to FPGA-only and CPU-only execution as shown is Figure 3.2. The CPU-only configuration is a multi-core implementation utilizing 24 cores in the Intel Xeon Gold 6128 CPU. Figure 3.2 shows that the workload distribution is beneficial in almost all of the test cases, with an average speedup of 5× across the 100 tested graphs. However, the test case (G4) where GAHS's recommendation causes a slowdown is discussed in Section 3.4.4.

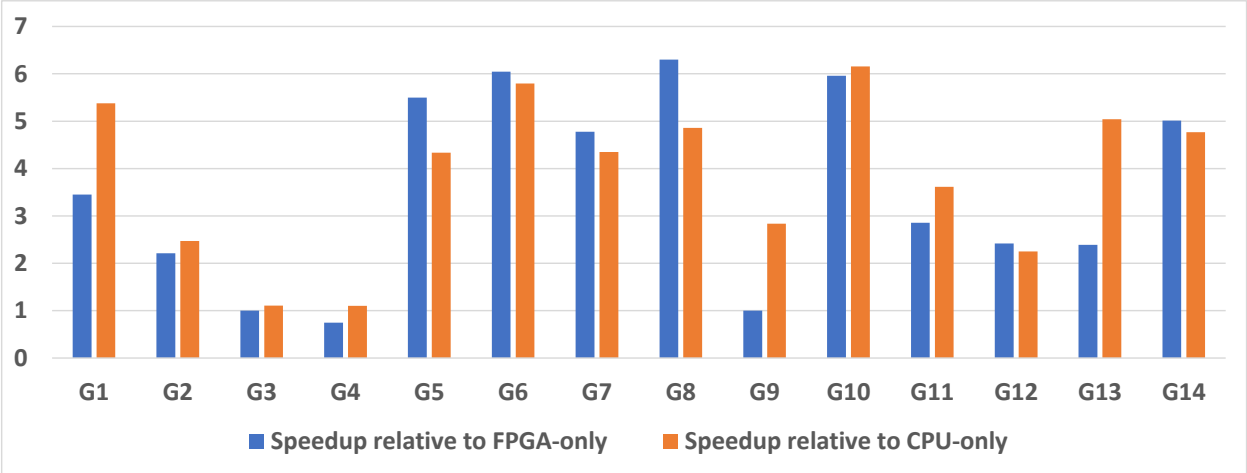


Figure 3.2: GAHS's recommendation speedup relative to FPGA- and CPU-only execution.

While Utilizing the available hardware resources and distributing the workload shows significant speedup; in some cases, as shown in Table 3.4, GAHS's recommendation was not to partition the workload. Instead, GAHS recommended utilizing the FPGA resource only. In this particular case, most of the kernels are expected to be more performant on the FPGA, while only one kernel was predicted to be more performant on the CPU. Initially, GAHS's decision is based on analyzing each kernel separately, then it evaluates the overhead coefficient. If the workload distribution overhead (depicted by the overhead coefficient) has more of an impact on performance than offloading the one kernel to the faster CPU (for this particular kernel), then all the computation is offloaded to the FPGA.

3.4.4 Recommendation Accuracy

Testing GAHS's recommendation accuracy was an elaborate job that included executing the entire testing data-set over all the configuration choices. Each graph is executed using all the available configuration choices to figure out which workload distribution configuration is the fastest for that graph. The fastest configuration is compared to GAHS's recommendation,

eleven out of the hundred tested graphs were not the most performant choice. This aggregates to a recommendation accuracy of 89%. However, interesting findings were revealed when studying inaccurate cases where GAHS failed. We find that although GAHS's choice was sub-par in some cases, it chooses a configuration with comparable performance to the best choice. Figures 3.3 and 3.4 show two sample cases of GAHS's miss-prediction.

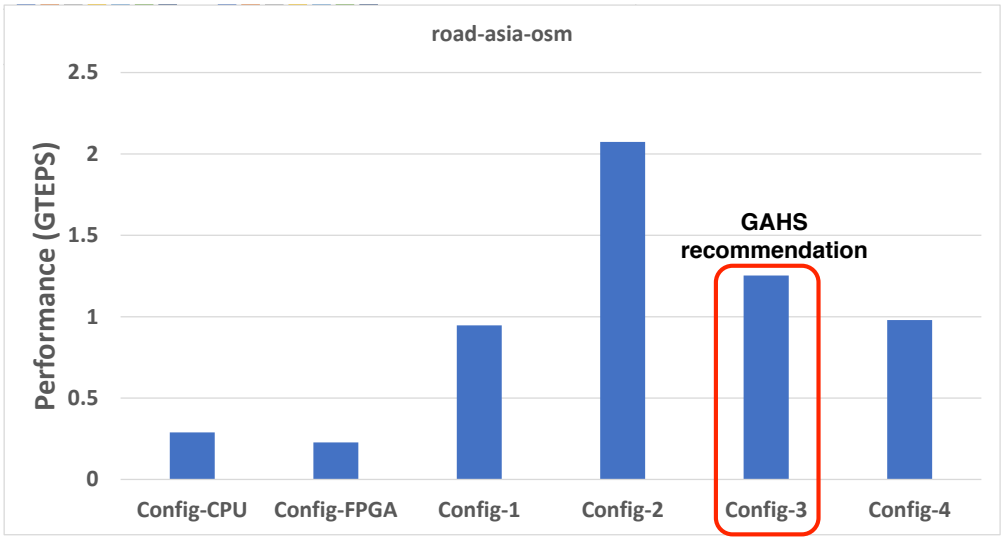


Figure 3.3: Performance (GTEPS: Giga Traversed Edges Per Second) of `road-asia-osm` graph across all workload distribution configurations.

Figure 3.3 shows the performance of `road-asia-osm` across all the configuration choices. The fastest configuration is `Config-2` while GAHS recommended `Config-3`. This is a 60% slowdown compared to the fastest configuration. However, GAHS's choice still showed a 4.3× speedup over CPU-only and 5.5× speedup compared to FPGA-only configuration. Hence, while GAHS mis-predicted the best workload distribution, it still improved performance.

On the other hand, Figure 3.4 shows performance of the graph `email-enron-large` across all configurations. In that case, GAHS recommended `Config-4`, while the fastest implementation was of `Config-FPGA`. This is attributed to a bad overhead coefficient calculation. Although each separate kernel would be faster using `Config-4`, there was a significant over-

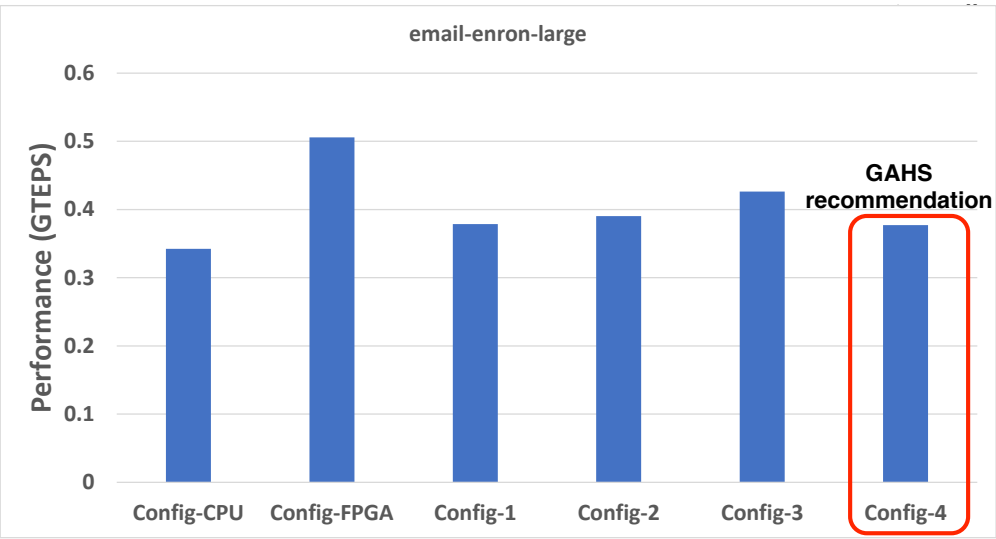


Figure 3.4: Performance (GTEPS: Giga Traversed Edges Per Second) of `email-enron-large` graph across all workload distribution configurations.

head of 22% when executing this graph using `Config-4`. The overhead was underestimated by GAHS causing it to avoid choosing `Config-FPGA` which offloads all the computation to the FPGA only.

3.4.5 Comparison with State-of-the-Art

Executing the graphs data-set on GAHS's recommended design configuration choice achieved a maximum absolute performance of **4.8 GTEPS**, with an average of **1.78 GTEPS**. This amounted to an average speedup of 18× compared to the state-of-the-art CPU-FPGA hybrid designs as shown in Figure 3.5.

GAHS is compared with state-of-the-art highly optimized hand-tuned designs that use CPU-FPGA hybrid systems proposed in [71, 75, 81, 87] as shown in Figure 3.5. The work in [71] achieved an average traversal speed of 172 MTEPS when executing synthetically generated graphs as explained earlier in Chapter 2. GAHS framework achieves 10× improvement in performance simply by distributing the workload. Moreover, while employing hand-tuned

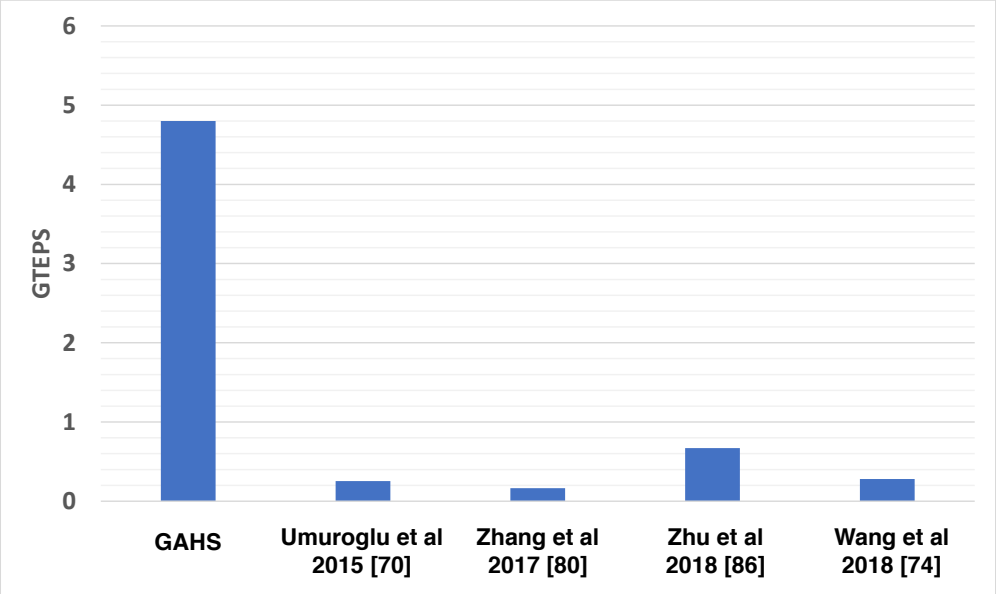


Figure 3.5: Maximum performance achieved by GAHS compared to related work

optimizations in [75], they achieved a maximum performance of approximately 280 MTEPS for sparse graphs. GAHS's workload distribution achieves 17.1× speedup compared to [75] without the need for tedious hand-tuned optimizations. Hence, GAHS outperformed their optimized solution by only efficiently distributing the workload.

3.5 Summary

This chapter presents a means of harnessing the computational power a CPU-FPGA hybrid platform. The performance of graph processing is enhanced using workload distribution over the available hardware resources in such a hybrid ecosystem. GAHS (Graph Analytics on hybrid Systems) is an analysis and recommendation framework that automatically decides the best workload distribution over the available hardware resources. The framework decides the best design configuration choice based on properties of the input graph and the hardware specification to select the workload distribution most suited for that specific graph input. Furthermore, GAHS learns as it goes. The framework includes a feedback loop that relays new graph properties and execution profiles back to GAHS's back-end. This feedback loop enhances the decision-making process of GAHS and makes it more accurate in choosing the best workload distribution.

GAHS shows significant performance improvement executing the PageRank application on a CPU-FPGA hybrid system. Up to 6.5× speedup was observed compared to CPU-only and FPGA-only implementations. It also shows an average of 18× improvement compared to state-of-the-art hybrid FPGA solvers, though these implementations usually include hand-tuned hardware optimizations.

Chapter 4

HLS Design Space and Optimization Exploration

When targeting execution on a heterogeneous platform, a portable programming language is essential to facilitate coding on all the available devices. OpenCL is considered for graph processing development in this dissertation as it can be easily used to program both CPU, and FPGA based platforms. While OpenCL is emerging as a high-level hardware description language, it may not yield the best FPGA designs. OpenCL addresses the productivity challenges of developing applications on FPGAs by providing an abstract interface to facilitate high productivity.

However, these OpenCL-realized accelerators are unlikely to make efficient use of the reconfigurable fabric without adopting FPGA-specific optimizations [55, 88], particularly for irregular OpenCL applications [26, 47]. Consequently, this chapter explores the FPGA-specific optimization space for OpenCL applications and presents insights on which optimization techniques improve application performance and resource utilization. Exploring this optimization space will enable end users to harness the computational potential of the FPGA using HLS tools.

4.1 Motivation & Goal

FPGAs have been used to accelerate a wide spectrum of applications, due to their superior power efficiency over general-purpose architectures such as CPUs and GPUs. However, these performance and power gains come at the cost of complex programming with hardware description languages (HDLs). OpenCL compilers for FPGAs were introduced to address this problem [3, 80].

4.1.1 Challenges

OpenCL Programming Model

Unlike HDLs, OpenCL provides an abstract machine model and high-level programming approach for reconfigurable architectures [12, 64], making it easier for end users to develop custom hardware accelerators for their applications and benefit from the power efficiency of FPGAs. Moreover, OpenCL employs a hierarchical memory structure with strong support for parallel execution. Hence, the parallelism can be specified at different granularity levels and data movement can be easily manipulated, enabling OpenCL compilers to potentially generate efficient FPGA designs.

The OpenCL programming model targets heterogeneous systems with different types of accelerators, including CPUs, GPUs, Intel MICs, DSPs, and FPGAs. While OpenCL provides *functional* portability across these accelerators, *performance* portability is not guaranteed. In particular, generic (architecture-agnostic) OpenCL kernels are unlikely to make efficient use of the FPGA resources, which leads to performance degradation. This has been shown for multiple application domains such as dense linear algebra, structured grid, unstructured grid, dynamic programming, and N-Body [7, 17, 39, 40, 47, 55, 77, 88]. Moreover, existing OpenCL codes that target CPUs and GPUs are not directly applicable to FPGAs, due to

the different hardware capabilities and execution models.

