Characterization of Sparsity-aware Optimization Paths for Graph Traversal on FPGA

Atharva M. Gondhalekar

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Wu-chun Feng, Chair
Cameron Patterson
Binoy Ravindran

August 12, 2022
Blacksburg, Virginia

Keywords: FPGA, Graph Traversal, BFS, Sparse graphs.

Characterization of Sparsity-aware Optimization Paths for Graph Traversal on FPGA

Atharva M. Gondhalekar

(ABSTRACT)

Breath-first search (BFS) is a fundamental building block in many graph-based applications, but it is difficult to optimize for a field-programmable gate array (FPGA) due to its irregular memory-access patterns. Prior work, based on hardware description languages (HDLs) and high-level synthesis (HLS), address the memory-access bottleneck of BFS by using techniques such as data alignment and compute-unit replication on FPGAs. The efficacy of such optimizations depends on factors such as the sparsity of target graph datasets. Optimizations intended for sparse graphs may not work as effectively for dense graphs on an FPGA and vice versa. This thesis presents two sets of FPGA optimization strategies for BFS, one for near-hypersparse graphs and the other designed for sparse to moderately dense graphs. For near-hypersparse graphs, a queue-based kernel with maximal use of local memory on FPGA is implemented. For denser graphs, an array-based kernel with compute-unit replication is implemented. Across a diverse collection of graphs, our OpenCL optimization strategies for near-hypersparse graphs delivers a 5.7× to 22.3× speedup over a state-of-the-art OpenCL implementation, when evaluated on an Intel Stratix 10 FPGA. The optimization strategies for sparse to moderately dense graphs deliver 1.1× to 2.3× speedup over a state-of-the-art OpenCL implementation on the same FPGA. Finally, this work uses graph metrics such as average degree and Gini coefficient to observe the impact of graph properties on the performance of the proposed optimization strategies.

Characterization of Sparsity-aware Optimization Paths for Graph Traversal on FPGA

Atharva M. Gondhalekar

(GENERAL AUDIENCE ABSTRACT)

A graph is a data structure that typically consists of two sets – a set of vertices and a set of edges representing connections between the vertices. Graphs are used in a broad set of application domains such as the testing and verification of digital circuits, data mining of social networks, and analysis of road networks. In such application areas, breadth-first search (BFS) is a fundamental building block. BFS is used to identify the minimum number of edges needed to be traversed from a source vertex to one or many destination vertices. In recent years, several attempts have been made to optimize the performance of BFS on reconfigurable architectures such as field-programmable gate arrays (FPGAs). However, the optimization strategies for BFS are not necessarily applicable to all types of graphs. Moreover, the efficacy of such optimizations oftentimes depends on the sparsity of input graphs. To that end, this work presents optimization strategies for graphs with varying levels of sparsity. Furthermore, this work shows that by tailoring the BFS design based on the sparsity of the input graph, significant performance improvements are obtained over the state-of-the-art BFS implementations on an FPGA.

Dedication

To my parents and grandparents

Acknowledgments

I would like to express my gratitude to my parents and grandparents. I am grateful to my advisor Dr. Wu-chun Feng. I have learned many valuable lessons because of his feedback on my work.

I want to thank my labmates for providing constructive feedback on my work and creating a friendly and inspiring atmosphere in the lab.

I want to take this opportunity to acknowledge the sponsors. This work was supported in part by NSF I/UCRC CNS-1822080 via the NSF Center for Space, High-performance, and Resilient Computing (SHREC). Lastly, I would like to thank Intel for access to the Intel Devcloud for providing access to the hardware resources and software tools.

This thesis is adapted from and expands upon the text originally published in:

Atharva Gondhalekar and Wu-chun Feng, "Exploring FPGA Optimizations in OpenCL for Breadth-First Search on Sparse Graph Datasets," 2020, 30th International Conference on Field-Programmable Logic and Applications (FPL), 2020, pp. 133-137.

Contents

List of Figures

List of Tables

List of Abbreviations

BFS Breadth-First Search

FPGA Field Programmable Gate Array

HDL Hardware Description Language

HLS High-level Synthesis

OpenCL Open Compute Language

PE Processing Element

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) are gaining traction in the high-performance computing (HPC) community to accelerate a wide range of applications due to their reconfigurable computing capability. FPGA-based solutions can deliver comparable performance to the graphics processing units (GPUs) while achieving much better performance per watt [3, 31]. For example, Hitgraph [31] is an FPGA-based graph processing framework. On a Xilinx UltraScale+ FPGA, Hitgraph delivers comparable performance to the NVIDIA K40 GPU while consuming 20× less power. Traditionally, FPGA programming has been done via a hardware description language (HDL). An HDL provides fine-grained control over reconfigurable hardware but at the cost of a steep learning curve and tedious design cycle, resulting in *low productivity*.

In contrast, high-level synthesis (HLS) offers complementary trade-offs to HDL. That is, HLS frameworks offer *higher productivity* by presenting a significantly higher level of programming abstraction, but this comes at the expense of fine-grained control over the hardware and, in turn, *lower performance*. High-level synthesis (HLS) has been a significant boon to productivity as it enables design space explorations while hiding the complexities of the HDL.

In recent years, the Open Computing Language (OpenCL) [23] has emerged as a vendor-agnostic HLS language that offers the added benefit of interoperation with other OpenCL platforms (e.g., CPU, GPU, DSP, FPGA) and existing OpenCL software. This thesis uses

OpenCL as an HLS framework for exploring optimizations for breadth-first search.

1.1 Breadth-First Search on FPGAs

Breadth-first search (BFS) traversal is a fundamental algorithm across a broad set of application domains such as the testing and verification of digital circuits, data mining of social networks, and analysis of road networks. In recent years, many efforts have been made to optimize and accelerate graph traversal across a diverse set of fixed [4, 9, 20, 22, 29] and reconfigurable architectures [6, 8, 24, 31, 32] at different levels of programming abstraction, e.g., OpenCL [4, 6, 10, 11, 17] and HDL [31, 32].

While some OpenCL implementations for FPGA achieve performance that is comparable to HDL-based implementations, e.g., [19], most OpenCL implementations deliver substantially poorer performance due to the irregular memory-access pattern of graph traversal [6, 11]. Chen et al. [6] use an edge-centric approach along with data shuffling to address the workload imbalance, while Liu et al. [19] use data alignment and graph reordering to mitigate the irregular memory-access bottleneck.

Recent work also makes use of compute-unit (CU) replication to create multiple processing elements to improve performance [6, 10, 19].

A level-synchronous BFS can be described using a filter-apply-expand abstraction, as implemented in the OpenDwarfs benchmark suite [17] and Gswitch system [22]. The filter stage identifies the "vertices in a given level" to process. This set of vertices is known as the active set, and the vertices in the active set are known as active vertices. The apply stage updates the cost of the vertices in the active set. In the expand stage, unvisited neighbors of the active vertices are marked to be used in the active set of the next iteration. The process

continues until the size of the active set reaches zero, as shown in Fig. 1.1.

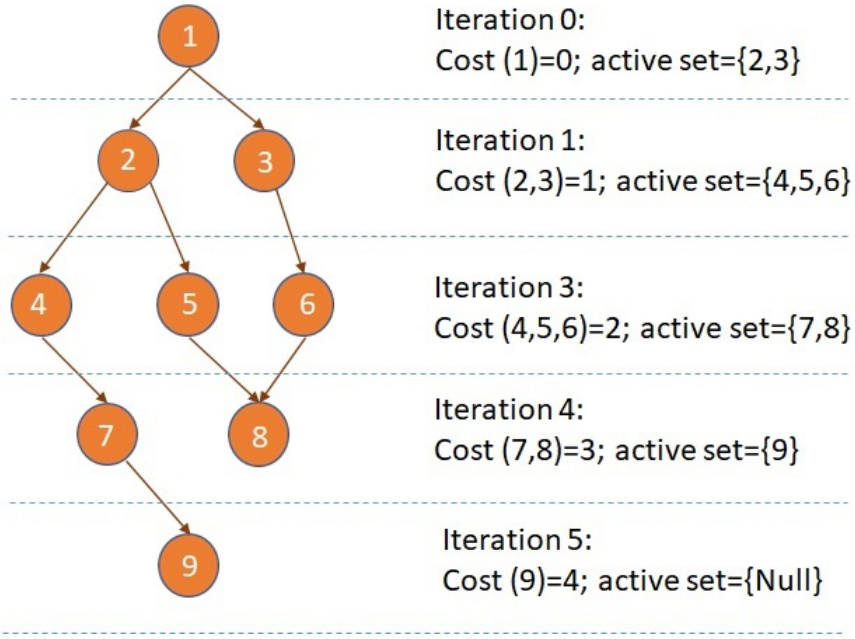

Figure 1.1: Level-synchronous BFS

Optimizing BFS for sparser graph inputs is important in many practical applications. For example, an essential component in the applications such as vehicle navigation [26] is performing BFS on sparser graphs such as road networks. The proposed optimizations in [6, 10, 19] perform well for sparse to moderately dense graphs (hereinafter referred to as denser graphs for brevity) but *not* for near-hypersparse graphs with low average degree, i.e., average degree < 3 (hereinafter referred to as sparser graphs for brevity). Optimizing BFS for sparser datasets remains a challenging task due to factors such as kernel launch overhead and poor workload distribution among processing elements (PE). Performance is impacted further when the bandwidth for certain global memory accesses is only a fraction of the peak bandwidth. To illustrate the problems associated with optimizing BFS for sparser graphs, an existing implementation of BFS from Spector [10] is used, and its performance is measured for a denser graph and a sparser graph. While both graphs have approximately the same

number of vertices, there is a large difference in the number of edges. The denser graph, Hollywood-2009, is a collaboration network that requires only 11 iterations of filter-apply-expand stages while the sparser graph, a road network denoted as inf-road-PA, needs 542 iterations, as shown in Fig. 1.2. The portion of the OpenCL code that runs on an FPGA is known as the OpenCL kernel. A kernel is launched for each iteration of BFS in Spector [10]. There is a non-zero overhead associated with kernel launch as there can be delays between when the kernel launch was issued on the host CPU and when the kernel starts running on an FPGA. For the sparser graph in Table 1.1 which requires 542 iterations, the kernel launch overhead is approximately 11× more than it is for the dense graphs.

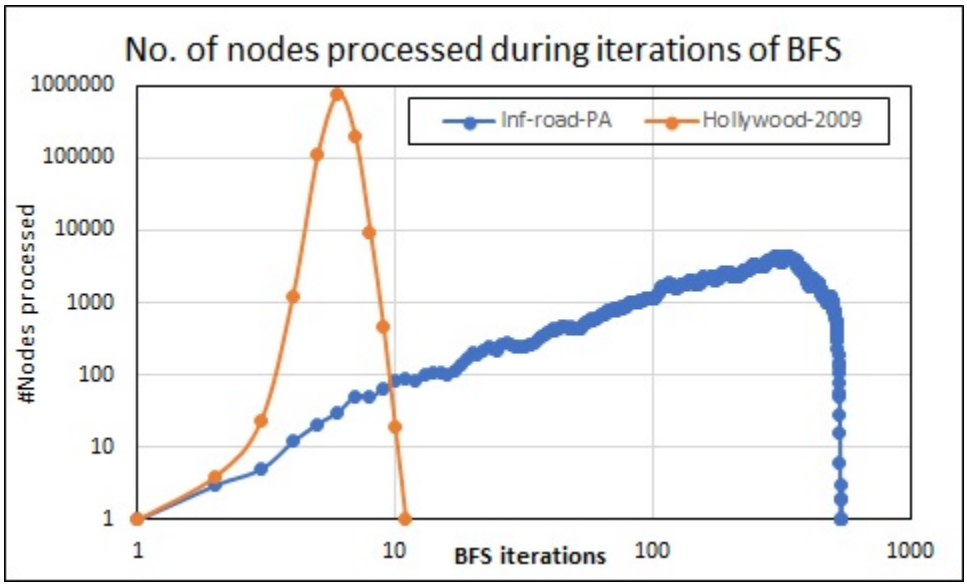

Figure 1.2: Progression of BFS on sparser and denser graphs

For the denser Hollywood-2009 graph, the majority of the work is done in iterations 5-7, where over 100,000 vertices are processed in each iteration. In this case, a design with multiple PEs [6, 19] or with SIMD work-item execution [10] can be used effectively. In contrast, only a small percentage of the total number of vertices are processed in each iteration for the sparser inf-road-PA graph, meaning that the work available for multiple PEs is limited. As a

result, Spector takes 5.2× *longer* to process the sparser graph than the denser Hollywood-2009 graph, as shown in Table 1.1.

This performance gap is further investigated using the Intel FPGA dynamic profiler for OpenCL [13]. For every global-memory operation, the profiler reports efficiency and average bandwidth. Efficiency is the percentage of total bytes acquired from the global memory that is used by the kernel.

Table 1.2 shows the bandwidth reported by the dynamic profiler when running Spector BFS. While many global memory transactions occur in all three stages (i.e., filter-apply-expand), the bandwidth and associated information for the operations that have the highest stall percentage from every stage is reported in Table 1.2. In addition to the workload distribution issue, running BFS on sparser graphs also has considerably low bandwidth efficiency for critical operations, as shown in Table 1.2.

Table 1.1: Performance evaluation of Spector [10] BFS for denser and sparser graphs

Graph	Description	Vertices	Bidirectional edges	Runtime of Spector [10] on Intel Stratix 10 (Seconds)	OpenCL kernel launch overhead Seconds)
Hollywood-2009	Collaboration network	1.1M	112.6M	1.201	0.061
Inf-road-PA	Road network	1.1M	3M	6.249	0.691

Table 1.2: Bandwidth utilization during the BFS execution stages

Graph	Filter		Apply	Expand	
	Bandwidth (MB/s)	Efficiency	Bandwidth (MB/s)	Bandwidth (MB/s)	Efficiency
Hollywood-2009	41.7	64.2	34.5	125.1	54.0
Inf-road-PA	16.0	18.1	22.2	13.0	11.4

To address the aforementioned limitations because of the kernel launch overhead and low bandwidth utilization for sparser graphs, this thesis presents a set of architecture-aware, framework-specific, and application-specific optimizations for sparser graphs, as summarized

Table 1.3: Summary of optimization strategies for sparser and denser graphs

	Strategy for sparser graphs	Strategy for denser graphs
Data structure	Queue and bitmaps	Arrays
OpenCL configuration	Single-task (ST) - §3.1.2	NDRange - §3.1.2
OpenCL framework-specific optimizations	Kernel fusion - §3.3.1.1 Elimination of host-based synchronization - §3.3.1.2	-
Architecture-aware optimizations	Maximal use of local memory - §3.3.2.1 Memory banking - §3.3.2.2	Compute-unit replication - §3.3.2.5 Constant work-group sizes - §3.3.2.4
Application-specific optimization	Merged apply and expand - §3.3.3.1 Elimination of duplicate entries - §3.3.3.2	Merged apply and expand - §3.3.3.1

in Table 1.3. The optimization strategy for sparser graphs includes a set of optimizations for a queue-based implementation of BFS in OpenCL's single-task (ST) configuration, which is effectively a sequential execution with pipelining. For sparser graphs, a queue-based implementation is presented, where the queue resides entirely in the local memory of FPGA. Kernel launch overhead and CPU-based synchronization during the BFS stages of filter-apply-expand is avoided by merging the three BFS stages into a single kernel, as shown in Table 1.3. The experiments were performed on an Intel Stratix 10 2800 SX FPGA, where the presence of a large number of BRAMs allowed implementation of local-memory optimizations on graphs with up to 2^{21} vertices (or 2,097,152 vertices). In addition to the optimizations for sparser graphs, this work evaluates the performance of an optimization strategy for denser graphs. The approach for the denser graph is based on an array-based BFS kernel that uses compute-unit replication in OpenCL's NDRange configuration, as summarized in Table 1.3. Finally, this work classifies the graph datasets based on the performance of BFS for the two different optimization strategies. Metrics such as average degree and Gini coefficient [18] are used to classify the graphs.

1.2 Contributions

The primary contributions of this thesis are:

- Two sets of optimization strategies — one targeted at sparser graphs, while the other for denser graphs.
    1. A queue-based implementation of BFS in OpenCL that incorporates local memory optimizations and delivers a speedup of 5.7× to 22.3× over OpenCL-based state-of-art implementations for sparser graphs.
    2. An implementation of BFS with compute-unit replication that delivers a speedup of 2.0× to 4.4× over OpenCL-based state-of-art implementations for denser graphs.
- Categorization of graph datasets using graph metrics such as average degree and Gini index [18] along with the performance of BFS for the aforementioned contrasting optimization strategies.

1.3 Attribution

This thesis expands upon the work published in the 30th International Conference on Field-Programmable Logic and Applications (FPL):

Atharva Gondhalekar and Wu-chun Feng, "Exploring FPGA Optimizations in OpenCL for Breadth-First Search on Sparse Graph Datasets," *30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 133-137, August-September 2020.

1.4 Thesis Organization

The rest of this document is organized as follows. §2 presents a review of the related work and a comparison of this work with prior OpenCL-based work on optimizing BFS. In §3, a description of the implementation of the queue and array-based BFS, along with a detailed description of two contrasting optimization approaches to BFS, is provided. §4 presents the performance evaluation and classification of graph datasets based on the performance of BFS. Finally, this thesis finishes with §5, which provides the conclusion and future directions for this work.

Chapter 2

Review of Literature

Breadth-first search is a widely studied graph algorithm that has been implemented across a diverse set of architectures, including GPUs and FPGAs. GPU-based implementations use SIMD parallelism to achieve high performance, whereas FPGA-based solutions improve performance by replicating a fine-grained custom pipeline on multiple processing elements (PEs). In this chapter, prior efforts to optimize BFS on GPUs as well as FPGAs are described and compared to the work presented in this thesis.