FPGA Optimization

The FPGA-specific optimizations proposed in this chapter are general and applicable to any application; however, the expected performance gain and resource-utilization efficiency vary depending on the application characteristics. In particular, we pursue the more challenging problem of irregular OpenCL applications, which suffer from workload imbalance, unpredictable control flow, and irregular memory-access patterns.

Irregular Applications

Irregular applications, such as graph processing, typically achieve a small fraction of the peak performance on general-purpose architectures due to their workload imbalance, unpredictable control flow, and irregular memory-access patterns. As a consequence, they have the potential to benefit from acceleration using custom hardware architectures. However, when targeting irregular applications, identifying which optimization (or combination of optimizations) to use to enhance the performance and resource utilization on FPGAs is challenging.

4.1.2 Approach

Experiments using representative kernels from the graph traversal, combinational logic, and sparse linear algebra application domains show that FPGA-specific optimizations can improve the performance of irregular OpenCL applications by up to 27-fold in comparison to the architecture-agnostic OpenCL code from the OpenDwarfs benchmark suite. Specifically, hardware profilers are used to analyze the limitations of OpenCL application kernels and to

guide the development of FPGA-optimized implementations.

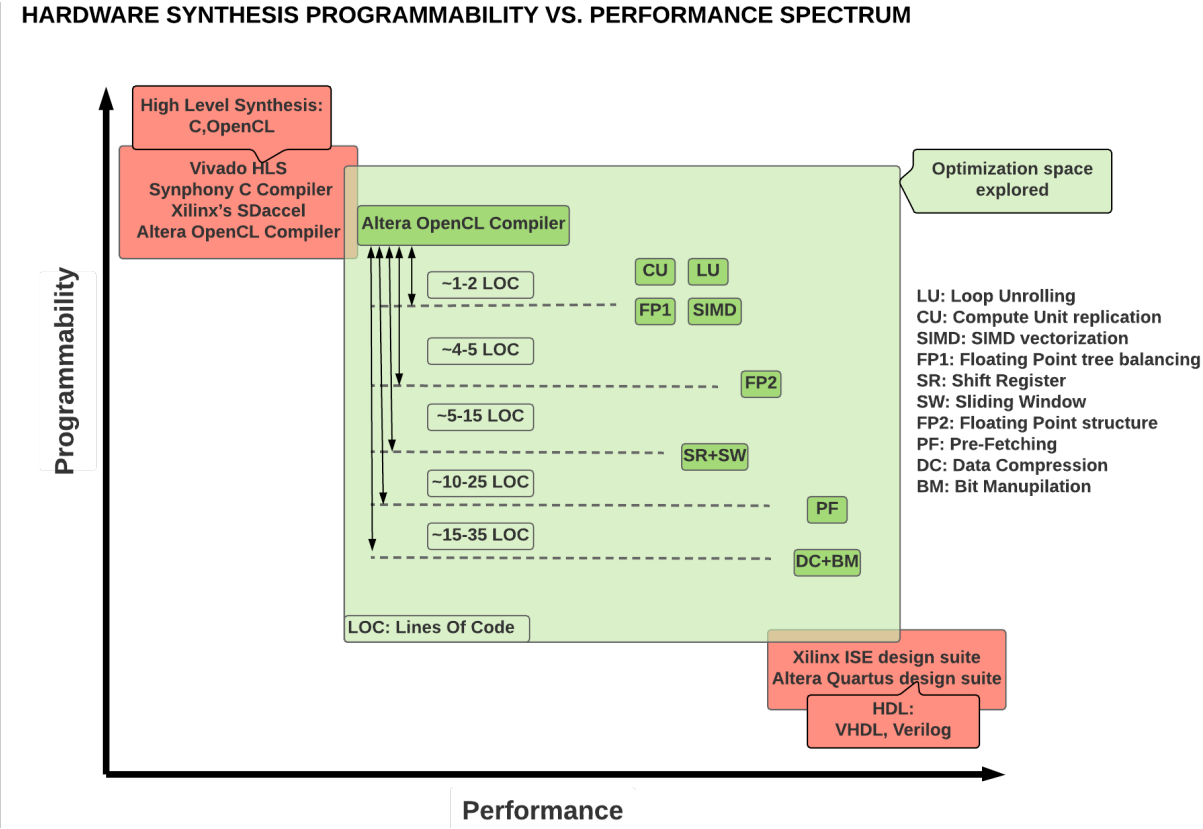


Figure 4.1: Programmability vs. performance spectrum for FPGAs.

Figure 4.1 illustrates the performance versus programmability spectrum and shows the design-space exploration and how FPGA-specific optimizations can be used to enhance the performance with little impact on OpenCL programmability. While the figure is not drawn to scale, it shows the additional programmability overhead (in terms of the average lines of code used to apply the optimization) with respect to the expected performance gain. The optimization techniques depicted in Figure 4.1 are explained later in Section 4.2.

4.1.3 Contributions

This study aims to extract the best possible performance of irregular applications on FPGAs using high-level OpenCL programming model. Moreover, it guides non-expert users to the appropriate FPGA-specific optimizations for irregular application domains, including graph traversal, combinational logic, and sparse linear algebra.

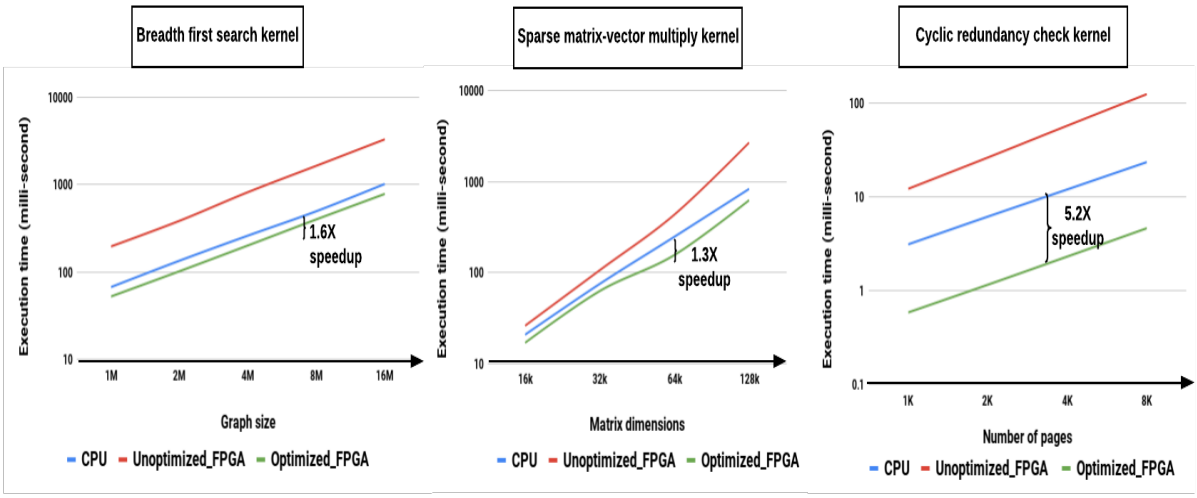


Figure 4.2: The performance of irregular OpenCL kernels on CPU and FPGA architectures. The optimized FPGA execution uses a deeply-pipelined, compute unit running at 200-260 MHz, while the CPU platform consists of 16 compute units running at 3.5 GHz.

Figure 4.2 illustrates the benefits of applying such optimizations to generate efficient accelerators on FPGA architectures. For a set of irregular OpenCL kernels, the optimized FPGA designs achieve up to 5.2× speedup over the corresponding parallel execution on a 16-core CPU, while running at an order-of-magnitude slower frequency.

This study aims to enhance the performance of irregular applications such as graph processing when using a high-level portable programming model like OpenCL. This goal is achieved by conducting the following three phases:

- *Identification of and insight on the FPGA-specific optimizations for OpenCL kernels.* The identified optimizations apply to any user application; however, the expected performance gain and resource utilization depend on the characteristics of the application kernels. Furthermore, end users need insight on which optimizations to use for their target applications.

- *Profiling and analyzing of the OpenCL kernels to identify the execution bottlenecks and, in turn, guide the FPGA optimization of irregular codes.* We use Intel/Altera's hardware profiler to facilitate the analysis and optimization of irregular application kernels from the OpenDwarfs benchmark suite.

- *A detailed study of the FPGA-specific optimizations for representative irregular applications, namely graph traversal, combinational logic, and sparse linear algebra applications.* The results show that the FPGA-specific optimizations improve performance by an order of magnitude when compared to the architecture-agnostic OpenCL code from OpenDwarfs.

4.2 Background

We categorize the FPGA-specific optimization space as follows: (1) exploiting parallelism at different levels, (2) optimizing floating-point operations, and (3) minimizing data movement across the memory hierarchy. By default, FPGA OpenCL compilers exploit deep pipeline parallelism, which, in turn, generally achieves higher throughput than data parallelism or task parallelism on FPGAs. Data and task parallelism are constrained due to the limited resources on FPGAs, which restrict the number of concurrently active work items.

4.2.1 Parallelism Optimizations

There are two main OpenCL execution models on FPGAs: multi-threaded execution and single-task execution. Multi-threaded execution attempts to expose the maximum parallelism by executing multiple threads concurrently, if possible. On the other hand, single-task execution exploits pipeline parallelism and runs the work items (i.e., units of computation) sequentially as a single task. The baseline, architecture-agnostic OpenCL kernels from OpenDwarfs are based on the multi-threaded execution model. However, both single task and multi-threaded execution models are explored in this work.

Loop Unrolling. Unrolling loops improves performance by decreasing the number of loop iterations executed and, in turn, the number of branches. However, there is a trade-off between the loop unrolling factor and the extra hardware cost incurred.

Kernel Vectorization. Vectorization enables multiple work items to execute in a single-instruction, multiple-data (SIMD) fashion. This technique achieves *higher computational throughput* and automatically performs *memory coalescing*. The SIMD approach vectorizes the data path of the kernel while keeping a single control logic path shared across the SIMD lanes. Therefore, backward branches with thread ID dependencies prohibit this optimization technique, as they can serialize the execution process.

Compute Unit Replication. Generating multiple compute units, where data and control paths are replicated, fully parallelizes the kernel execution. This optimization divides the workload on the available compute units which can mitigate the limitations of the SIMD approach, namely the thread ID dependency problem. However, compute unit replication uses more hardware resources than the SIMD approach. It also increases the stress on the global memory bandwidth, as more load/store units would be competing for accessing the global memory.

4.2.2 Floating-Point Optimizations

The floating-point operations in a specific kernel may not be balanced, leading to pipeline stalls and higher hardware cost [2]. The Altera OpenCL Compiler provides command-line options to optimize the floating-point operations using balanced trees. Moreover, removing the floating-point rounding operations and conversions, whenever possible, introduces hardware savings. While the amount of hardware resources employed for the floating-point operations can be reduced, the accuracy of the results might get affected. Therefore, these optimizations are used only when the application can tolerate small differences in the floating-point results.

Floating-Point Accumulator. The newer FPGA platforms, such as Altera's Arria 10, include a floating-point accumulator that performs the accumulations in a single cycle; however, only single work-item kernels that perform accumulation in a loop without branching can leverage this feature. Modifications are required in the kernel code for the compiler to infer the use of the accumulator structure. The accumulator must be part of a loop, it must have an initial value of 0 and it cannot be conditional.

```
1 int i;
2 float acc = 0.0f;
3 for (i = 0; i < k; i++)
4   acc+=5;
5 a[0] = acc;
```

As shown in the code example above, the accumulator must be part of a loop, it must have an initial value of 0 and it cannot be conditional.

4.2.3 Data Movement Optimizations

Shift Registers (SR) and Sliding Windows (SW). Several computational kernels, such as sparse matrix-vector multiplication (SPMV), have loop-carried data dependencies. On FPGA architectures, cross-iteration dependencies may increase the initiation interval of the loop, where the next iteration is stalled until the dependency is resolved. To relax this cross-iteration dependency, the loop body is modified to employ shift registers with a sliding-window technique, which resolves this problem by eliminating the pipeline stalls.

Data Compression (DC) and Bit Manipulation (BM). The OpenCL standard instantiates Boolean variables as 32-bit integers. Programming bit-wise operations and masks allows single-bit on-chip memory (BRAM/register) access by the OpenCL code.

4.3 Evaluation & Analysis of Irregular Applications

The Intel FPGA Dynamic Profiler for OpenCL was used to analyze the execution profile of the architecture-agnostic (generic) OpenCL kernels from the OpenDwarfs suite [41]. Analyzing the execution profile pinpoints the bottlenecks of the execution pipeline. This study applies the above optimizations, both in isolation and in combination, to the target OpenCL kernels and evaluates the resulting performance, which typically outperforms the architecture-agnostic and GPU-optimized OpenCL implementations on FPGA architectures. The FPGA resource utilization is also considered in evaluating the hardware cost of each optimization technique. Finally, the performance analysis of the different optimizations provides key insights into which optimizations to use for each target application and how to apply such optimizations to address the execution bottlenecks and to achieve the required performance gain. The optimized kernels are available at *https://github.com/vtsynergy/OpenDwarfs*.

Experimental Setup: The experiments use an Altera Arria 10 1150-GX FPGA connected to two 4-GB DDR3 memory with peak bandwidth of 25 GB/s. The FPGA attaches to the host machine via PCIe 8x 3.0 interface. The host includes an Intel Xeon E5-2637 CPU and runs Ubuntu 14.04 along with Altera OpenCL SDK version 16.0.

4.3.1 Graph Traversal

Breadth-first search (BFS) is used by the OpenDwarfs suite as a representative kernel for the graph traversal dwarf. The target graphs are undirected and unweighted in the form $G = (V, E)$, where V is the set of vertices or nodes and E is the set of edges connecting them. To avoid processing a node more than once, a Boolean visited array is used. As such, the graph is traversed in levels, where all nodes at each level are explored before the next level is processed. The final output is the cost C, which represents the shortest distance from the source node to each visited node on the graph. The time complexity is $O(V + E)$.

The original OpenCL kernel (from OpenDwarfs) is multi-threaded, executing each graph level update in a separate kernel launch. After computing a level of the graph, synchronization with the host is required, followed by a new kernel launch for the new graph level. Our hardware profiling pinpointed two major bottlenecks. First, the cross-iteration dependencies stalled the execution pipeline for more than 800 clock cycles in *each* iteration. This high

Optimization	Description	Frequency	Logic utilization	BRAM
Generic	Architecture agnostic OpenDwarfs kernel	248 MHz	29%	20%
MT-LU8	Loop unrolling factor 8	205 MHz	37%	37%
MT-PE2	Compute unit replication (2 PEs)	204 MHz	32%	24%
MT-PE4	Compute unit replication (4 PEs)	202 MHz	35%	42%
ST-Regular	Simple conversion to single work item by inserting outer `for-loop`	201 MHz	28%	20%
ST-NoSync	Eliminate host side synchronization by inserting outer `while-loop`	212 MHz	29%	20%
ST-Mem	Use bit manipulation on integer arrays to implement Boolean arrays	160 MHz	27%	74%

Table 4.1: BFS optimizations and resource utilization. [MT]: Multi-Threaded, [ST]: Single Task.

initiation interval of the loop is caused by (a) the serial execution of the `for-loop`,where loop pipelining optimization isn't applied by default due to unrelieved cross iteration dependency and (b) the host side synchronization step between kernel invocations. Second, the global memory access pattern for five different arrays is inefficient, lowering the bandwidth efficiency of data transfer to an average of 13% of the peak memory bandwidth. Table 4.1 shows the different optimization techniques employed to address these bottlenecks, along with their operating frequency and logic utilization.