2.1 Graph Processing on GPUs

With the emergence of general-purpose computing on GPUs, many BFS implementations have been proposed. Gunrock [29] and Enterprise [20] are high-performance graph-processing frameworks on NVIDIA GPUs. Rodinia [4] and Pannotia [5] are OpenCL-based benchmark suites with BFS and other graph workloads. More recent advancements include Gswitch [22] and Sep-graph [28]. Gswitch is a pattern-based auto-tuner that attempts to dynamically select the best optimization strategy for a BFS iteration from a set of parallel implementations at runtime. Sep-graph also proposes runtime adaptation between multiple strategies to optimize BFS. While GPUs provide significant performance gains, many FPGA-based solutions for graph workloads and other applications have been presented that achieve comparable performance and a significantly higher performance per watt [1, 31].

2.2 Graph Processing on FPGAs

The research in graph processing on FPGAs is extensive. Hitgraph is an HDL-based edge-centric graph-processing framework [31]. Hitgraph consumes 20× less power than Gunrock on an NVIDIA K40 GPU; it also performs as well (or better than) the GPU-based Gunrock even though the GPU has 4× higher bandwidth than the target FPGA used in Hitgraph. Zhou et al. also present an edge-centric HDL-based BFS [32]. CyGraph is a reconfigurable architecture for parallel graph processing using queues [2]. HyVE is a hybrid vertex- and edge-centric implementation of BFS aimed at energy-efficient graph processing [12]. GraphOps presents a modular hardware library for energy-efficient processing of graph analytics algorithm [24]. Luo et al. evaluate the irregular memory accesses in a Monte Carlo simulation application and explore techniques to hide memory latency [21]. ForeGraph is a graph-processing framework for multi-FPGA systems [8]. A hybrid CPU-FPGA implementation is presented in [27]. None of the above, however, specifically addresses the run-time issues that arise when processing sparser graphs with BFS, namely the kernel launch overhead and poor workload distribution across processing elements.

2.3 OpenCL-based BFS on FPGAs

Czajkowski et al. present a compilation framework for OpenCL to generate high-performance hardware and show the efficacy of the OpenCL computing paradigm on FPGA [7]. Several implementations of BFS have been proposed in recent years. OpenDwarfs is one of the first OpenCL-based implementations for FPGA [17]. It is a vertex-centric and architecture-agnostic BFS kernel that is functional across CPU, GPU, APU, and FPGA.

In the BFS implementation from Spector [10], architecture-aware optimizations for FPGA

are applied to the OpenDwarfs kernel. These optimizations include compute-unit (CU) replication, loop unrolling, and SIMD work-item execution. Both Spector and OpenDwarfs make use of arrays to keep track of the active vertices. Hassan et al. propose a bitmap-based implementation of OpenDwarfs and achieve up to 5× improvement over an architecture-agnostic OpenDwarfs kernel on synthetic graphs [11]. Chen et al. propose an edge-centric BFS with multiple PEs as well as a novel data-shuffling technique to handle run-time dependencies caused by data dispatch across multiple PEs [6]. OBFS [19] further improves the performance over [6] by 5.5×. OBFS makes use of techniques such as graph re-ordering, data alignment, and overlapping the apply stage with other tasks [19].

The implementation described in this thesis differs from prior implementations in two ways. First, the bitmaps used in [11] and arrays in [10, 17] reside in the global memory. The optimizations described in this work avoid expensive global-memory accesses by maintaining the data structures in the local memory. A queue-based implementation is explored in addition to an array-based implementation. Unlike OBFS, the three stages (filter-apply-expand) are merged in a single kernel, thus avoiding expensive kernel launch overhead. OBFS makes use of shared virtual memory for data transfer during BFS iterations. The implementation of the BFS kernel described in this work does not require any additional data transfers from the CPU after it is invoked.

Chapter 3

Implementation and optimization of BFS

This chapter is organized as follows. §3.1 describes the queue-based and array-based implementation of BFS. §3.2 summarizes the process of high-level synthesis via OpenCL. In §3.3, the optimizations for improving the performance of BFS for sparser graphs are discussed. Finally, in §3.4, two contrasting optimization strategies for sparser and denser graphs are presented.

3.1 Implementation Details

This work follows the filter-apply-expand approach and uses vertex-centric BFS. For a graph $G(V, E)$, where V is the set of vertices and E is the set of edges between the vertices, BFS computes the hop-count between the source vertex S and all other vertices of the graph. In the first iteration of BFS, the active set has only one element: the source vertex. The filter-apply-expand process terminates when the size of the active set remains zero after the completion of the expand stage. The active set size becomes zero when all the vertices in the path of the source vertex are processed. §3.1.1 describes the data structures used to represent the active set.

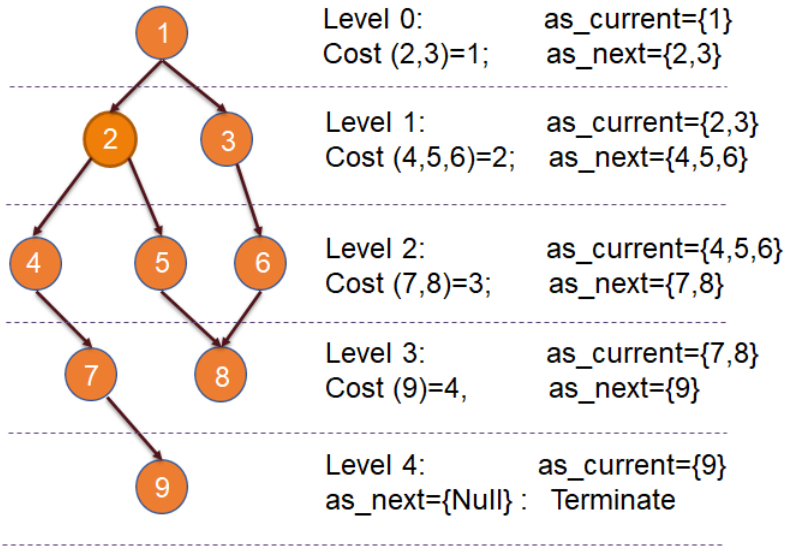

Figure 3.1: Queue-based traversal

3.1.1 Active set representation

The active set can be represented in two ways. It is possible to use a data structure such as a queue to maintain the set of active vertices [29]. The second approach involves arrays where the vertex is active if logic one is stored at the index that is the same as the vertex number [4, 10, 17].

3.1.1.1 Queue-based BFS

Figure 3.1 shows an example of a queue-based traversal. Active set vertices for current and next iteration, (as_current, as_next in Fig. 3.1) are stored in two different queues. In a typical queue-based implementation, vertices are added and removed using enqueue and dequeue operations. Queue-based traversal takes place, as shown in Algorithm 1. While this approach works on CPUs, it cannot be implemented in the same way using OpenCL. It is not possible to implement operations such as enqueue or dequeue as dynamic memory management is not supported by OpenCL. However, it is possible to mimic the behavior of

the queue by using arrays. In this work, two arrays represent the queue for storing active vertices and their neighbor set. Instead of performing push and pop operations, this work uses local variables that point to the first and last added entries in the array. The algorithm terminates when no new vertices are added to the array that stores neighbors. In this work, the arrays are entirely stored in the local memory of the FPGA, and expensive global-memory accesses are avoided.

Algorithm 1 BFS implementation using queue

Require: Graph $G(V, E)$, Source S, Vertices n
1: Create queues for current and next iterations *cqueue*, *nqueue*;
2: Create an array to keep track of visited vertices $visited[n]$
3: Create an array to store the cost *cost*
4: *cqueue*.enqueue(S);
5: **while** $sizeof(cqueue) \neq 0$ **do**
6: **for each** vertex $v \in cqueue$ **do**
7: **for each** neighbor $v2$ of v **do**
8: **if** $visited[v2] \neq 0$ **then**
9: $visited[v2] \leftarrow 0$
10: $cost[v2] \leftarrow cost[v] + 1$
11: *nqueue*.enqueue($v2$)
12: **end if**
13: **end for**
14: *cqueue*.dequeue(v)
15: **end for**
16: swap cqueue and nqueue
17: **end while**

3.1.1.2 Array-based BFS

Active set vertices can be represented using arrays. Figure 3.2 shows an example of array-based traversal. Active set vertices for current and next iteration, (as_current, as_next in Fig. 3.2) are stored in two different arrays. Typically, arrays with a size equal to the number of vertices are initialized. A vertex v is active if the value at the index v of the array is one. It is possible to use a bitmap instead of an array. One integer array element is four

bytes long and therefore can hold information of up to 32 vertices. The active set can be filtered from the bitmap by performing level shift and logical XOR, AND, and NOT operations. Hassan et al. [11, 19] make use of bitmap, whereas Krommydas et al. [17] and Gautier [10] use arrays. Array-based traversal takes place, as shown in Algorithm 2. The choice of vertex-centric traversal is intentional. Typically, a graph has a significantly large number of edges compared to the number of vertices. An edge-centric implementation with a local memory-resident data structure for the active edges would require large local memory for many graphs. The vertex-centric approach is used in this work to produce results for considerably large graphs.

Algorithm 2 BFS implementation using arrays

Require: Graph $G(V, E)$, Source S, Vertices n
1: Initialize arrays $visited[n]$, $mask[n]$, $update[n]$
2: Create an array to store the cost $cost$
3: $cost[S] \leftarrow 0$
4: $mask[S] \leftarrow 1$
5: **while** $stopbfs \neq 0$ **do**
6: $stopbfs \leftarrow 0$;
7: **for each** $v \in G$ **do**
8: **if** $mask[v] \leftarrow 1$ **then**
9: $mask[v] \leftarrow 0$
10: **for each** neighbor v2 of v **do**
11: **if** $visited[v2] \neq 0$ **then**
12: $cost[v2] \leftarrow cost[v] + 1$;
13: $update[v2] \leftarrow 1$;
14: **end if**
15: **end for**
16: **end if**
17: **end for**
18: **for each** vertex $v \in G$ **do**
19: **if** $update[v] \leftarrow 1$ **then**
20: $(visited[v], mask[v], stopbfs) \leftarrow 1$
21: $update[v] \leftarrow 0$
22: **end if**
23: **end for**
24: **end while**

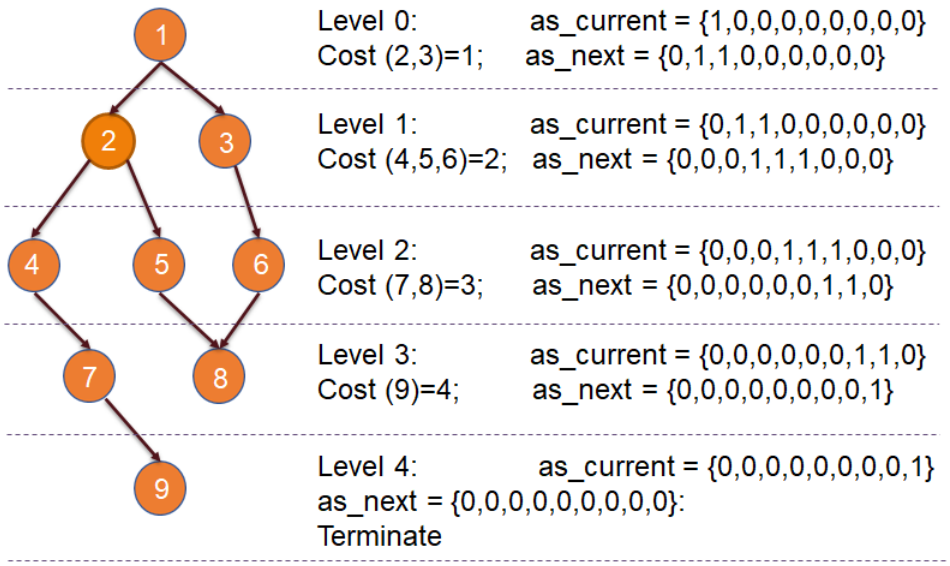

Figure 3.2: Array-based traversal

3.1.2 Implementation of BFS in OpenCL

OpenCL, as a form of HLS, supports a bifurcated programming model with host code and kernel code. The host code runs on the CPU. It handles tasks such as data management between the CPU and FPGA, synchronization between two kernels, and result collection. A kernel is the part of the application program that runs on the target device. OpenCL allows programmers to invoke the kernel in one of the two configurations: NDRange (NDR) and single-task (ST).

In the NDRange configuration, kernel execution occurs using a group of work items, also known as work-groups, that execute the kernel in a parallel. The hardware scheduler is responsible for dispatching the work-groups to the implemented hardware. In a single-task kernel launch, only a single work item is used. However, this configuration allows pipelined execution within the kernel loops as opposed to the pipeline of work items from NDRange. Figure 3.3 shows the differences in the code structure of the NDRange and single-task kernels. The NDRange kernel, as shown in Fig. 3.3a requires a thread identifier function (for ex.

`get_global_id()`) for each work item. This function is replaced by a `for` loop in a single-task kernel where a single work item is responsible for the execution of all iterations, as shown in Figure 3.3b.

In a level-synchronous BFS, the kernel can be invoked in both NDRange and single-task configurations. When launched as NDRange, a host-based synchronization is necessary to ensure correctness, as shown in [6, 10, 11, 17]. The NDRange configuration allows the use of attributes for compute-unit replication. In the single-task configuration, it is possible to implement local memory optimizations. OBFS makes use of a single-task configuration [11, 19].

```
__kernel void kernel1(
__global Node* restrict h_graph_nodes,
__global int* restrict h_graph_edges,
__global int* restrict h_graph_mask,
__global int* restrict h_updating_graph_mask,
__global int* restrict h_graph_visited,
__global int* restrict h_cost_ref,
__global int no_of_nodes)
{
    int tid = get_global_id(0);
    if(tid<no_of_nodes && h_graph_mask[tid] != 0)
        {
        .... //BFS kernel
        }
}
```

(a) NDRange kernel for BFS

```
__kernel void kernel1(
__global Node* restrict h_graph_nodes,
__global int* restrict h_graph_edges,
__global int* restrict h_graph_mask,
__global int* restrict h_updating_graph_mask,
__global int* restrict h_graph_visited,
__global int* restrict h_cost_ref,
__global int no_of_nodes)
{
    int tid;
    for (tid=0; tid<no_of_nodes; tid++)
    {
        if(h_graph_mask[tid] != 0)
        {
        .... //BFS kernel
        }
    }
}
```

(b) Single-task kernel for BFS

Figure 3.3: Difference in the structure of NDRange and single-task kernels

3.2 High-level Synthesis via OpenCL

In recent years, OpenCL has emerged as an HLS framework for rapid application development, with both Intel [15] and Xilinx [30] having toolchains that support OpenCL runtime. This section summarizes the programming model of OpenCL along with its use for high-level synthesis for FPGA.

OpenCL allows programmers to create programs that can be compiled and executed across a diverse set of target architectures, called devices, such as CPUs, GPUs, DSPs, and FPGAs. A portion of the OpenCL code performs tasks such as memory allocation, data transfer between the host CPU and the target device, and synchronization. This code is called host-code. The portion of the code that runs on the target architecture is called the OpenCL kernel.

The OpenCL framework contains two sets of APIs – the platform layer API and the runtime API. The platform layer API is run on the host CPU and enables the discovery of one or more target devices. The platform API allows programmers to select and initialize the devices they want to use. The runtime API enables the kernels to be compiled for the devices on which they are going to run. The runtime API is also used to gather the results after the kernel finishes its execution. OpenCL is based on C99 and is defined as part of the OpenCL specification [23].

OpenCL gives programmers control over where kernels are run, how the memory is allocated, and how the devices and host CPU synchronize their operations to ensure correctness. Several FPGA-specific attributes for performance optimization are also included in the part of Intel's vendor-specific extensions to OpenCL [14]. The optimizations explored in this thesis are described in §3.3.

3.3 Optimizations

This section describes the optimization employed to speed up the BFS. The optimizations are categorized into three groups: OpenCL framework-specific optimizations, FPGA architecture-aware optimizations, and application-specific optimizations.

3.3.1 OpenCL framework-specific optimizations

The use of a single-task kernel launch configuration allows the implementation of OpenCL framework-specific optimizations discussed below.

3.3.1.1 Kernel fusion

While the NDRange configuration necessitates synchronization during the filter and expand stages, a single-task configuration does not require the synchronization between the two stages due to the sequential execution of a single work-item. Therefore, it is possible to merge all the BFS stages into a single kernel in a single-task configuration. Kernel fusion reduces the number of synchronization calls made using `clFinish` between the BFS stages.

3.3.1.2 Elimination of host-based synchronization

Level-synchronous BFS necessitates synchronization between compute units at the end of each iteration. As of now, OpenCL does not have synchronization primitives for OpenCL devices. The only way to ensure synchronization is by using `clFinish` at the host or CPU, as shown in Fig. 3.4. Host-based synchronization can adversely affect the performance of BFS for graphs with a large number of levels. There is no need for the host-based synchronization in a single work-item BFS kernel, which is effectively a sequential execution

of BFS. Therefore, this work avoids host-based synchronization altogether by inserting an outermost while loop in the single-task kernel. The loop terminates when the number of active vertices is zero. Insertion of an outermost while loop in the kernel eliminates the need for a while loop in the host code that governs the execution of BFS, along with the FPGA to CPU data transfer operations for determining its exit condition. This optimization improves the performance by avoiding the host-based synchronization and reducing the kernel launch overhead. A non-zero kernel launch overhead is caused because of the slight difference in the time when a BFS iteration is scheduled for execution on an FPGA and when the execution of a BFS iteration on an FPGA begins. With the insertion of the outer while loop in the kernel, all BFS iterations are performed in a single kernel launch as opposed to the kernel launches for each iteration. As a result, kernel launch overhead is minimized as the kernel needs to be launched once.