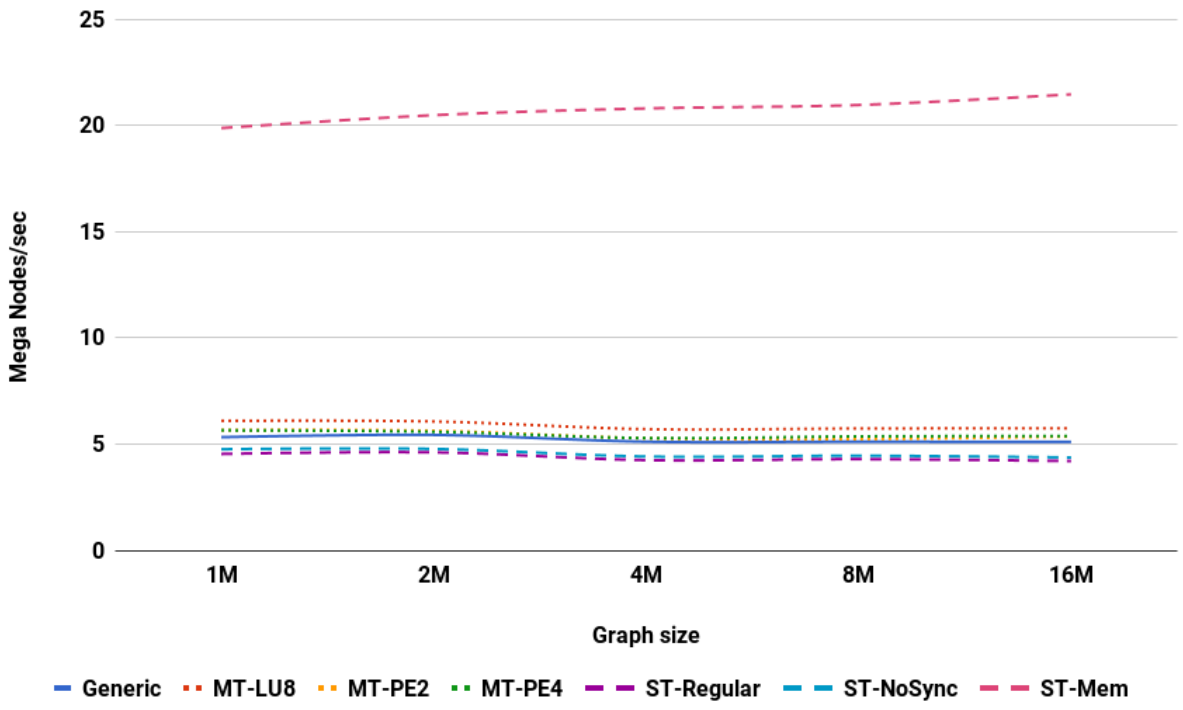


Figure 4.3: The performance of BFS (nodes processed per second) across different graph sizes for multiple optimization techniques.

Multi-Threaded Execution

The global memory access bottleneck and kernel launch overhead are the main reasons that none of the multi-threaded optimization techniques yielded any significant performance

gains. Compute unit replication does not address the memory access bottleneck or solve the pipeline stall problem. Hence, the performance improvement was 6% at most, as shown in Figure 4.3. Loop unrolling enhanced the performance by a maximum of 14%, due to memory coalescing which increases the bandwidth efficiency. The combination of loop unrolling with compute unit replication is unnecessary as neither have the potential to address the pinpointed bottlenecks. The multi-threaded model does not leverage data-level parallelism (vectorization), due to the loop-carried data dependencies and the thread ID dependent branching in its inner and outer loop. So, SIMD optimization was not applied to this application kernel.

Single Task Execution

In the single work-item execution model, multiple optimizations were tested to enhance performance. The "ST-Regular" implementation fully pipelines the `for-loop` without modifying the global synchronization scheme. On the other hand, "ST-NoSync" avoids synchronizing with the host, which accounted for an average 5% of the execution time, and moves all computations to the FPGA. The Altera OpenCL compiler was not able to pipeline the outer `while-loop` in this implementation, as cross iteration dependency is critical for functional correctness. Figure 4.3 shows a slight decrease in performance for "ST-Regular" and "ST-NoSync", as these optimizations do not address the memory access bottleneck, which has the most impact on performance.

The following step was to optimize the memory access operations by moving the Boolean arrays to the local on-chip memory (BRAMs) in "ST-Mem". The OpenCL standard supports Boolean variables; however, they are treated as 32-bit integers with a constant value of "0" or "1", which wastes the on-chip memory. Therefore, integer arrays are used, as shown in Figure 4.5, where each integer represents 32 Boolean flags that can be accessed through a

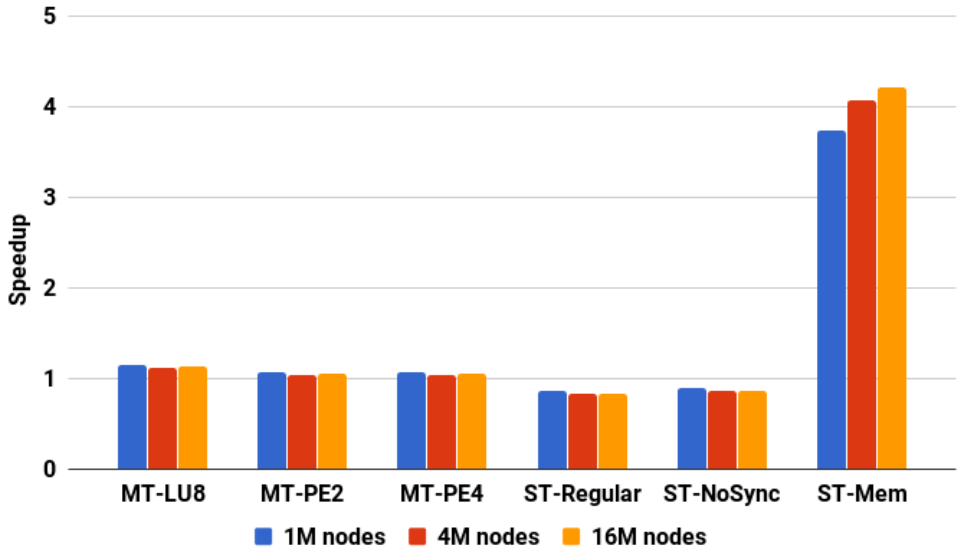


Figure 4.4: BFS speedup across the different optimization techniques. The baseline is the OpenDwarfs, architecture-agnostic OpenCL code.

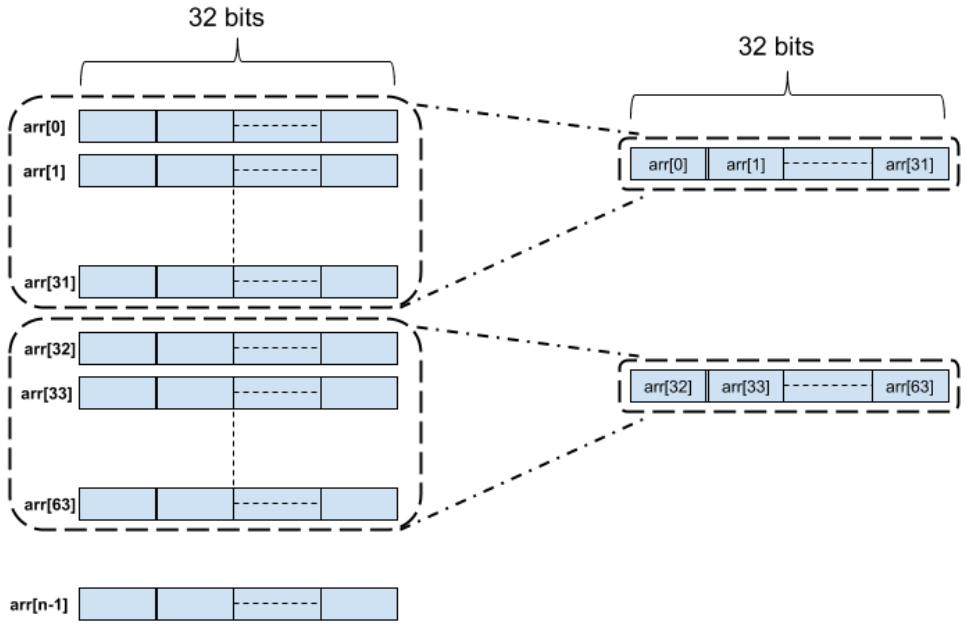


Figure 4.5: Boolean array data compression.

series of bit-wise manipulations (shift, AND, OR and XOR operations). This optimization technique enables fast Boolean checking which yielded 4× speedup as shown in Figure 4.4. However, due to the limited on-chip memory, this approach can only support graphs of sizes up to 32M nodes.

For smaller graphs with size up to 512K nodes, the algorithmic refactoring showed great performance. The kernel was modified to use a local-memory queue instead of the Boolean mask. The new unvisited nodes are inserted into the FIFO queue, and one node is popped in each iteration, which greatly reduces the total number of iterations. However, this evaluation targets large-scale graphs with at least 1M vertices; hence, this approach was excluded from the results.

4.3.2 Sparse Linear Algebra

The OpenDwarfs suite includes SPMV (sparse matrix-vector multiplication) as a representative kernel of sparse linear algebra. While computation across the rows of the input sparse matrix (outer loop) are independent, the operations required to compute a single output element (inner loop) have data dependencies. A series of memory accesses are required by each iteration of the outer loop to retrieve the indices of non-zero elements of each sparse row, and read the respective values of these elements and the corresponding elements of the input vector. On hardware architectures with limited memory bandwidth, such memory operations introduce a global memory bottleneck. The hardware profiler showed that bandwidth efficiency is limited to 55% at the bottlenecked inner loop. Moreover, the number of iterations in the inner loop is input dependent (i.e., depends on the sparsity pattern of the input matrix).

Optimization	Description	Frequency	Logic utilization	BRAM
Generic	Architecture agnostic OpenDwarfs kernel	255 MHz	26%	21%
MT-LU16	Loop unrolling factor 16	225 MHz	33%	31%
MT-LU32	Loop unrolling factor 32	211 MHz	39%	46%
MT-PE2	Compute unit replication (2 PEs)	245 MHz	28%	25%
MT-PE4	Compute unit replication (4 PEs)	230 MHz	31%	36%
MT-PE2-LU16	Compute unit replication (2 PEs) + Loop unrolling factor 16	203 MHz	41%	47%
MT-PE2-LU32	Compute unit replication (2 PEs) + Loop unrolling factor 32	174 MHz	54%	77%
MT-PE4-LU16	Compute unit replication (4 PEs) + Loop unrolling factor 16	164 MHz	57%	80%
ST-PF-SR-LU8	Pre-fetching+SR and sliding window+Loop unrolling factor 8	205 MHz	41%	50%
ST-PF-SR-LU12	Pre-fetching+SR and sliding window+Loop unrolling factor 12	192 MHz	51%	66%
ST-PF-SR-LU8-LU4	Pre-fetching+SR and sliding window+LU8(outer loop)+LU4(inner loop)	166 MHz	57%	69%

Table 4.2: SPMV optimizations and resource utilization. [MT]: Multi-Threaded, [ST]: Single Task.

Multi-Threaded Execution

The multi-threaded version of SPMV exploits different parallelism levels: task-level (compute unit replication), and instruction-level (loop unrolling). However, the multi-threaded code does not leverage data-level parallelism (vectorization), due to the loop-carried data dependencies and the thread ID dependent branching in the inner loop.

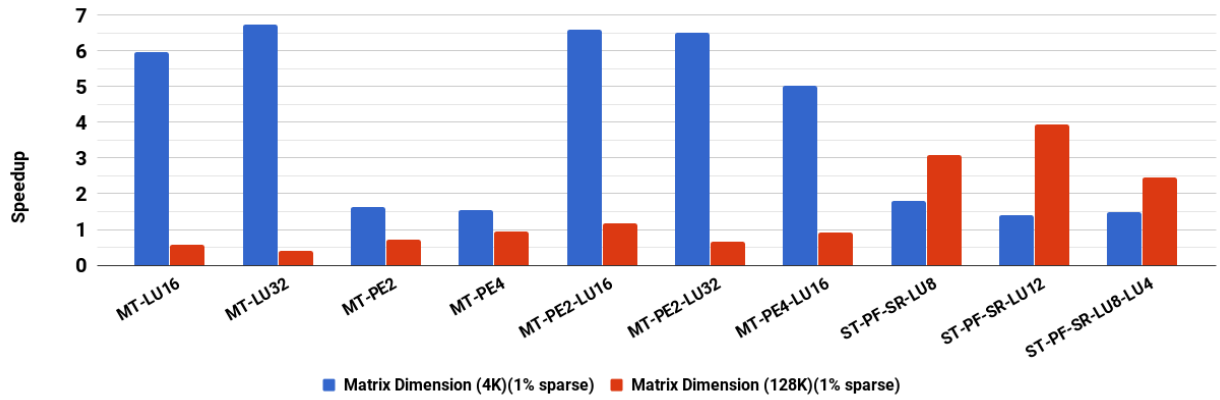


Figure 4.6: SPMV speedup for small and large matrix sizes.