The two framework-specific optimizations discussed above pave the way for architecture-aware optimizations discussed in the next subsection.

3.3.2 Architecture-aware optimizations

While OpenCL presents a ”write once, run anywhere” programming paradigm targeted at a diverse set of architectures, extracting performance out of specific hardware via OpenCL necessitates architecture-aware optimizations. This section describes how the baseline single work-item implementation is modified to achieve high performance on the Stratix 10 FPGA.

3.3.2.1 Local queue

In the array-based BFS, active vertices are found by searching all the elements of an array. In a queue-based implementation, vertices are active if they are present in the queue. While the

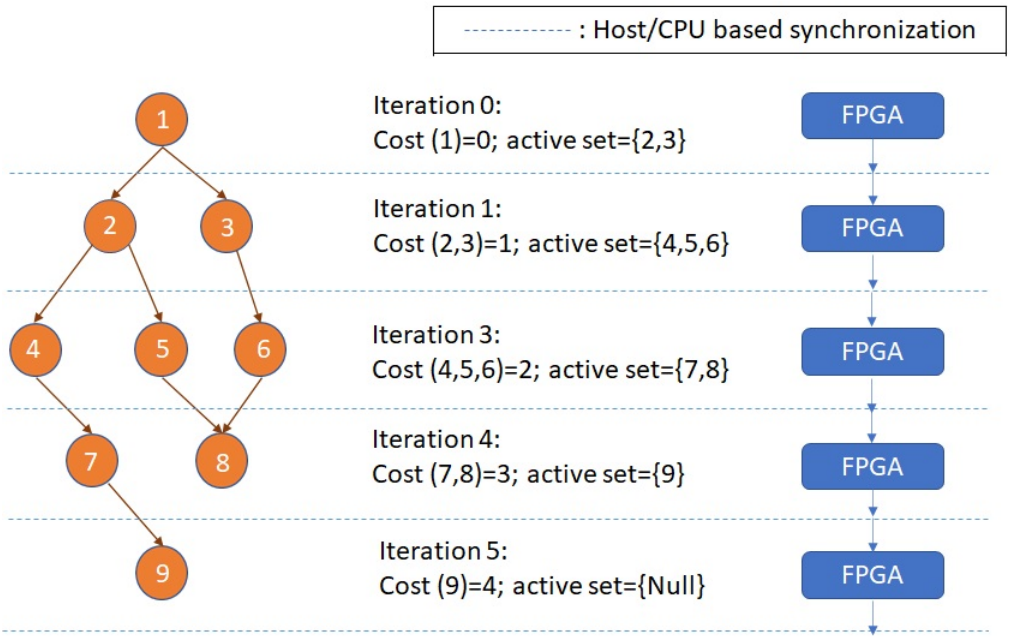

Figure 3.4: BFS with host-based synchronization

queue eliminates the need for redundant comparisons, it still necessitates expensive irregular reads and writes to the global memory. This work mitigates the expensive irregular global accesses by initializing the queue in the local memory (BRAM) of FPGA. This puts a limit on the size of the graphs that can be processed as BRAMs are limited. However, with the application-specific optimizations discussed in §3.3.3, it is possible to process graphs with up to 2^{21} vertices (or 2,097,152 vertices) on the target Stratix 10 FPGA. In addition to the local memory-based queue, a local array-based BFS from the Algorithm 2 is explored. §4.1 compares the performance of two BFS variants.

3.3.2.2 Using multiple memory banks

Kernel performance can be further improved by specifying the number of banks in the local queue [13, 14]. Having multiple banks allows parallel accesses to the queue, as shown in Fig. 3.5. Even when two different elements of the same array are accessed in different

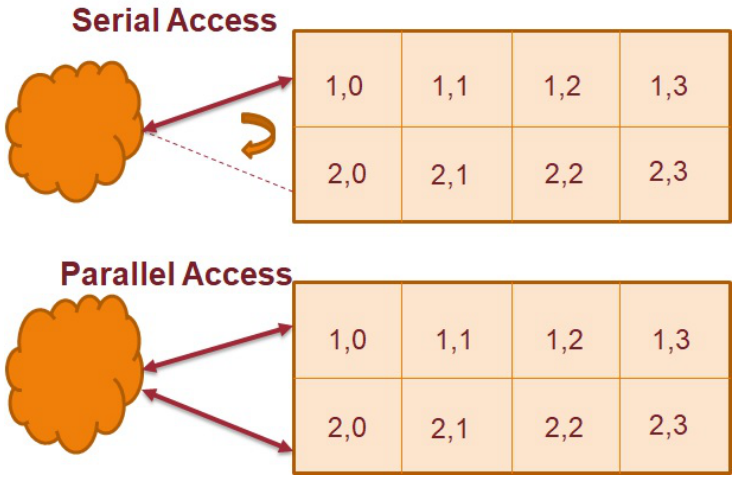

Figure 3.5: Parallel accesses using memory banks

pipeline stages, there could be stalls in the pipeline due to contention for memory access. Parallel accesses, in turn, reduce the pipeline stalls in such cases.

3.3.2.3 Speculated iterations

The performance of pipelined loops can be improved by specifying the number of speculated iterations before the execution of the loop. The offline compiler generates the hardware to run `N` more iterations of the loop while ensuring that extra iterations do not affect the correctness of the results [14]. Speculated iterations can reduce the loop initiation interval and increase the frequency [14]. The value of `N` must be carefully chosen. If the exit condition of the loop is calculated in very few iterations, the redundant iterations performed after the exit condition can negatively impact the performance. In this work, the speculated iteration count is set to five (5).

3.3.2.4 Constant work-group sizes

In the NDRange kernels, the hardware scheduler dispatches work-groups to the compute units. Knowledge of the work-group size at compile-time allows the compiler to perform aggressive optimizations to match the kernel to hardware resources without any redundant logic [13]. The compiler attribute `reqd_work_group_size` is used to instruct the compiler to allocate the correct amount of hardware to manage the execution of each work-group. Specifying the `reqd_work_group_size` attribute prevents the compiler from generating additional hardware to support work-groups of unknown sizes.

3.3.2.5 Compute-unit replication

Intel's compiler for FPGA can generate multiple compute units for each kernel. Generating multiple compute units allows the FPGA to achieve higher throughput [13]. For example, Fig. 3.6 shows the interaction of four compute units with the global memory of the FPGA device. The compiler implements each compute unit as a unique pipeline. Each compute unit can execute multiple work-groups in parallel. The hardware scheduler in the FPGA dispatches work-groups to available compute units. The compiler does not determine the optimal number of compute units for a given application. Instead, the programmer has to decide the optimal number of compute units for a target application. The attribute `num_compute_units(N)` is used to instruct the compiler to generate `N` compute units for a given kernel. Increasing the number of compute units achieves higher throughput. However, the performance improvements due to replication of compute units come at the expense of increased hardware resource utilization.

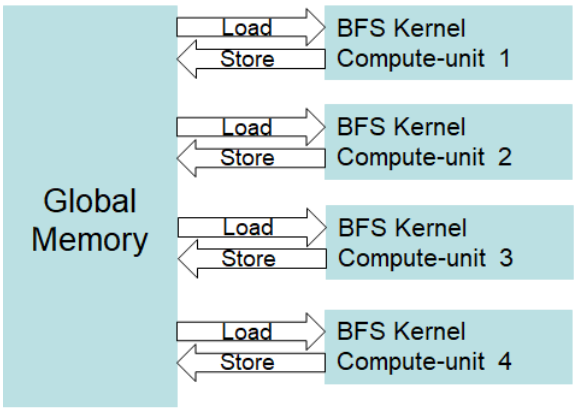

Figure 3.6: Compute-unit replication by a factor of 4

3.3.3 Application-specific optimizations

Architecture-agnostic kernels of OpenDwarfs and their optimized variants in the Spector provide a baseline for FPGA, but there remains scope for the application-specific optimizations discussed below.

3.3.3.1 Merged apply and expand

The apply stage of BFS can be merged with the expand stage. Instead of reading the cost of a vertex from global memory and writing the updated cost back to it, a local variable is used to keep track of the current level. At each iteration, the value of the level is assigned to the neighbors of active vertices. The value of this variable is incremented by one at the end of the expand stage. OBFS [19] also employs level-based cost update. This optimization improves performance in two ways. Firstly, it avoids the expensive global read of the dependent variable in the loop that stalls the pipeline in the filter stage. The initiation interval (II, for short) is the number of clock cycles that the pipeline must stall before it can process the next loop iteration. The initiation interval for the loop in the filter stage becomes one due to this step, as shown in Table 3.1. Secondly, it avoids redundant cost updates that take

Table 3.1: Initiation interval (II) for the loop in the filter stage

Implementation	II (# clock cycles)
Single-task OpenDwarfs [17] with array	237
Single-task OpenDwarfs [17] with bitmap	225
This work (Using local queue and a copy of level)	1

place in Algorithm 2 at line 10 when two vertices from the active set have shared neighbors. Merging apply and expand stages ensures that no redundant updates take place when two vertices have shared neighbors.

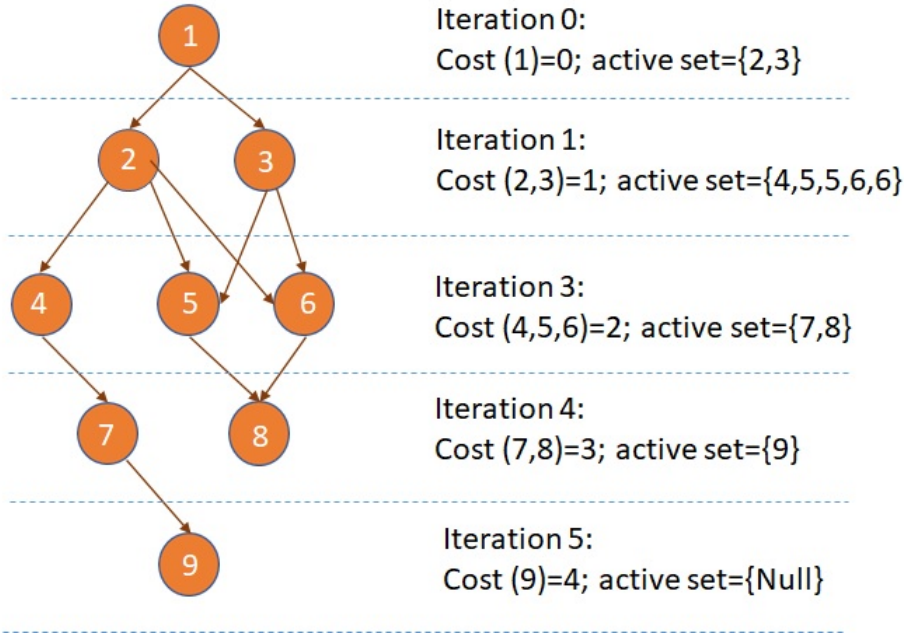

Figure 3.7: Duplicate entries in the active set

3.3.3.2 Elimination of duplicate entries

When two or more vertices have a shared neighbor, the shared neighbor may get inserted into the queue multiple times, as shown in Fig. 3.7. The array "visited" in Algorithm 1 eliminates the introduction of duplicate entries in the queue. However, using the array puts a limit on

the size of the queue as local memory is limited. Using two arrays for the queue and one array to keep track of visited vertices allows graphs with up to 2^{20} vertices (or 1,048,576 vertices) to be processed on the target FPGA used in this work. This array is replaced with a bitmap to allocate more space for the queue, as shown in Figure 3.8. The use of a bitmap instead of an array allows graphs with up to 2^{21} vertices (or 2,097,152 vertices) to be processed on the target FPGA.

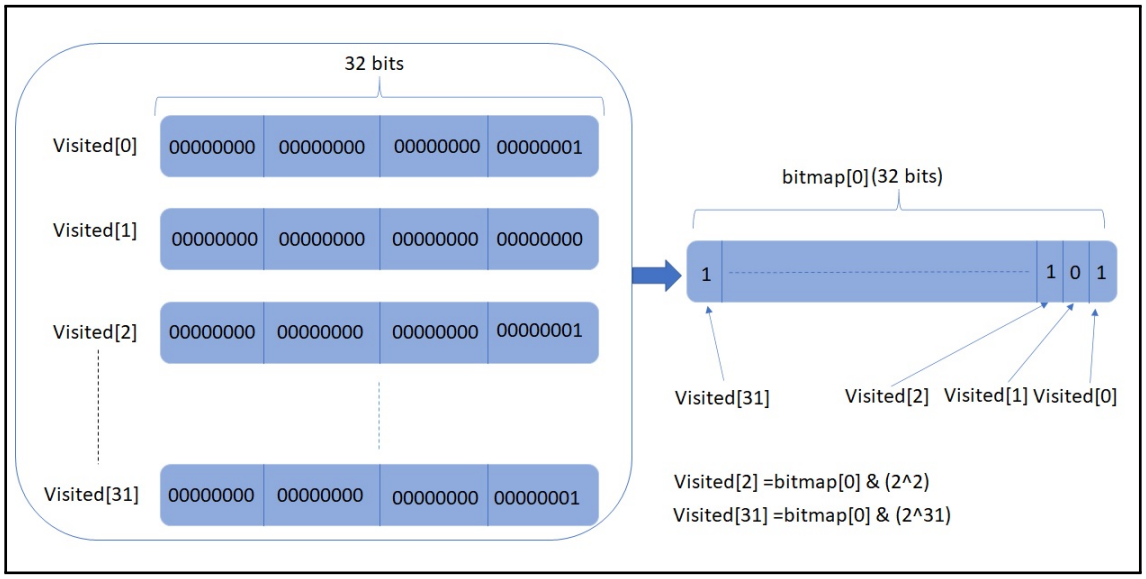

Figure 3.8: Storing data in bitmaps

3.4 Optimization Strategies

This thesis explores two optimization approaches – one for the single-task configuration (ST) and the other for NDRange configuration (NDR). For the single-task kernel, a baseline single-task implementation which makes use of optimizations from §3.3.1.1, §3.3.1.2, §3.3.2.3, §3.3.3.1, and §3.3.3.2 is used as a starting point. This work then evaluates the per-

formance of a number of variants of baseline single-task kernel that implement local memory optimizations from §3.3.2.1 and §3.3.2.2.

The baseline implementation for NDRange configuration makes use of optimizations from §3.3.3.1, §3.3.2.5, and §3.3.2.4. To evaluate the impact of compute-unit replication, the number of compute units is varied, and the performance of this approach on a diverse set of graphs is evaluated. The NDRange implementation requires multiple invocations of BFS kernels. Data stored in the local memory does not remain persistent between the successive invocations of NDRange BFS kernels. Therefore any local memory optimizations in the NDRange configuration are not implemented in this work.

The single-task configuration does not support the compiler attribute for compute-unit replication. However, it allows the use of local memory for storing the data structures as the BFS execution takes place in a single invocation of the BFS kernel. At a high level, single-task implementation is equivalent to a serial implementation of BFS, but it leaves scope for a number of local memory optimizations. The NDRange implementation is equivalent to a parallel implementation of BFS that is realized using compute-unit replication. The optimizations in single-task configurations are expected to work well for sparser graphs, while the optimizations in the NDRange configuration are expected to work well for denser graphs.

3.4.1 Productivity comparison between NDRange and single-task implementations

This thesis presents optimizations for BFS in both NDRange and single-task configuration of OpenCL. In terms of productivity, implementing single-task kernels with local memory optimizations required more effort than implementing NDRange kernels and associated optimizations. The local queues and bitmaps discussed in §3.3.2.1 were manually implemented

in single-task kernels. In contrast, writing NDRange kernels with fixed local work-group sizes and compute-unit replication, as discussed in §3.3.2.4 and §3.3.2.5, respectively, was relatively straightforward. The optimizations discussed in §3.3.2.4 and §3.3.2.5 were implemented by inserting compiler attributes, which effectively required one additional line of code for both optimizations.

Chapter 4

Performance Evaluation

This chapter presents the performance evaluation of the optimization strategies described in §3. Along with the performance evaluation, this chapter categorizes the graph dataset using metrics such as average degree and Gini index [18] along with the performance of BFS for the two contrasting optimization strategies described in §3.4.

4.1 Performance Evaluation

4.1.1 Experimental setup

The experiments were performed on Stratix 10 SX 2800 FPGA. Intel(R) Xeon(R) Gold 6128 CPU with an operating frequency of 3.4 GHz was the OpenCL host. Graphs used in the experiments were taken from network repository [25] and are shown in Table 4.1. The reference implementation of BFS is taken from [16] with commit SHA b536909. The performance of reference implementation and other NDRange implementations is measured by gathering timing information before and after the while-loop where BFS kernel invocations take place. In all other single-task implementations, the performance is measured by getting timing information before and after the kernel invocation command.