Figures 4.6 and 4.7 show the effect of the different optimizations on the SPMV performance in comparison with the baseline OpenCL code, while Table 4.2 shows the resource utilization of each implementation. For the small input data, these optimizations showed a maximum speedup of 6.8× (matrix size 4K in Figure 4.6). "MT-LU32" provides the best performance

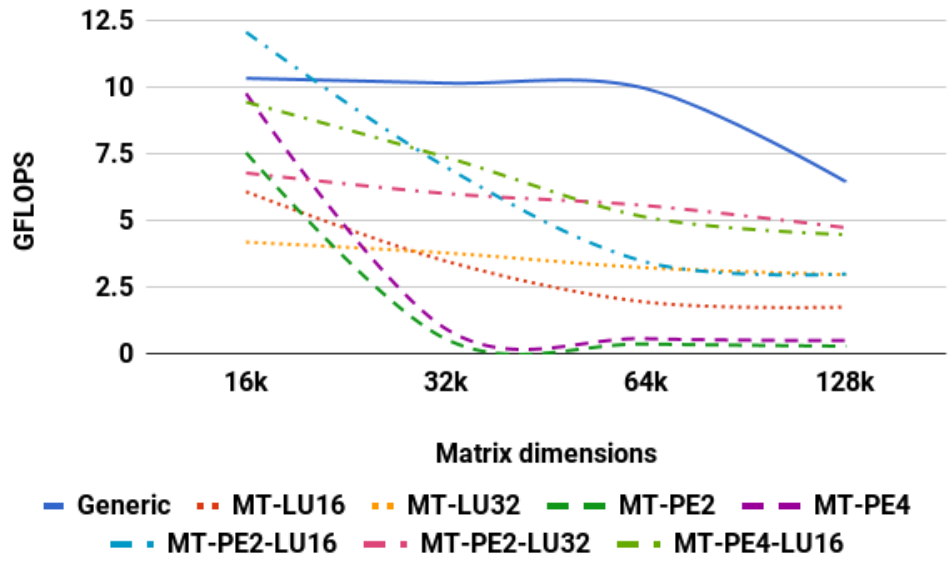


Figure 4.7: The performance of SPMV for the multi-threaded optimizations. The baseline is the OpenDwarfs, architecture-agnostic code.

only with 13% more logic utilization than the baseline kernel, but with double the on-chip memory usage. Compute unit replication has limited performance improvement, it achieved at most 1.7× speedup by "MT-PE2" and "MT-PE4". This is mainly due to the contention on the limited global memory bandwidth. Combining both optimizations of multiple compute units and loop unrolling shows comparable performance to simply just unrolling the loop, but with much higher hardware cost. "MT-PE4" increases the logic utilization by 30% and the on-chip memory usage by 60%.

Figures 4.6 and 4.7 show the limited scalability of the multi-threaded execution model for SPMV. Scaling up the input matrix puts the multi-threaded SPMV at a great disadvantage, due to the limited global memory bandwidth. In fact, these optimizations do not address the issue of the bandwidth efficiency of the inner loop. So, as the matrix size increases these optimization add more stress to the global memory access and the performance decreases and becomes slower than the baseline code.

Single Task Execution

The single task (work item) execution of SPMV throttles the parallelism (concurrent work items) to reduce the contention on the limited FPGA resources, specifically the global memory bandwidth. However, due to the extra unused FPGA resources, the single task code can leverage advanced techniques to minimize the data movement across the memory hierarchy, such as caching (pre-fetching), shift registers, and enhanced floating-point units. The caching optimization pre-fetches the data into the private memory (BRAM) and maximizes the data reuse. The shift register (SR) optimization uses a sliding window technique to alleviate the loop-carried dependencies in the SPMV inner loop, which enables the compiler to efficiently pipeline the inner loop with successive iterations initiated into the pipeline every clock cycle. Finally, the code was modified to allow the compiler to infer floating-point accumulator (see section 4.2.2) to further enhance the performance of the inner loop.

These techniques relieved the inner loop contention on global memory access increasing the bandwidth efficiency to 100% (according to the hardware profiler). However, the hardware profiler showed that this execution model limits the efficiency of the store unit that writes the final result at the end of each outer loop iteration to 20%. Nevertheless, the effect of this limitation has much less impact on performance, since the number of store operations is significantly less than the number of load operations. The number of load operations is $O(NZE)$, where NZE is the (number of Non Zero Elements). The number of store operations is $O(R)$, where R is the (number of matrix Rows).

Figures 4.6 and 4.8 show the performance of the single task execution compared to the baseline and the multi-threaded versions. Unrolling the outer loop and minimizing the data movement in "ST-PF-SR-LU12" showed a speedup of 4× over the baseline kernel. Moreover, single task execution has scalable performance and sustainable speedup, as depicted in Fig-

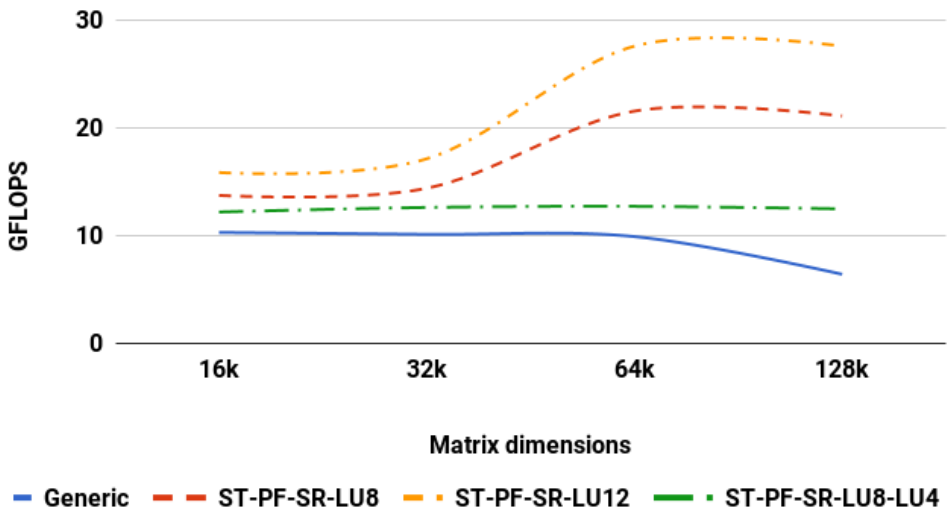


Figure 4.8: The performance of the single-task optimizations for SPMV. The baseline is the OpenDwarfs, architecture-agnostic OpenCL code.

ure 4.8, with input matrix size up to 128K. Unrolling the inner loop in "ST-PF-SR-LU8-LU4" didn't provide performance advantages, due to inefficient pipeline structure in addition to the input dependent number of inner loop iterations (loop bounds are not constants). The Altera OpenCL Compiler might fail to meet scheduling because it cannot unroll this nested loop structure easily, resulting in a high II (number of stall clock cycles before issuing the next loop iteration) [2, 3]. In summary, the results showed the importance of alleviating the contention on the limited FPGA global memory bandwidth and inferring an efficient pipeline structure to attain scalable performance.

4.3.3 Combinational Logic

The OpenDwarfs adopts cyclic redundancy check (CRC) as a representative kernel of combinational logic applications, which rely on bit-wise logic operations. This application domain is amenable to acceleration using FPGA architectures with fine-grain logic fabric. The CRC

Optimization	Description	Frequency	Logic utilization	BRAM
Generic	Architecture agnostic OpenDwarfs kernel	270 MHz	26%	18%
MT-LU8	Loop unrolling factor 8	236 MHz	32%	23%
MT-PE2-LU8	Compute unit replication (2 PEs) + LU8	230 MHz	40%	31%
MT-PE4-LU8	Compute unit replication (4 PEs) + LU8	230 MHz	56%	47%
MT-SIMD2-LU8	SIMD (2 vector lanes) + LU8	227 MHz	39%	29%
MT-SIMD4-LU8	SIMD (4 vector lanes) + LU8	224 MHz	52%	44%
MT-SIMD8-LU8	SIMD (8 vector lanes) + LU8	203 MHz	78%	89%
ST-Regular	Simple conversion to single work item by inserting outer `for-loop`	148 MHz	41%	30%
ST-PF-SW	Pipelining `for-loop` by pre-fetching+relaxing cross iteration dependency	231 MHz	26%	18%

Table 4.3: The CRC optimizations and their resource utilization. [MT]: Multi-Threaded, [ST]: Single Task.

kernel computes the 32-bit CRC code of a set of input data pages (packets) using the "Slice-By-8" algorithm developed by Intel [35, 38]. The CRC32 generation process consists of a single table lookup, bit-wise and shift operations for each byte. The hardware profiler showed that there is 67 clock cycles of stalls in the pipeline execution for each loop iteration, due to inefficient loop structure.

Multi-Threaded Execution

Figure 4.9 shows the performance of the different multi-threaded versions in comparison with the baseline, architecture-agnostic code. The results show that fully-unrolled loops in "MT-LU8" yield around 2× speedup compared to the baseline with minimal hardware cost (6% logic utilization increase).

Exploiting task-level parallelism in "MT-PE2-LU8 and MT-PE4-LU8" achieves 4-5× speedup with additional hardware cost of 14% and 30% for two and four processing elements (PEs) respectively as shown in Table 4.3. However, Figure 4.9 shows that the compute unit replication approach does not provide scalable speed up, due to the contention on the global memory access. In particular, as the number of input data pages increases, beyond 1K, the performance degrades and converges to a consistent 2× speedup.

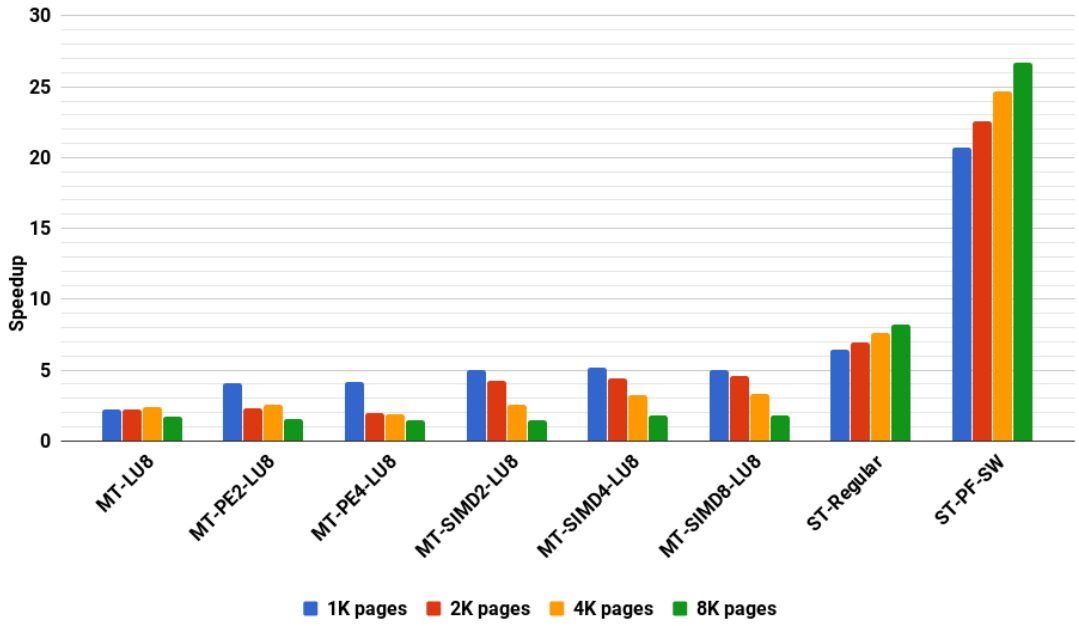


Figure 4.9: CRC speedup across different optimization techniques. The baseline is the OpenDwarfs, architecture-agnostic OpenCL code.

SIMD vectorization shows a speedup of 6× over the baseline in "MT-SIMD2-LU8"; however, increasing the number of vector lanes does not improve the performance over using two SIMD lanes, while incurring up to 52% additional area overhead. The increased hardware cost for using more vector lanes suggests that using two lanes would be the best option. SIMD vectorization works well with this specific kernel, as there is no thread ID dependent, backward branching.

Although the multi-threaded execution model shows some performance gain, it can be noticed in Figure 4.10 that, as the input size grows larger, the performance advantages degrade. The above multi-threaded execution versions suffer from a major bottleneck: the limited global memory bandwidth, where multiple threads in flight are competing for global memory access. Therefore, as the size of input data increases (more than 4K data pages), the performance takes a severe hit and the speedup decreases to 1.5-2× over the baseline.

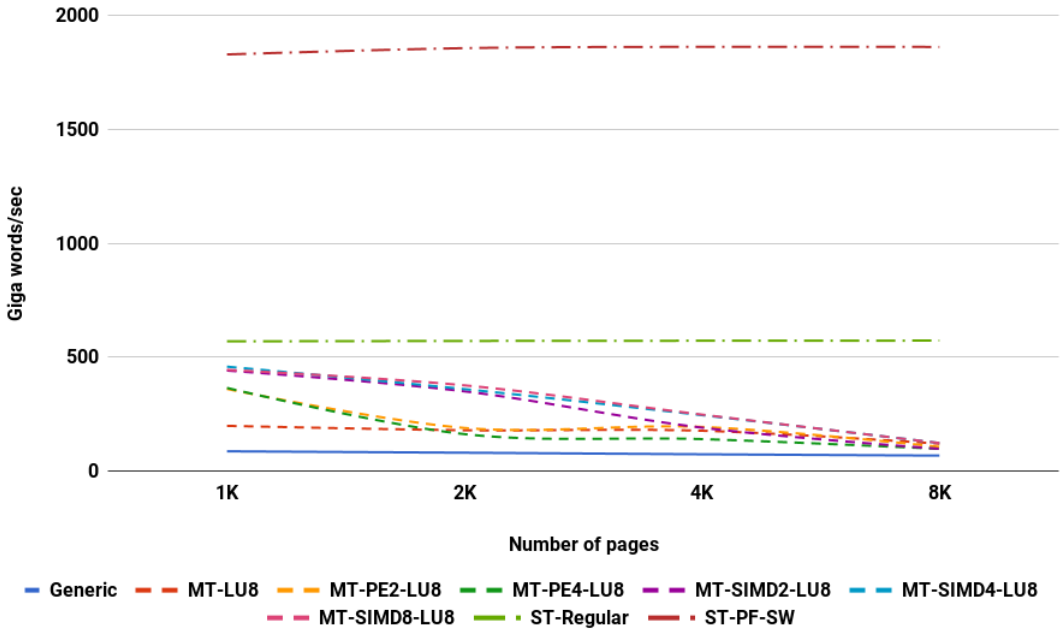


Figure 4.10: The performance (words processed per second) of CRC across different optimization techniques.

Single Task Execution

Moving to the single task execution model alleviated the problem of having multiple threads competing for the limited global memory bandwidth. Figure 4.9 shows that the single task CRC versions achieved scalable speedup with the growing input size. Moreover, Figure 4.10 shows that "ST-Regular" and "ST-PF-SW" process a constant number of words per second, that does not decrease. The execution profile pulled from the hardware profiler shows that the efficient pipeline execution with no stalls allows high bandwidth efficiency for each loop iteration. Memory accesses are pipelined and latency is efficiently hidden by the computation in each loop iteration, sustaining scalable performance.

Hence, unlike the multi-threaded execution model, the single task execution is scalable, i.e., its speedup improves as the input data size increases. After inspecting the FPGA execution

profile, further advanced optimizations were not needed, as the computational loops are efficiently pipelined with a loop iteration issued every two clock cycles (according to the compiler optimization report).