Table 4.1: Graph dataset used in evaluation

Graph	Application Area	#Vertices	#Bidirectional Edges	Average Degree
Luxembourg-OSM	Road Network	0.1M	0.2M	2
Inf-Roadnet-PA	Road Network	1.1M	3.0M	3
Ecology1	Landscape Ecology	1M	1.9M	2
Roadnet-CA	Road Network	2M	5.6M	3
G3-Circuit	Circuit Simulation	1.6M	3M	2
Sc-msdoor	Scientific Computing	0.4M	18.8M	45
Sc-ldoor	Scientific Computing	0.9M	20.8M	43
soc-youtube	Social Media	0.4M	1.9M	7
Citation-DBLP	Collaboration Network	0.5M	30.4M	56
Hollywood-2009	Collaboration Network	1.1M	112.6M	105

4.1.2 Approach I – optimizations for the single-task configuration

Along with the reference implementation from Spector, this work evaluates the performance of six (6) implementations in single-task configuration (ST), with variations in data structures [Arrays / Queue + Array (Q + A) / Queue + Bitmap (Q + BM)], and number of memory banks (B). The performance of six different implementations is compared with Spector [10]. All kernels other than Spector use OpenCL framework-specific optimizations discussed in §3.3.1 and are invoked as single-task kernels. These implementations are discussed below.

4.1.2.1 ST

This is the baseline single-task implementation of the BFS kernel. It is based on the OpenDwarfs [17] BFS implementation. It is an array-based BFS implementation from Algorithm 2, where the arrays reside in the global memory of the FPGA device. This implementation differs from OpenDwarfs in three ways. First, apply and expand steps are merged, as discussed in § 3.3.3.1. Second, the number of speculated iterations is set to five(5). Third,

the OpenCL-framework specific optimizations from §3.3.1.1 and §3.3.1.2 are implemented, which are not implemented in the reference BFS implementation in OpenDwarfs.

4.1.2.2 ST + Q + A

This implementation uses the architecture-aware and OpenCL framework-specific optimizations from the baseline ST implementation. The difference lies in the data structures used for storing active vertices. ST + Q + A is a queue-based implementation where active vertices are stored in the queue. An array is used to keep track of all the visited vertices during the progression of BFS. A key difference between this implementation and the baseline single-task implementation is that both the queue and array are stored entirely in the local memory (BRAM) of the FPGA. A relatively smaller size of local memory puts a limit on the sizes of the queue and array that can be stored in the local memory. Limited queue size, in turn, puts a limit on the size of the graph that can be processed with this implementation. This implementation can process graphs with up to 2^{20} or 1,048,576 vertices for the target FPGA. Graphs RoadNet-CA, G3-circuit, and hollywood-2009 from the Table 4.1 cannot be processed in this implementation because of insufficient local memory.

4.1.2.3 ST + Q + BM

This implementation is similar to the one discussed in §4.1.2.2. It is a queue-based implementation. The only difference is that instead of arrays, this implementation makes use of a bitmap to keep track of all the visited vertices during the progression of BFS. Both queue and bitmap reside in the local memory of the FPGA. The use of a bitmap instead of an array increases the limit on the size of the graphs that can be processed in this kernel. For the target Stratix 10 SX FPGA, this implementation can process graphs with up to 2^{21} or

2,097,152 vertices. Graphs RoadNet-CA, G3-circuit, and hollywood-2009 from the Table 4.1, which could not be not processed in §4.1.2.2 can be processed in this implementation.

4.1.2.4 ST + Q + BM + 2B

Along with the optimizations used in §4.1.2.2, in this implementation, the number of memory banks on the local queue and bitmap is set to two.

4.1.2.5 ST + Q + BM + 4B

The implementation is similar to the one described in §4.1.2.3. The only difference is that the number of memory banks is increased to four.

4.1.2.6 ST + Q + BM + 8B

The implementation is similar to the one described in §4.1.2.3. The only difference is that the number of memory banks is increased to eight.

4.1.3 Evaluation of implementations from the approach I

For all six implementations described above, the average of the kernel runtime over ten runs is reported. In addition to kernel performance, the resource utilization results for each implementation are also reported.

Table 4.2: Resource utilization summary

Implementation	Resource utilization		Frequency (MHz)
	Logic Utilization	RAMs	
Spector [10]	287952 (31%)	1912 (16%)	208
ST	231351 (25%)	997 (9%)	253
ST + Q + A	228610 (24%)	6879 (59%)	71
ST + Q + BM	226834 (23%)	8983 (77%)	108
ST + Q + BM + 2B	235114 (25%)	9057 (77%)	123
ST + Q + BM + 4B	235276 (25%)	9057 (77%)	147
ST + Q + BM + 8B	235678 (25%)	9057 (77%)	149
NDR + REQ + CU2	254066 (27%)	1473 (13%)	216
NDR + REQ + CU4	302242 (32%)	2229 (19%)	225

4.1.3.1 Resource utilization analysis

Resource utilization and frequency report is shown in Table 4.2. The operating frequency is higher for baseline single-task BFS implementation, and it is considerably lower for the queue-based implementations. In the three local memory-based implementations with bitmaps, RAM usage is considerably higher than in Spector, 77% in the queue-based kernel with bitmaps, compared to 16% in Spector. Compared to the queue-based kernel, Spector has 6% more logic utilization. This is expected as Spector makes use of optimizations such as compute-unit replication.

4.1.3.2 Performance improvement

Runtime of the implementations discussed in §4.1.2 is shown in the Table 4.3. Local queue-based implementations with bitmap outperform the queue-based implementations with array by a factor of 1.3 on average. ST + Q + BM + 8B outperforms all other single-task implementations for almost all of the graphs. It outperforms Spector for sparser graphs such as Luxembourg-osm and ecology1, where a speedup of 22.3× and 21.8× is observed, respectively.

Table 4.3: Performance evaluation of optimization strategies for single-task configuration

Graph	Runtime(Seconds) of Implementation							Speedup of (ST+Q+BM+8B) over Spector
	Baseline (Spector [10])	ST	ST + Q + A	ST + Q + BM	ST + Q + BM + 2B	ST + Q + BM + 4B	ST + Q + BM + 8B	
Luxembourg-OSM	1.731	2.357	0.186	0.154	0.113	0.088	0.078	**22.3**
Inf-Roadnet-PA	6.249	11.703	2.265	1.858	1.577	1.106	1.008	**6.2**
Ecology1	20.679	39.666	2.241	1.687	1.494	1.109	0.946	**21.8**
Roadnet-CA	10.901	21.563	Insufficient BRAM	3.229	2.95	1.938	1.873	**5.8**
G3-Circuit	8.314	16.186	Insufficient BRAM	2.672	2.243	1.542	1.441	**5.7**
Sc-msdoor	0.472	1.709	1.358	0.895	0.886	0.712	0.697	**1.9**
Sc-ldoor	1.353	3.498	3.026	1.997	1.984	1.551	1.561	**1.9**
soc-youtube	0.114	1.558	1.167	0.815	0.774	0.623	0.618	0.2
Citation-DBLP	0.298	3.27	2.029	1.424	1.329	1.033	1.046	0.3
Hollywood-2009	1.201	4.706	Insufficient BRAM	3.308	3.606	2.884	2.818	0.4

4.1.3.3 Impact of banking

The presence of two banks in the local queue improves the performance by a factor of 1.28 on average. Performance improvements are due to parallel accesses to local memory and increased frequency. A marginal increase in the performance by increasing the number of banks to 4 and 8 is observed. Scheduled Fmax for the queue-based kernel with eight banks is 149 MHz compared to 108 MHz without banking. For all the sparser graphs in Table 4.1 with average degree <3, a similar trend is observed where queue-based implementation with bitmap and eight banks shows the most improvement over Spector.

4.1.3.4 Impact of sparsity

Table 4.3 shows the speedup for the best performing implementation of this approach over Spector for the sparser graphs. Denser graphs such as Hollywood-2009 from Figure 1.2 benefit from parallel designs. It is observed that Spector outperforms ST + Q + BM + 8B for denser graphs; this is expected since any parallel design is not implemented in this approach. Sparser graphs benefit the most from the optimizations used in ST + Q + BM

+ 8B. A notable characteristic of sparser graphs is that they process a small number of vertices in each iteration. Queue-based traversal is preferable over bitmap/array in such cases as it avoids the need to iterate over the entire array to find active vertices. A small amount of workload in individual iterations also means that parallel hardware cannot be utilized effectively. The optimizations explored for sparser datasets deliver 5.7× to 22.3× improvement over Spector [10].

4.1.4 Approach II – optimizations for the NDRange configuration

The reference implementation, Spector [10], uses NDRange configuration (NDR) with compiler attributes for SIMD vectorization and compute-unit replication. Spector, as well as the NDRange implementations from this work, use global arrays. Using queues in the NDRange configuration would necessitate the use of atomic operations. Atomic operations significantly reduce the performance of OpenCL kernels on FPGA, as explained in [13]. Therefore, the performance of BFS with concurrent queues is not explored in this work. In addition to Spector, this work evaluates the performance of two(2) other implementations of NDRange kernels with work group size fixed to 256 at compile time using the attribute `req_work_size(N)` (REQ), and with variations in the number of compute units (CU).

4.1.4.1 NDR + REQ + CU2

This implementation is similar to Spector; it uses NDRange configuration, along with the compile-time constant work-group size, which is set to 256. A host-based synchronization is required between the execution of filter and expand stages. Number of compute units is set to two using attribute `num_compute_unit(2)`.