4.4 Discussion

The results across three representative irregular OpenCL applications showed that the limited global memory bandwidth of the FPGA architecture hinders the scalability of the multi-threaded OpenCL execution model. When the input data set is small enough to not increase the contention on the global memory access, the multi-threaded execution model can provide significant gains (e.g., up to 6.8× performance gains in SPMV kernel). However, with larger input data, the performance of the multi-threaded kernels is severely affected which leads to saturated and limited speedup. On the other hand, the single task execution model resolves the global memory bottleneck and enables streamed memory access to/from the main memory without competition between multiple threads.

Figure 4.2 shows the FPGA performance relative to the execution on a multi-core CPU platform. The CPU platform includes an Intel Xeon E5-2637 with 16 compute units running at 3.5 GHz, and it has a memory bandwidth of 80 GB/s and a cache size of 15 MB. The performance of the architecture-agnostic OpenCL kernels on FPGA (Unoptimized_FPGA) is an order of magnitude slower than the CPU execution. Even though the significantly higher frequency and larger number of compute units of the CPU platform put the FPGA platform at a great disadvantage, applying the FPGA-specific optimizations yields sustainable speedups over the CPU execution. While the optimized FPGA implementation runs at a frequency range of 200-260 MHz and uses a single deeply-pipelined compute unit, it achieves 5.2×, 1.3× and 1.6× speedup for the CRC (ST-PF-SW), SPMV (ST-PF-SR-LU12),

and BFS (ST-Mem) kernels, respectively, compared to the CPU execution.

The experiments show that applying FPGA-specific optimizations to the architecture-agnostic OpenCL code can significantly enhance the performance. Exploring the aforementioned optimization space, code patterns were identified according to the hardware profiler and the optimization reports. The effects of the optimizations on these code patterns are analyzed aiming to help in two aspects. First, providing guidelines and best practices for the development of new OpenCL kernels with similar code patterns towards the best performance. Second, guiding the future work of automating the optimizations process of architecture-agnostic OpenCL kernels. Below is a list of the identified code patterns along with their relative FPGA-specific optimizations.

- *OpenCL kernels with Boolean data structures that reside in the global memory* can be optimized using data compression techniques and bit-mask arrays to reduce the memory usage and to be able to place such arrays in on-chip BRAMs, allowing fast Boolean array look up.

- *Kernels that use conditional statements depending on a global memory read transaction* should be handled using pre-fetching of the conditional variable to the on-chip local memory to enable fast conditional checking.

- *Using DEF-USE chain analysis, loop carried dependencies can be detected*, and then the performance can be improved by relaxation using shift registers and sliding window operation, or by elimination using temporary on-chip storage before offloading the results to the corresponding output.

- *Floating-point accumulation can be easily detected in the code* and modified for optimization by balancing the floating-point operations tree and/or inferring floating point accumulation structures (see section 4.2.2).

- *Loop unrolling factor is critical for performance.* The unrolling factor should be closely coupled to the expected number of iterations of the loop. Unrolling a loop with a higher than necessary value would waste space (area) and time (frequency), which might lead to performance degradation.

4.5 Summary

This work discusses the FPGA-specific optimization space for the OpenCL programming model, with a specific focus on the irregular applications that suffer from workload imbalance, fine-grain bit-wise operations, dynamic control flow, and scattered memory access pattern. Applying such optimizations enables the OpenCL kernels to deliver both functional *and* performance portability on the FPGA architectures by synthesizing more efficient hardware designs. Moreover, hardware profiling was used to pinpoint the execution bottlenecks and to guide the optimization process. A detailed analysis of the FPGA-specific optimizations on the target application domains is provided to guide the end users to extract high performance from the energy-efficient reconfigurable architectures.

The experiments showed the potential of the single task execution model to resolve the contention on the shared FPGA resources and demonstrated *scalable* speedup on the three tested application domains. Specifically, the Breadth first search (BFS), cyclic redundancy check (CRC), and sparse matrix-vector multiplication (SPMV) applications achieved up to 4.2×, 27×, and 6.8× speedup, respectively, over the architecture-agnostic kernels from the OpenDwarfs benchmarks suite.

While significant performance improvements were obtained using the FPGA-specific optimizations of the original algorithms, previous studies [30, 33, 34] showed that algorithmic refactoring can result in multiplicative performance gain. As such, there are many opportu-

nities to expand the current work by analyzing and modeling the inherent characteristics of the different algorithms [31, 32] to guide the algorithmic innovation and refactoring of the irregular applications to better match the capabilities of the FPGA platforms. This work gives FPGA designs a fair chance when being evaluated alongside other hardware accelerators in a heterogeneous platform, such as many core CPUs and GPUs. specifically when targeting data-driven graph processing, FPGAs can provide a significant performance advantage. However, high-level approaches such OpenCL programming may not provide the best performance on FPGAs without FPGA-specific optimizations. Since OpenCL is used to address a heterogeneous ecosystem for graph processing, OpenCL-based FPGA designs should be optimized.

Chapter 5

Domain-Specific Performance Modelling for FPGA designs and automated optimization space exploration

High Level Synthesis (HLS) tools [2, 36, 80] significantly improved the ease of use of FPGAs and allowed it to be accessible to the masses. Since then, FPGAs' potential has been maximized making it integral in high performance computing facilitating its integration into mainstream computing. Nevertheless, it has been established that FPGA-specific optimizations are required to achieve the true potential of such a customizable architecture. Since the conception of HLS tools, there has been a myriad of research attempts to optimize kernels implemented in a High Level Language (HLL) such as OpenCL [7, 17, 26, 39, 40, 47, 55, 71, 75, 77, 82, 88]. However, performance modelling research was not advancing at the same pace. Performance modelling of HLS realized hardware designs has been mainly focused on regular applications with uniform memory access patterns. These performance models fail to accurately capture the performance of graph applications with irregular memory access patterns. This chapter presents a domain-specific performance model targeting graph applications synthesized using HLS tools for FPGAs.

5.1 Motivation & Goal

Modelling the performance and exploring the optimization space of graph applications is extremely challenging [27]. The use of general-purpose performance models to capture the execution profile of graph applications proved to be a futile endeavor as explained previously in Chapter 2. Hence the work presented in this chapter explains the development of a novel domain-specific performance model for graph application when targeting HLS designs on FPGAs.

5.1.1 Challenges

The main challenge is that graph applications have a data-driven execution profile that is hard to capture using an analytical performance model. In many instances, the processing requirements of a given graph computation are heavily data-dependent with a major impact on the computational solution and in turn the resulting performance. Performance models previously developed to target HLS designs may display a measure of accuracy when tested against regular applications such as dense linear algebra. The uniform memory access pattern of these applications make it easy enough to predict performance. However, in the case of irregular applications, these performance models fail to accurately capture the expected performance.

Moreover, not only performance modelling is of concern, exploring the optimization space is also a major concern. Many optimization techniques can be applied to the hardware design such as loop unrolling to make use of pipeline parallelism, and replicating compute units to make use of data parallelism. For an FPGA developer to test the efficacy of these optimizations, one has to spend days synthesizing multiple designs with various optimization techniques. Many FPGA developers wish for an answer for some optimization questions

without the need to fully synthesize many design optimizations. Some of these questions are: would loop unrolling enhance or degrade performance? How many compute units would achieve the best performance without overloading the memory interface? The answers to these questions may be straightforward for regular applications. However, for graph applications, these answers are directly tied not only to the hardware specifications, but also to the graph input characteristics.

5.1.2 Approach

The goal of this work is to provide FPGA users with a performance modelling framework for graph applications, to estimate performance and explore the design and optimization space (without going through the lengthy compilation process of the full hardware generation). The framework is built using information form three sources; the hardware intermediate compilation report, the application's kernel, and the graph input size. The intermediate compilation report can be generated in seconds; hence, we acquire the information needed without resorting to the lengthy compilation process of full hardware generation. Moreover, the model derives useful information about the graph input characteristics using information readily available upon reading the graph. Therefore, no time-intensive pre-processing of the graph input is required to extract the graph characteristics. Finally, additional information is extracted directly from the high-level representation of the application. Using such a collection of parameters, the framework was able to depict the execution profile of a specific graph input operated on by a specific graph application. The framework is also used to explore the optimization space and guides the user on how to optimize the hardware design for a specific graph input.

5.1.3 Contributions

The framework was tested on Intel's Devcloud platform and achieved speedup up to 3.4× by applying the recommended optimization strategy compared to the baseline implementation of Pannotia benchmark [8]. Furthermore, the framework recommended the best optimization strategy in 90% of the test cases without the need for time intensive synthesis of multiple optimization strategies to maximize performance.

The contributions of this chapter can be summarized as follows:

- A domain-specific performance model is devised to estimate the performance of graph applications,
- The performance and optimization space are explored based on graph input characteristics and hardware specifications, and
- Multiple optimization strategies are formulated and a framework is devised to guide FPGA users to choose the most appropriate optimization technique.

5.2 Background

This section introduces the applications and the data-set used to evaluate the performance model. Their characteristics are discussed to capture the scope of the domain-specific aspects of the performance model.

5.2.1 Graph Applications

In this work, three graph applications are used to evaluate the efficacy of the performance model; PageRank (PR), Single Source Shortest Path (SSSP), and Maximal Independent Set (MIS).

PageRank Algorithm

(PR) is an algorithm used by the Google search engine to rank web pages [54]. In this case, the web search space is represented as a graph. In each iteration, each node contributes a share of its PR value to its neighbors along its outgoing edges. Our implementation is based on the SPMV-based PageRank algorithm from [8]. The SPMV-based algorithm transforms the computation to an SPMV update operation. This transformation eliminates the need for frequent atomic addition operations, (which are required in the original algorithm). For instance, the web search space can be represented as a graph where nodes represent web pages. Initially, PR assigns an equal PR value of the reciprocal of the number of vertices to all the vertices of the graph. Afterward, a series of *superstep* iterations are executed that terminate either after a user-defined iteration count or an automated convergence check condition. In each iteration, each vertex donates a share of its PR value to its neighbors along its outgoing edges.

Single Source Shortest Path Algorithm

(SSSP) is the problem of finding the shortest path tree from one source node to the rest of the graph nodes [11]. Dijkstra (DJK) algorithm is used to solve this problem for a graph with non-negative edge path costs. This algorithm is used as a building block for many graph applications. An example is finding the shortest path between two intersections on a

road map, where the nodes correspond to intersections and the edges correspond to roads connecting them. Given a source node, the algorithm finds the path with the lowest cost between the source node and all the other nodes in the graph. A distance array is used to keep track of visited nodes as well as the path of nodes along that path. Visiting a new node triggers computation for its neighbors. If a new shorter path is discovered, the path is committed as the new shortest path. Otherwise, the old shortest path is kept, and the new node is marked as visited.

Maximal Independent Set Algorithm

(MIS) finds a maximal subset of nodes in a graph such that no nodes outside that set may join it (per the independent set theory) [1]. This is also another example of a popular building block for many graph applications. Initially, each node is labeled with a random integer value. The computation progresses iterating on each node of the graph, where the decision is made whether to include that node in the set. Nodes eligible for addition to the set are added to the array storing the up-to-date status of the independent set. The neighbors of each node added to the set are traversed and marked as inactive, to be removed from consideration. The algorithm will terminate when all nodes are visited and evaluated.

5.2.2 Graph Data-set & features

Graph processing applications follow a data-driven execution profile, which emphasizes its high sensitivity to the properties of the input graph. Hence, the network data repository [59] is used as the source of real-world graphs used in this work. To ensure the generality of different types of graphs, we use a set of graphs with a wide range of properties as shown in Table 5.1.

Graph	Vertices	Edges	Diameter	Degree
socfb-uci-uni	58,790,782	92,208,196	9	4
ca-hollywood-2009	1,069,126	56,306,654	4	105
road-road-usa	23,947,347	28,854,313	10	2
socfb-A-anon	3,097,165	23,667,395	6	15
cit-patent	3,774,768	16,518,948	9	9
ca-coauthors-dblp	540,486	15,245,730	7	56
road-asia-osm	11,950,757	12,711,604	1377	2
web-BerkStan	685,230	7,600,596	10	22
web-it-2004	509,338	7,178,414	8	28
web-wikipedia2009	1,864,433	4,507,316	9	5
rt-retweet-crawl	1,112,702	2,278,853	6	4
road-roadNet-CA	1,957,027	2,760,389	442	3
web-Stanford	281,903	2,312,498	10	16
socfb-Duke14	9,885	506,438	3	103
socfb-Georgetown15	9,414	425,639	3	90
email-enron-large	33,696	180,812	5	11
cit-DBLP	12,591	49,744	5	7
web-google	1,299	2,774	8	4
eco-foodweb-baydry	128	2,138	2	33
eco-everglades	69	911	6	26
eco-stmarks	54	353	3	13

Table 5.1: Data-set used for evaluating the performance model

5.2.3 Hardware Parallelism Extensions

Parallel extension optimizations in the scope of HLS can be categorized into three main groups; SIMD vectorization, Compute Unit replication (CU), and Loop Unrolling (LU). Firstly, SIMD optimization is not available in this graph applications domain due to the presence of unavoidable cross iteration dependency in its loops. HLS compilers automatically remove SIMD optimizations for these loops. However, we use multiple compute units to explore the data parallelism space. On the other hand, loop unrolling synthesizes multiple copies of the loop while decreasing the iteration count by the same factor. This deepens the pipeline, so we categorized it as a pipeline parallelism extension.

5.3 The Performance Model Framework

This section discusses the construction of the performance model starting by the introduction of some terminology that will be used, then elaborate on the details of the performance model.

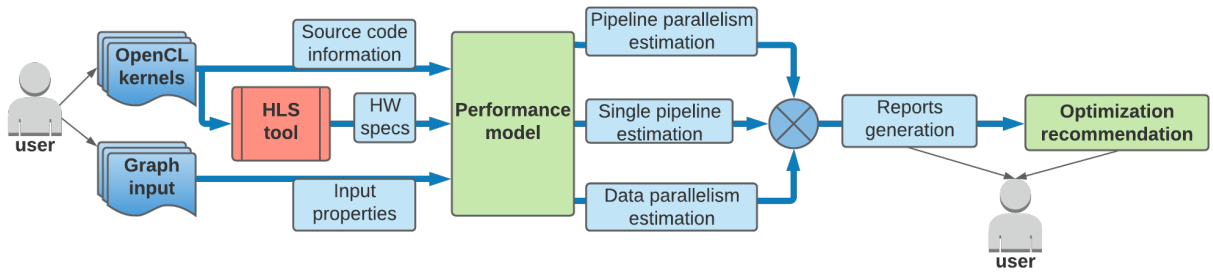


Figure 5.1: Overview of the performance model framework. Information flowing from the user, through the framework, then back to the user.

5.3.1 Information Collection & Terminology

The approach is simple and easy to use, with an easily obtainable set of information, one can predict the performance of a graph application. Not only that, but the optimization space is automatically explored without any additional information. As shown in figure 5.1, a user starts by generating the intermediate compilation report of the original application using the HLS tool, which takes seconds to compile. The second step is simply to collect a small amount of information from the source code and the data-set to be executed, as seen in Table 5.2. The final step would be running the analytical model to retrieve performance predictions and optimization recommendations. This is all done without the need for any long compilation process, or any dynamic profiling, or pre-processing of the graph data-set to extract information.

Parameter	Source	Description
N	Compiler	Number of pipeline stages
F_{pipe}	Compiler	Pipeline frequency
F_{mem}	Compiler	Memory frequency
II	Compiler	Initiation Interval: Number of stall cycles between iterations
Mem_cycles_{outer}	Compiler	Number of clock cycles to access the global memory in the outer loop
Mem_cycles_{inner}	Compiler	Number of clock cycles to access the global memory in the inner loop
$BURST\_SIZE$	Compiler	Number of maximum memory access transactions per cycle
CU	Compiler	Data parallelism degree: The number of Compute Units
LU	Compiler	Pipeline parallelism degree: Loop unrolling factor
$Mem\_accesses_{outer}$	Application	Number of global memory requests in the outer loop
$Mem\_accesses_{inner}$	Application	Number of global memory requests in the inner loop
$Nodes$	Input	Number of graph nodes
$Edges$	Input	Number of graph edges
$Chunk\_size$	Input	Ratio of edges to nodes. An indication of the average number of neighbours for each node

Table 5.2: Performance model parameters, source, and description

Performance Model Terminology

As shown in Table 5.2, the parameters are divided into three groups; hardware specifications, application characteristics, and input properties. Table 5.2 explains the meaning of each parameter and its source. These parameters form the basic set of parameters utilized; however, other parameters are derived from this basic set. These derived parameters are explained in detail in Subsection 5.3.2.

Model Parameters' Impact on Performance Estimation

Introduced here are the key parameters of the performance model and their impact on performance modelling and optimization techniques, starting by discussing the parameters obtained from the input graph. Pre-processing a graph input is time-intensive and defeats the purpose of this fast exploration of the optimization search space. Hence, the framework only depends on information readily available by reading the graph, the number of nodes ($Nodes$), and the number of edges ($Edges$). However, the ratio of edges to nodes as depicted in Equation 5.1, directly translates to the average degree of the graph, which is used to infer the average number of memory requests in the inner loop of the graph application.

$$Chunk\_size = Edges/Nodes \tag{5.1}$$

As for the application parameter, by studying the kernel structure of the graph application, it was observed that most graph applications have a double nested loop structure. In some instances, the outer loop fetches a graph node while the inner loop fetches its neighbors, such as BFS and SSSP. Other instances, the outer loop fetches pointers to the nodes in a graph level while the inner loop fetches the nodes value or state, such as PR. Hence, the performance model features $Mem\_accesses_{outer}$ & $Mem\_accesses_{inner}$. Finally, the compiler parameters are extracted from the intermediate compilation report, which compiles in seconds to minutes. This information is easily obtained without fully synthesizing the hardware design.

5.3.2 Performance Model Construction

To better understand the performance model, its structure is broken down to first introduce the single pipeline model, then we extend the approach for pipeline and data parallelism. Equation 5.2 shows the construction of the total estimated time.

$$est\_tot\_time = est\_pipe\_time + est\_mem\_time \tag{5.2}$$

Pipeline Latency

The performance model starts by quantifying an estimate of the pipeline's computational time est_pipe_time. As shown in Equation 5.3, the number of pipeline stages N is added to the pipeline throughput to quantify the number of computational clock cycles.

$$est\_pipe\_time = \frac{N + (Nodes \times (II + 1))}{F_{pipe}} \tag{5.3}$$

Memory Access Latency

First, the memory access model calculates the total number of memory requests that will be generated in the outer loop, as shown in Equation 5.4. It makes use of the $BURST\_SIZE$ parameter to accurately model its latency. Then the number of clock cycles required to fetch this amount of data from global memory is quantified in Equation 5.5.

$$Loop\_accesses_{outer} = \frac{Nodes \times Mem\_accesses_{outer}}{BURST\_SIZE} \tag{5.4}$$

$$Loop\_latency_{outer} = Loop\_accesses_{outer} \times Mem\_cycles_{outer} \tag{5.5}$$

On the other hand, inner loop modelling uses $Chunk\_size$ to quantify how it utilizes the burst coalescing feature of the global memory access as depicted in Equation 5.6. With the number of inner loop invocations $Iters_{inner} = Nodes$, the total memory access requests generated by the inner loop are estimated as shown in Equations 5.7, 5.8. The total estimated memory access latency is quantified as shown in Equation 5.9.

$$Loop\_reqs_{inner} = \lceil \frac{Chunck\_size}{BURST\_SIZE} \rceil \tag{5.6}$$

$$Loop\_accesses_{inner} = Loop\_reqs_{inner} \times \\ Iters_{inner} \times Mem\_accesses_{inner} \tag{5.7}$$

$$Loop\_latency_{inner} = Loop\_accesses_{inner} \times Mem\_cycles_{inner} \tag{5.8}$$

$$est\_mem\_time = \frac{Loop\_latency_{inner} + Loop\_accesses_{outer}}{F_{mem}} \tag{5.9}$$

5.3.3 Parallelism Extension

The benefits of parallelism extensions are described here as well as how the framework captures the pressure imposed on the global memory interface.

Loop Unrolling Extension - Pipeline Parallelism

The offline compiler synthesizes loop unrolling by coalescing the load operations so that the compute unit can maximize its memory bandwidth efficiency. Loop unrolling increases the number of parallel operations by deepening the pipeline of the loop; therefore, extending the pipeline parallelism. This is reflected in the reduced trip count of the loop as shown in Equation 5.10. Moreover, it increases the memory bandwidth utilization as shown in Equation 5.11.

$$Iters_{inner\_LU} = Nodes/LU \tag{5.10}$$

$$Loop\_reqs_{inner\_LU} = \lceil \frac{Chunck\_size \times LU}{BURST\_SIZE} \rceil \tag{5.11}$$

The model accounts for the extra memory pressure exerted on the global memory interface. Through studying the execution schedule, we observed that the memory access units are overloaded by a factor of $(LU - LU_{coef})$, where LU_{coef} was empirically evaluated to a value of 2. Hence, we calculate a metric to quantify the memory overhead pressure factor as shown in Equation 5.12. It can be observed that as long as $Loop\_reqs_{inner\_LU} > 1$, loop unrolling will show great performance gains. However, when $Loop\_reqs_{inner\_LU} < 1$ degradation in performance is observed, on the count of extra memory pressure without maximizing the memory bandwidth utilization to mitigate such memory pressure.

$$Mem\_ovh\_factor = \frac{LU - LU_{coef}}{Loop\_reqs_{inner\_LU}} \tag{5.12}$$

Equation 5.13 demonstrates how to calculate the total number of accesses estimated for the unrolled loop.

$$Loop\_accesses_{inner\_LU} = Iters_{inner\_LU}(Loop\_accesses_{inner} + Mem\_ovh\_factor) \tag{5.13}$$

Multiple Processing Entities (PEs) Extension - Data Parallelism

The major difference between loop unrolling and compute unit replications is in the synthesized load/store units. Unlike loop unrolling, each compute unit has its own memory access interface. While this diminishes the memory bandwidth available to each compute unit, graphs with low $Chunk\_size$ show significant performance gains compared to the Single Work Item (SWI) implementation. The memory overhead coefficient is now calculated as in Equation 5.15, where CU_{coef} was evaluated to $CU/2$.

$$Loop\_reqs_{inner\_CU} = \lceil \frac{Chunck\_size \times CU}{BURST\_SIZE} \rceil \tag{5.14}$$

$$Mem\_ovh\_factor = Loop\_reqs_{inner\_CU} + (CU \times CU_{coef}) \quad (5.15)$$

Hence, to calculate the total $Loop\_latency_{inner\_CU}$, we quantify the size of memory accesses as shown in Equation 5.16, where $Iters_{inner\_CU} = Nodes/CU$.

$$Loop\_accesses_{inner\_CU} = Iters_{inner\_CU}(Loop\_accesses_{inner} + Mem\_ovh\_factor) \quad (5.16)$$

5.4 Evaluation & Analysis of the Framework

This section explores how the performance model is capable of predicting the performance of irregular graph applications. It starts by discussing the hardware resources used in this work, then steps into the details of performance prediction, optimization recommendations, and a deep analysis of the framework's results.

5.4.1 Experimental Setup

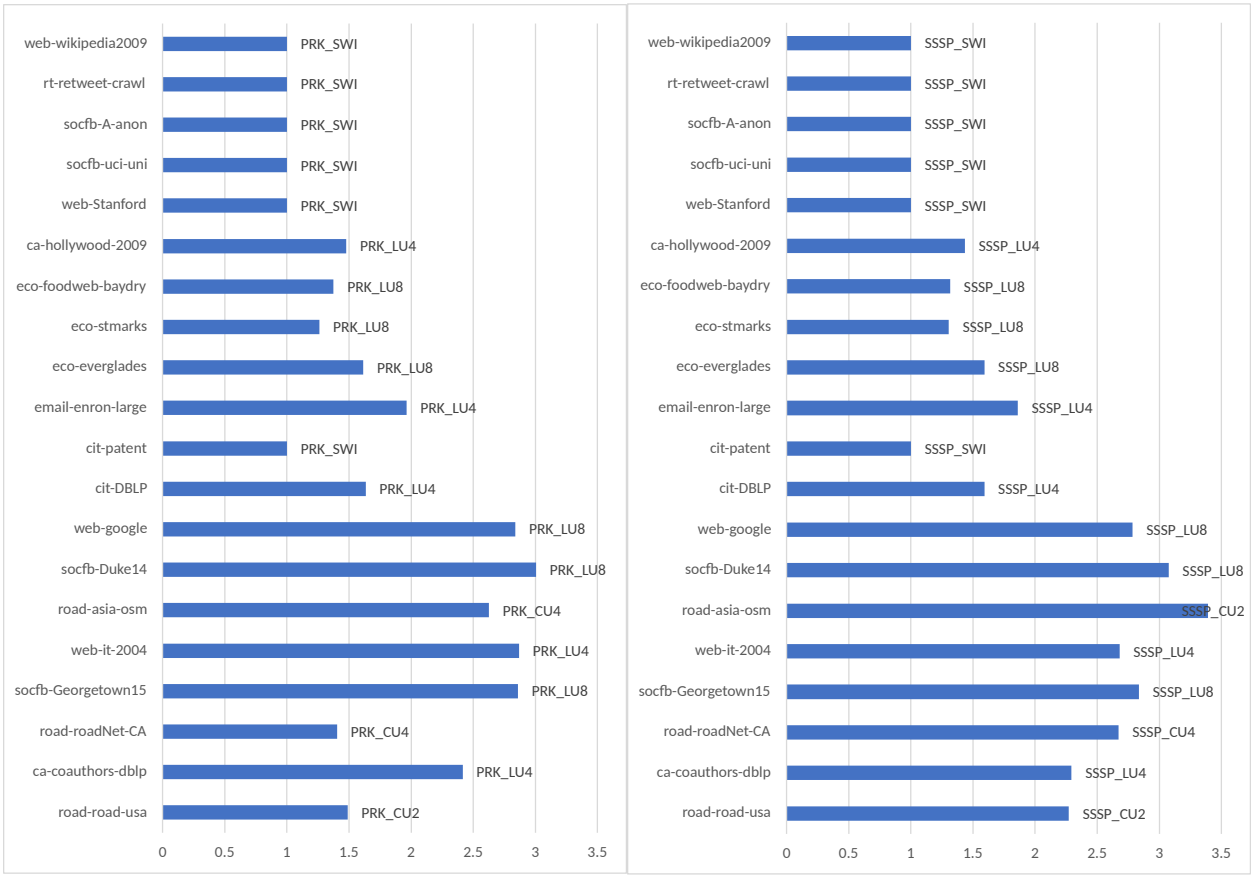
Computational Platform: The computational platform used is the new Intel Devcloud [36]. The computational node includes an Intel Arria 10 GX FPGA (Intel PAC Platform) and an Intel Xeon Gold 6128 CPU running at 3.40GHz. The FPGA board has two 4-GB DDR-4 memory banks, 1150-K logic elements, 54,260-Kb of M20K on-chip BRAM memory, and attaches to the host through a PCIe interface. The FPGA compiled designs run at an average frequency of 240 MHz. On the other hand, the CPU includes 24 cores, 32-KB L1 cache, and is running Ubuntu 18.04.

Hardware Designs: To test the framework and verify its correctness, six different designs were implemented for each of the three tested applications (PR, SSSP, and MIS). One design for the single pipeline (also known as Single Work Item "SWI"), three designs for pipeline parallelism (2, 4, and 8 loop unrolling factors), and two designs for data parallelism (2, and 4 CUs). All designs were compiled and executed across the input data-set to retrieve measured timing information for accuracy calculation purposes. Hence, there is a total of 18 designs executed over 20 graphs for a total of 360 instances used to test the performance model. As mentioned earlier a dataset with a wide spectrum of graph properties is used to ensure the generality of the performance model to any graph type.

5.4.2 Optimizations & Performance Analysis

To evaluate the performance gains of applying the recommended optimizations by our framework, all the design configurations were implemented for exhaustive testing. By retrieving the real measured time of each graph over all the design configurations for the three tested applications, we were able to identify whether our framework's recommendation did in fact achieve the best performance. Results show that **90%** of the test cases, the framework recommended the optimal choice. This section details the speedup achieved by using the recommended optimization strategy as well as elaborate on the wrong predictions.

As depicted in Figure 5.2, the framework achieves up to 3.4× speedup with an average speedup of 2.1×. It should be noted that a speedup of 1× is not a bad result, on the other hand, it is counted as an advantage of the framework. Speedup of 1× means that the framework recommended the SWI implementation, with no optimizations applied. After exhaustively measuring the real execution, it showed that expanding pipeline or data parallelism slows down the performance in these cases. Hence, the single pipeline SWI implementation turned out to be the best choice indeed.



(a) PR application.

(b) SSSP application.

Figure 5.2: Speedup of the recommended optimization strategy compared to the single pipeline implementation.

While SSSP application's recommendations were correct for all test cases, PR application had two errors in predicting the fastest optimization strategy, `road-road-usa` and `road-roadNet-CA`. This will be detailed later in Section 5.4.5. MIS application is not included in the figure because all speedups were 1×. The MIS application includes three memory accesses in the outer loop and four memory accesses in the inner loop, which is almost 3× the memory requirements for both PR and SSSP. Hence, the performance model always recommended the SWI implementation since parallelism extension only adds more pressure on the global memory interface.

5.4.3 Overall Performance Model Accuracy

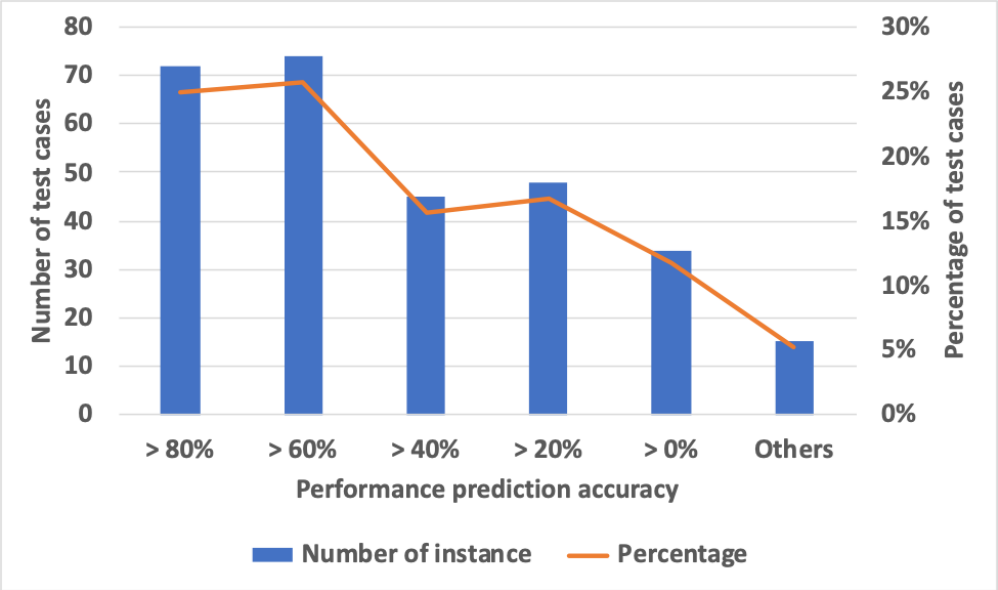


Figure 5.3: Overall prediction accuracy of the performance model, The figure is shown in percentiles

It should be noted that analytical performance models for FPGA designs, that use compile-time information only, rarely achieve accurate performance estimation for graph applications. Reproducing the performance model proposed in [88] showed up to two orders of magnitude deviation from actual performance when tested on graph applications. This is due to the irregular nature of these memory-bound graph applications which is not accurately captured in [88] as explained earlier in Chapter 2.

The domain-specific performance model presented in this chapter shows accuracy within 2× of the measured performance, which is considered acceptable considering the data-driven execution profile of graph applications. Figure 5.3 depicts the overall accuracy of the performance model broken down into percentiles. As shown in Figure 5.3, only 5% of the tested cases were out of bounds of the 2× performance prediction threshold. Moreover, 25% of the test cases achieved over 80% prediction accuracy.

5.4.4 Parallelism Extensions Evaluation

For the PR and SSSP applications, the performance model sustains good predictability for data parallelism extensions. However, as shown in Figure 5.4, pipeline parallelism prediction accuracy drops to 50%. The execution profile was analyzed to identify the root causes. The analysis showed that graphs with an average degree in the spectrum $4 < Chunk\_size < BURST\_SIZE$, show unpredictable memory access patterns that are not well captured by the model. This misprediction is compounded by the pipeline parallelism extension. However, graphs with an average degree outside of these bounds are well predicted achieving over 85% prediction accuracy.

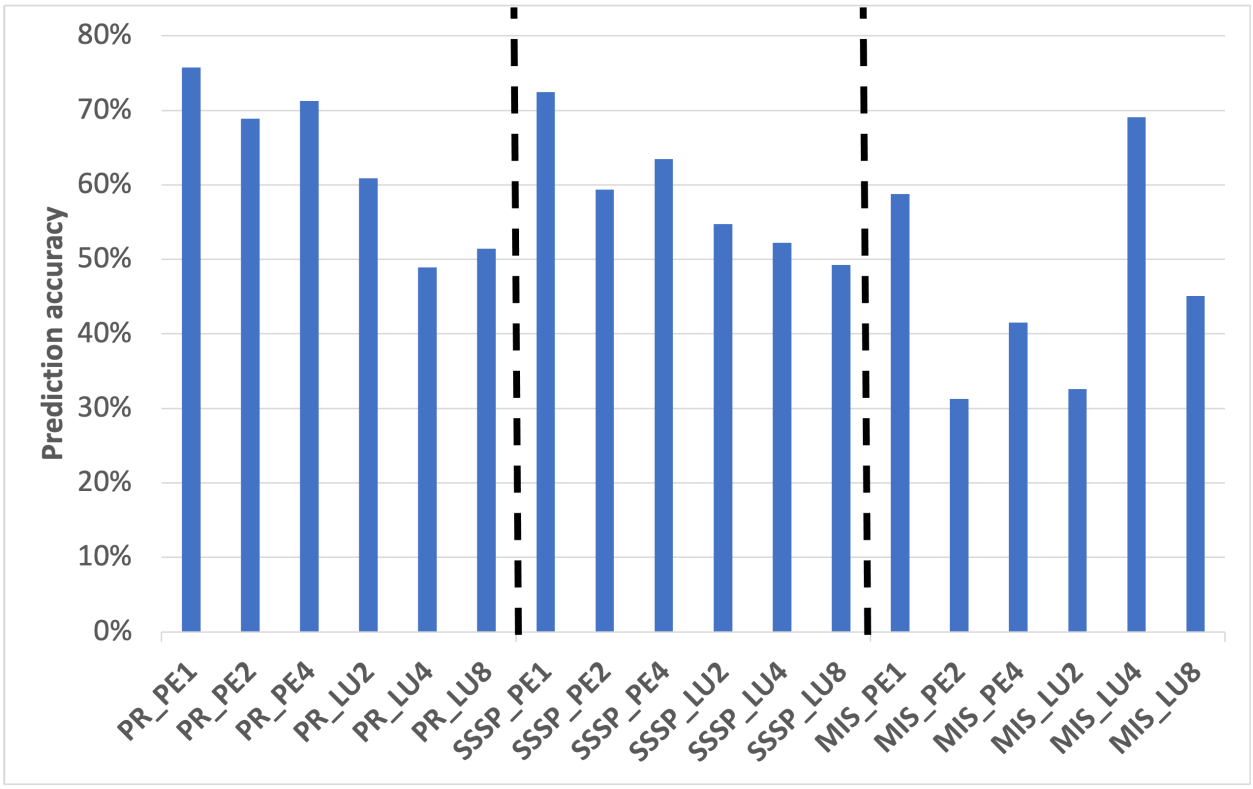


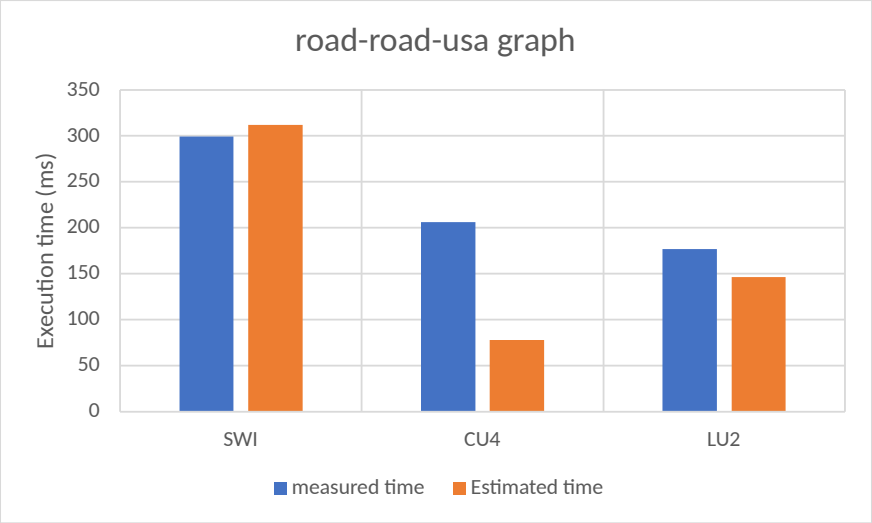
Figure 5.4: Average of prediction accuracy of each design across the whole data-set

Figure 5.4 also shows that the MIS application exhibits the lowest average prediction accuracy across the three applications. The MIS algorithm's behavior may choose to include a node in an independent set, marking its neighbors as ineligible to join that set. Then later in the execution cycle, that decision may be reversed if the reversal proved to maximize the independent set. Hence, the working set in each iteration may unexpectedly grow to cause a misprediction by the performance model. This is one type of profiling that can't be captured by an analytical model but would require dynamic profiling.

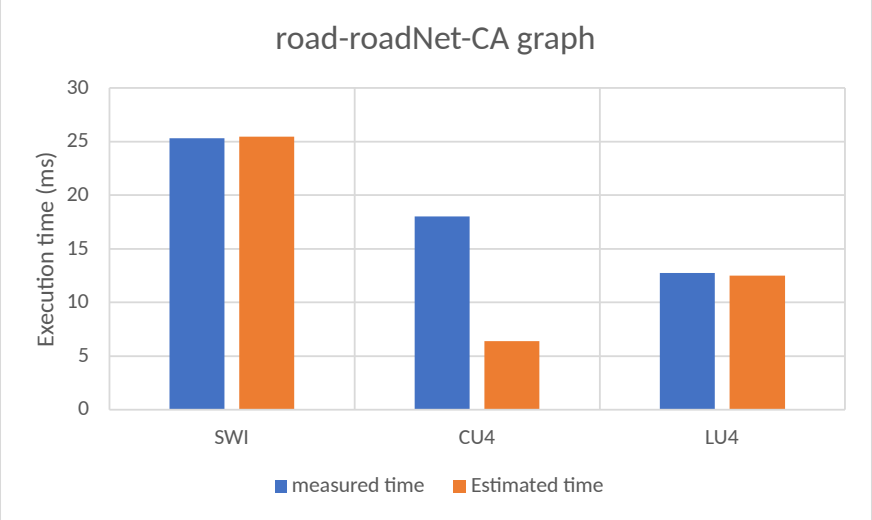
5.4.5 Discussion

Analyzing the PR application results showed that the framework diverges from optimal recommendation in two cases as shown in Figure 5.5. The performance model estimated that road network graphs showed the best performance by applying data parallelism extensions. This was true for all road network graphs for both PR and SSSP applications, except `road-road-usa` and `road-roadNet-CA` in the case of PR application. Figure 5.5 shows the measured execution time compared to the estimated execution time for three configurations, the baseline (SWI), the performance model recommended strategy (CU), and the optimal optimization strategy (LU). As depicted in Figures 5.5a and 5.5b, the performance model accurately estimated the execution time in both SWI and LU implementations. However, the performance model greatly underestimated the Mem_ovh_factor in the case of CU4 implementation.

On the other hand, the MIS application has as much as 3× the memory access requirements of both PR and SSSP. The performance model depicts pressure on the global memory interface and how parallel extensions may compound that problem. For that reason, the model seemed to always find the SWI implementation to be always the best choice. That was true for all



(a) `road-road-usa`



(b) `road-roadNet-CA`

Figure 5.5: Measured and estimated time for three PR configurations, the baseline (SWI), framework's recommendation (CU), optimal configuration (LU).

graphs except those exceptionally small graphs that make efficient use of on-chip memory, `eco-everglades`, `eco-stmarks`, and `eco-foodweb-baydry`. Using CU4 configuration for these three graphs shows an average of 1.4× speedup.

To sum up, it was observed that road network graphs that have low average degrees have a fairly predictable execution profile. The low number of memory accesses at the inner loop

will not overwhelm the external memory interface. The model also performs well with social network graphs with a very high average degree that directly translated to a bottle-necked inner loop, which is accounted for in the performance model. However, the gray area identified earlier where the average degree is in the spectrum $4 < Chunk\_size < BURST\_SIZE$ was shown to be prone to prediction estimation errors.

5.5 Summary

Graph applications have a challenging execution profile that is hard to optimize and harder to model. The data-driven execution profile makes it hard to capture performance estimations using compile time analytical models. However, in this work, a domain-specific performance modelling framework was presented. This work projects the performance of graph applications using information from intermediate compilation report of HLS tools, the high-level representation of the application, and the graph input properties. Furthermore, the framework explores the optimization space without the need for a lengthy hardware synthesis process for each optimization strategy to be tested. The framework shows up to 3.4× speedup with an average of 2.1× compared to the single pipeline implementation via using optimization strategies suggested by the framework. The optimization strategy recommended by the framework achieved the best performance in over 90% of the tested cases.

Chapter 6

Conclusion

With many real-world problems represented as graph applications, in addition to the exponentially growing graph sizes, there is an ever-growing interest in high performance graph solvers. Not only performance is important, scalability and power efficiency are also becoming increasingly important pillars in high performance computing. As a consequence, heterogeneous computing platforms that employ hardware accelerators are highly in demand. This dissertation targets the graph processing and heterogeneous computing communities. The research conducted here aims to enhance graph processing via exploiting heterogeneous computational power, more specifically CPU-FPGA hybrid systems. Moreover, innovative ideas are employed to address some of the inherent challenges of graph processing that hinder its optimization process, especially on FPGAs. Furthermore, FPGA design and optimization are explored to improve the efficiency of the synthesized hardware designs. Additionally, a domain-specific approach is employed to model the performance of graph applications and explore it optimization space.

This work presented automated frameworks to harness the computational power of heterogeneous systems, explore the optimization space of HLS-based FPGA designs, and model the performance of irregular applications on FPGAs. Extensive experimentation on representative datasets showed (1) the efficacy of the proposed frameworks to increase the efficiency of utilizing a heterogeneous system, (2) how this work facilitates integrating FPGAs into mainstream computing, (3) the performance and scalability improvement of graph processing.

6.1 Dissertation Summary

The collective outcome of this work makes this research well positioned to harness the computational power of CPU-FPGA hybrid systems, more specifically, when targeting the challenging problem of irregular graph applications. The scope of this work is focused on efficiently utilizing hardware accelerators, i.e. FPGA devices, in addition to cooperatively collaborating with general purpose CPUs. The architectural strengths of each hardware device in such a heterogeneous ecosystem are exploited to maximize performance of graph processing. The contributions of this work are explained below.

Workload distribution on CPU-FPGA hybrid systems: In many instances, the processing requirements of a given analytics computation are heavily data-dependent and rely heavily on the characteristics of the graph instance, which, if known could influence the computational solution. This research examined two hypotheses. First, a variety of properties of a set of graph instances are extracted and examined to determine if they can be used to predict the analytical execution profile. Second, this information is used to guide the workload partitioning and properly schedule it on a hybrid CPU-FPGA system. This work is intended to enhance the performance and scalability of graph processing. An automated framework (Graph Analytics on Hybrid Systems "GAHS") was presented for workload partitioning and scheduling of graph analytics on hybrid systems. Additionally, the decision-making process of the framework is configured to be guided by data input properties. The goal of this work is to expand the design space exploration to not only depend on the characteristics of the application and the available hardware resources but also include data input properties. Results show that up to 6.5× speedup can be attained over a CPU-only or FPGA-only implementations through proper partitioning. Moreover, an average of 18× speedup is achieved compared to state-of-the-art hybrid FPGA solvers.

HLS design space and optimization exploration: Unlike traditional hardware description languages (HDLs), OpenCL provides an abstract interface to facilitate high productivity, enabling end users to rapidly describe the required computations, including parallelism and data movement, to create custom hardware accelerators for their applications. This HLS approach made FPGAs accessible to the masses facilitating its use in a heterogeneous ecosystem. However, these OpenCL-realized accelerators are unlikely to make efficient use of the reconfigurable fabric without adopting FPGA-specific optimizations, particularly for irregular applications such as graph processing. Consequently, the FPGA-specific optimization space has been explored presenting insights on which optimization techniques improve application performance and resource utilization. Exploring this optimization space will enable end users to harness the computational potential of the FPGA. Experiments using representative kernels from the graph traversal, combinational logic, and sparse linear algebra applications show that FPGA-specific optimizations can improve the performance of irregular applications by up to 27-fold in comparison to the architecture-agnostic OpenCL code from the OpenDwarfs benchmark suite.

Domain-specific performance modelling for FPGA designs and automated optimization space exploration: Performance modelling research has been mainly focused on regular applications with uniform memory access patterns. These performance models fail to accurately capture the performance of graph applications with irregular memory access patterns. Hence, this work presents a domain-specific performance model targeting graph applications synthesized using HLS tools for FPGAs. The performance model utilizes information from three sources; the hardware intermediate compilation report, the application's kernel, and the graph input size. While the compilation process of HLS tools takes hours, the information required by the performance model can be extracted from the intermediate compilation report, which compiles in seconds. The goal of this work is to

provide FPGA users with a performance modelling framework for graph applications, to estimate performance and explore the design and optimization space (without going through the lengthy compilation process of the full hardware generation). The framework was tested on Intel's Devcloud platform and achieved speedup up to 3.4× by applying our framework's recommended optimization strategy compared to the single pipeline implementation. The framework recommended the best optimization strategy in 90% of the test cases.

6.2 Future Work

It can be considered that there are three main hardware devices involved in heterogeneous computing, CPUs, GPUs, and FPGAs. While this dissertation covers the scope of CPU-FPGA hybrid systems, augmenting GPU architectures is a strong candidate for future research. Each of the depicted hardware devices has its strengths and its weaknesses. Ultimately having a CPU-GPU-FPGA hybrid system, and being able to efficiently harness their collective computational power, has a great potential for revolutionizing the future of high performance computing, especially when targeting graph applications.

Hence, two research paths could be pursued following this dissertation. First, exploiting the strength of the massively parallel GPU architectures for graph processing while utilizing the innovative idea of using graph input properties to guide runtime decisions. Second, integrating GPU architectures in the automated framework for workload distribution and scheduling to enable exploiting the strengths of CPUs, GPUs, and FPGAs under the umbrella of the same heterogeneous ecosystem.

Bibliography

[1] Noga Alon, László Babai, and Alon Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of algorithms*, 7(4):567–583, 1986.

[2] Altera. Altera sdk for opencl: Best practices guide, 2015. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf.

[3] Altera. Altera sdk for opencl programming guide, 2015. URL https://www.altera.com/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[4] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on fpgas: Taxonomy, survey, challenges. *arXiv preprint arXiv:1903.06697*, 2019.

[5] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, pages 8–15, July 2012. doi: 10.1109/ASAP.2012.30.

[6] Aydin Buluç and John R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, 2011. ISSN 10943420. doi: 10.1177/1094342011403516.

[7] Shuai Che, Jeremy W. Sheaffer, Michael Boyer, Lukasz G. Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to

contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, IISWC '10, pages 1–11, 2010.

[8] Shuai Che, Bradford M Beckmann, Steven K Reinhardt, and Kevin Skadron. Pannotia: Understanding irregular gpgpu graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195. IEEE, 2013.

[9] Derek Chiou. Intel acquires altera: How will the world of fpgas be affected? In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, page 148, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338561. doi: 10.1145/2847263.2857658. URL https://doi.org/10.1145/2847263.2857658.

[10] Don Clark. Amd agrees to buy xilinx for $35 billion in stock, 2020. URL https://www.nytimes.com/2020/10/27/technology/amd-xilinx-35-billion-stock-deal.html.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Section 22.4: topological sort. *Introduction to Algorithms (2nd ed.), MIT Press and McGraw-Hill*, pages 549–552, 2001.

[12] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From opencl to high-performance hardware on fpgas. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, Aug 2012.

[13] Bruno Da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. Performance modeling for fpgas: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013, 2013.

[14] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. Fpgp: Graph processing

framework on fpga a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 105–110. ACM, 2016.

[15] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 217–226. ACM, 2017.

[16] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A high level API for fast development of high performance graph analytics on GPUs. *2nd International Workshop on Graph Data Management Experiences and Systems, GRADES 2014 - Co-located with SIGMOD/PODS 2014*, 2014. doi: 10.1145/2621934.2621936.

[17] Shanyuan Gao and Jeremy Chritz. Characterization of opencl on a scalable fpga architecture. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2014.

[18] Y. Gao and P. Zhang. A survey of homogeneous and heterogeneous system architectures in high performance computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 170–175, 2016. doi: 10.1109/SmartCloud.2016.36.

[19] Abdullah Gharaibeh, Lauro Beltrao Costa, Elizeu Santos-Neto, and Matei Ripeanu. On graphs, gpus, and blind dating: A workload to processor matchmaking quest. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 851–862. IEEE, 2013.

[20] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2:1–18, 2005.

[21] Robert J. Halstead, Jason Villarreal, and Walid Najjar. Exploring irregular memory accesses on fpgas. In *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '11, pages 31–34, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1121-2. doi: 10.1145/2089142.2089151. URL http://doi.acm.org/10.1145/2089142.2089151.

[22] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2016-Decem, 2016. ISSN 10724451. doi: 10.1109/MICRO.2016.7783759.

[23] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. *ACM SIGARCH Computer Architecture News*, 38(3):37–47, 2010.

[24] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[25] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013.

[26] Mohamed W Hassan, Ahmed E Helal, Peter M Athanas, Wu-Chun Feng, and Yasser Y Hanafy. Exploring FPGA-specific optimizations for irregular OpenCL applications. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2018.

[27] Mohamed W. Hassan, Peter M. Athanas, and Yasser Y. Hanafy. Domain-specific modelling and optimization for graph processing on FPGAS. In *International Symposium on Applied Reconfigurable Computing*. Springer, 2021.

[28] Mohamed W. Hasssan and Peter M. Athanas. Graph analytics on hybrid system (GAHS) case study: Pagerank. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021.

[29] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. Scalable Graph Processing Frameworks. *ACM Computing Surveys*, 51(3):1–53, 2018. ISSN 03600300. doi: 10.1145/3199523. URL http://dl.acm.org/citation.cfm?doid=3212709.3199523.

[30] Ahmed E Helal, Amr M Bayoumi, and Yasser Y Hanafy. Parallel circuit simulation using the direct method on a heterogeneous cloud. In *Proceedings of the 52nd Annual Design Automation Conference*, page 186. ACM, 2015.

[31] Ahmed E Helal, Wu-chun Feng, Changhee Jung, and Yasser Y Hanafy. Automatch: An automated framework for relative performance estimation and workload distribution on heterogeneous hpc systems. In *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, pages 32–42. IEEE, 2017.

[32] Ahmed E Helal, Changhee Jung, Wu-chun Feng, and Yasser Y Hanafy. Commanalyzer: automated estimation of communication cost and scalability on hpc clusters from sequential code. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 80–91. ACM, 2018.

[33] K. Hou, H. Wang, W. Feng, J. S. Vetter, and S. Lee. Highly efficient compensation-based parallelism for wavefront loops on gpus. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 276–285, May 2018.

[34] Kaixi Hou, Weifeng Liu, Hao Wang, and Wu-chun Feng. Fast segmented sort on gpus. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 12:1–12:10, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5020-4.

[35] Intel. Slicing-by-8, crc, 2014. URL https://sourceforge.net/projects/slicing-by-8/.

[36] Intel. Intel devcloud for oneapi projects, 2019. URL https://software.intel.com/en-us/devcloud/oneapi.

[37] Edward Kao, Vijay Gadepally, Michael Hurley, Michael Jones, Jeremy Kepner, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Siddharth Samsi, William Song, et al. Streaming graph challenge: Stochastic block partition. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–12. IEEE, 2017.

[38] M. E. Kounavis and F. L. Berry. A systematic approach to building high performance software-based crc generators. In *IEEE Symposium on Computers and Communications (ISCC'05)*, pages 855–862, June 2005.

[39] K. Krommydas, W. Feng, M. Owaida, C. D. Antonopoulos, and N. Bellas. On the characterization of opencl dwarfs on fixed and reconfigurable platforms. In *IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 153–160, June 2014.

[40] K. Krommydas, A. E. Helal, A. Verma, and W. Feng. Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 198–198, May 2016.

[41] Konstantinos Krommydas, Wu Feng, Christos D. Antonopoulos, and Nikolaos Bellas.

Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *J. Signal Process. Syst.*, 85(3):373–392, December 2016. ISSN 1939-8018.

[42] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a {PC}. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 31–46, 2012.

[43] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1): 1, 2016.

[44] B Lorica. Single server systems can tackle big data. *Retrieved July*, 25:2015, 2013.

[45] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on uncertainty in artificial intelligence (UAI)*, pages 340–349, 2010.

[46] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[47] Y. Luo, X. Wen, K. Yoshii, S. Ogrenci-Memik, G. Memik, H. Finkel, and F. Cappello. Evaluating irregular memory access on opencl fpga platforms: A case study with xsbench. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 135–145, 2010. ISSN 07308078. doi: 10.1145/1807167.1807184.

[49] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. A pattern based algorithmic autotuner for graph processing on gpus. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 201–213. ACM, 2019.

[50] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145832. URL http://doi.acm.org/10.1145/2145816.2145832.

[51] Geoffrey Ndu, Javier Navaridas, and Mikel Luján. Cho: towards a benchmark suite for opencl fpga accelerators. In *3rd International Workshop on OpenCL*, page 10. ACM, 2015.

[52] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin. Graphgen: An fpga framework for vertex-centric graph computation. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28, May 2014. doi: 10.1109/FCCM.2014.15.

[53] Tayo Oguntebi and Kunle Olukotun. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM, 2016.

[54] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[55] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka. Evaluating high-level design strategies on fpgas for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017.

[56] A. Prakash, C. T. Clarke, S. Lam, and T. Srikanthan. Rapid memory-aware selection of hardware accelerators in programmable soc design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(3):445–456, March 2018. doi: 10.1109/TVLSI.2017.2769125.

[57] Luis Carlos Maria Remis. *Breadth-first search for social network graphs on heterogenous platforms*. PhD thesis, 2016.

[58] Ryan A. Rossi and Nesreen K. Ahmed. Graph repository, 2013. URL http://www.graphrepository.com.

[59] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. URL http://networkrepository.com.

[60] Antony Rowstron, Dushyanth Narayanan, Austin Donnelly, Greg O'Shea, and Andrew Douglas. Nobody ever got fired for using hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, page 2. ACM, 2012.

[61] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[62] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 410–424. ACM, 2015.

[63] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey.

Navigating the maze of graph analytics frameworks using massive graph datasets. 2014. doi: 10.1145/2588555.2610518.

[64] Sean O Settle. High-performance dynamic programming on fpgas with opencl. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sep 2013.

[65] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[66] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[67] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. GoFFish: A sub-graph centric framework for large-scale graph analytics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8632 LNCS(1):451–462, 2014. ISSN 16113349. doi: 10.1007/978-3-319-09873-9_38.

[68] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. High-Performance Graph Analytics on Manycore Processors. *2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS 2015*, pages 17–27, 2015. doi: 10.1109/IPDPS.2015.54.

[69] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. GraphMat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015. ISSN 21508097. doi: 10.14778/2809974.2809983.

[70] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John

McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[71] Y. Umuroglu, D. Morrison, and M. Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sep. 2015. doi: 10.1109/FPL.2015.7293939.

[72] Anshuman Verma, Ahmed E Helal, Konstantinos Krommydas, and Wu-Chun Feng. Accelerating workloads on fpgas via opencl: A case study with opendwarfs. Technical report, Virginia Polytechnic Institute and State University, 2016.

[73] Guozhang Wang, Wenlei Xie, Alan J Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13, pages 3–6, 2013.

[74] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 11:1–11:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4092-2. doi: 10.1145/2851141.2851145. URL http://doi.acm.org/10.1145/2851141.2851145.

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

[76] Yu Wang, James C. Hoe, and Eriko Nurvitadhi. Processor assisted worklist scheduling for fpga accelerated graph processing on a shared-memory platform. *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 136–144, 2019.

[77] Z. Wang, B. He, W. Zhang, and S. Jiang. A performance analysis framework for optimizing opencl applications on fpgas. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 114–125, March 2016.

[78] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52 (4):65–76, 2009.

[79] Zhen Xie, Guangming Tan, Weifeng Liu, and Ninghui Sun. Ia-spgemm: an input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the ACM International Conference on Supercomputing*, pages 94–105. ACM, 2019.

[80] Xilinx. The xilinx sdaccel development environment, 2015. URL https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf.

[81] Jialiang Zhang, Soroosh Khoram, and Jing Li. Boosting the performance of fpga-based graph processor using hybrid memory cube: A case for breadth first search. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 207–216, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4354-1. doi: 10.1145/3020078.3021737. URL http://doi.acm.org/10.1145/3020078.3021737.

[82] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. Performance modeling and directives optimization for high-level synthesis on fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(7):1428–1441, 2019.

[83] Jianlong Zhong and Bingsheng He. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2013.

[84] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. Optimizing memory performance for FPGA implementation of pagerank. *2015 International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015*, 2016. doi: 10.1109/ReConFig.2015.7393332.

[85] Shijie Zhou, Kartik Lakhotia, Shreyas G. Singapura, Hanqing Zeng, Rajgopal Kannan, Viktor K. Prasanna, James Fox, Euna Kim, Oded Green, and David A. Bader. Design and implementation of parallel PageRank on multicore platforms. *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, pages 0–5, 2017. doi: 10.1109/HPEC.2017.8091048.

[86] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.

[87] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 Annual Technical Conference (ATC)*, pages 375–386, 2015.

[88] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing opencl kernels for high performance computing with fpgas. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 35:1–35:12, 2016.