4.1.4.2 NDR + REQ + CU4

This implementation is similar to NDR + REQ + CU2. The only difference is that the number of compute units is set to four. A similar implementation with eight compute units was scheduled for compilation. However, it was observed that increasing the compute units to eight caused timing violations, and the compiler was not able to generate the hardware logic because of insufficient RAM blocks.

4.1.5 Evaluation of implementations from approach II

This section presents the resource utilization summary and performance evaluation of the NDRange implementations.

4.1.5.1 Resource utilization analysis

Table 4.2 shows the resource utilization and operating frequency of the NDRange implementations from this work. Both NDRange implementations operate at a higher frequency than Spector. Of all the tested implementations, the NDRange kernel with four compute units has the highest logic utilization at 32%, and the same implementation has more RAM usage compared to Spector.

4.1.5.2 Performance improvement

Runtimes of the implementations discussed in §4.1.4 are shown in the Table 4.4. Implementation with four compute units outperforms the other two implementations for all the input graphs.

Table 4.4: Performance evaluation of optimization strategies for NDRange configuration

Graph	Runtime(Seconds) of Implementation			Speedup of (NDR +REQ +CU4) over Spector
	Baseline (Spector [10])	NDR + REQ +CU2	NDR +REQ +CU4	
Luxembourg-OSM	1.731	0.887	0.436	**3.9**×
Inf-Roadnet-PA	6.249	3.254	1.473	**4.2**×
Ecology1	20.679	9.687	4.845	**4.3**×
Roadnet-CA	10.901	5.906	2.557	**4.3**×
G3-Circuit	8.314	3.035	1.954	**4.4**×
Sc-msdoor	0.472	0.287	0.237	**2**×
Sc-ldoor	1.353	0.779	0.587	**2.3**×
soc-youtube	0.114	0.1	0.096	**1.2**×
Citation-DBLP	0.298	0.274	0.252	**1.2**×
Hollywood-2009	1.201	1.121	1.041	**1.1**×

4.1.5.3 Impact of compute-unit replication

Compared to Spector, the NDRange implementation with two compute units along with a work-group size of 256 is faster by a factor of 1.1× to 2.2× depending on the graph. Increasing the number of compute units to four further improves the performance by a factor of 1.1× to 2×. Performance improvements due to compute-unit replication come at the cost of increased resource utilization, as shown in Table 4.2. Overall, using four compute units delivers a speedup of 1.1× to 4.4× over Spector. It is observed that sparser graphs such as road networks benefit the most from compute-unit replication. However, for the same graphs, the optimized single-task implementation outperforms the NDRange implementation with four compute units. To further investigate the impact of compute-unit replication, all three implementations are profiled for the average bandwidth for each graph, using Intel's dynamic profiler for FPGA. Table 4.5 shows the measured average bandwidth during the execution of BFS for each graph. Compute-unit replication increases the average bandwidth for the sparser graphs. The average bandwidth for the BFS execution of denser graphs such as hollywood-2009 is relatively higher, even without compute-unit replication. For such graphs, compute-unit replication has a very limited impact on the average bandwidth.

Table 4.5: Measured average bandwidth during the execution of BFS

Graph	Average Bandwidth (GB/s)		
	Baseline (Spector)	NDR + REQ +CU2	NDR +REQ +CU4
Luxembourg-OSM	1.235	2.034	2.979
Roadnet-PA	1.325	2.388	3.611
Ecology1	1.242	2.188	3.424
Roadnet-CA	1.312	2.323	3.658
G3-Circuit	1.324	2.413	3.76
Sc-msdoor	3.855	4.175	4.728
Sc-ldoor	3.188	4.404	4.928
soc-youtube	6.078	6.129	6.269
Citation-DBLP	7.465	7.698	7.72
Hollywood-2009	8.655	8.723	8.904

§4.2 compares the performance of the most optimized implementations from the single-task and NDRannge optimization strategies and describes metrics to classify the graphs based on the runtime for both implementations.

4.2 Workload Classification

From the performance evaluation of the two approaches discussed in §4.1.2 and §4.1.4 respectively, it can be observed that the best performing implementations of both approaches do not improve the performance for all the graphs in the same way. ST + Q + BM + 8B significantly improves the performance for sparser graphs. The reference implementation, Spector outperforms ST + Q + BM + 8B for the denser graphs. On the other hand, NDR + REQ + CU4 outperforms Spector for all the graphs, but it still performs slower than ST + Q + BM + 8B on the sparser graphs.

This section attempts to classify the graph datasets based on their performance when running ST + Q + BM + 8B and NDR + REQ + CU4. Classification of graph datasets requires information about both the performance as well as the properties of the graph. Gswitch [22]

Table 4.6: Best of approach I vs. best of approach II

Graph	Avg degree	Gini index	Runtime of (ST + Q + BM + 8B) (sec.)	Speedup of (ST+Q+ BM+8B) over Spector	Runtime of (NDR +REQ +CU4) (sec.)	Speedup of (NDR +REQ + CU4) over Spector
Luxembourg-OSM	2	0.068	**0.078**	**22.3**	0.436	3.9
Inf-roadnet-PA	2	0.187	**1.008**	**6.2**	1.473	4.2
Ecology1	2	0.0009	**0.946**	**21.8**	4.845	4.3
Roadnet-CA	3	0.1844	**1.873**	**5.8**	2.557	4.3
G3-Circuit	2	0.077	**1.441**	**5.7**	1.954	4.4
Sc-msdoor	45	0.142	0.697	1.9	**0.237**	**2**
Sc-ldoor	43	0.144	1.561	1.9	**0.587**	**2.3**
soc-youtube	7	0.7274	0.618	0.2	**0.096**	**1.2**
Citation-DBLP	56	0.544	1.046	0.3	**0.252**	**1.2**
Hollywood-2009	105	0.7346	2.818	0.4	**1.041**	**1.2**

uses a number of graph parameters such as average degree and Gini coefficient to decide the optimal optimization strategy for running a graph workload on a GPU. Similar to Gswitch, average degree and Gini coefficient are used to observe how graphs are sensitive to one of the two optimization strategies. These metrics are briefly discussed below.

4.2.1 Average degree

The degree of a vertex is the number of edges in which a given vertex is an endpoint. The average degree of a graph is the average number of edges per vertex in the graph. It is a measure of how connected the graph dataset is.

4.2.2 Gini index

Gini index or Gini coefficient is a measure of the inequality in a distribution. In the case of graphs datasets, it is a measure of inequality in the degree distribution of vertices [18]. The value of the Gini index lies in the interval [0,1). A Higher Gini index indicates greater inequality in the degree distribution of a graph dataset. A higher Gini index in a graph

Table 4.7: Measured throughputs of the best performing implementations

Graph	Throughput Millions of traversed edges per second (MTEPS)	
	Implementation	
	ST + Q + BM + 8B	NDR + REQ + CU4
Luxembourg-OSM	**3.051**	0.549
Inf-Roadnet-PA	**2.976**	2.036
Ecology1	**4.2**	0.887
Roadnet-CA	**2.989**	2.19
G3-Circuit	**4.162**	3.092
Sc-msdoor	26.972	**79.324**
Sc-ldoor	26.649	**70.868**
soc-youtube	6.245	**40.208**
Citation-DBLP	29.063	**120.634**
Hollywood-2009	39.957	**108.165**

implies that a small number of vertices have a very high degree, while most of the vertices have a low degree. A near-zero value of the Gini index implies that the degree of almost all vertices is effectively the same.

Table 4.6 shows the values of average degree, Gini Coefficient, and runtime of ST + Q + BM + 8B and NDR + REQ + CU4 respectively. Table 4.7 shows the throughput of both implementations in millions of traversed edges per second. A key observation from Table 4.6 is that the optimized single-task implementation outperforms the optimized NDRange implementation when both the average degree and Gini coefficient are low. For graphs with a low average degree and low Gini index, ST + Q + BM + 8B outperforms NDR + REQ + CU4. In contrast, for graphs with high Gini coefficient and average degree, NDR + REQ + CU4 outperforms ST + Q + BM + 8B.

Table 4.8: Average number of vertices processed per iteration of BFS

Graph	Avg degree	Gini Index	Avg #vertices processed per BFS iteration
Luxembourg-OSM	2	0.008	110.72
Inf-Roadnet-PA	2	0.187	2006.57
Ecology1	2	0.0009	500.25
Roadnet-CA	3	0.1844	3526.17
G3-Circuit	2	0.077	3084.59
Sc-msdoor	45	0.142	3187.28
Sc-ldoor	43	0.144	5109.76
soc-youtube	7	0.7274	35425.5
Citation-DBLP	56	0.544	36032.4
Hollywood-2009	105	0.7316	97193.3

4.2.3 Implications of average degree and Gini index

A graph with a near-zero Gini index is expected to have an approximately even degree distribution. A graph with a low Gini index and a low average degree typically involves processing considerably fewer vertices per iteration compared to the denser graphs, as shown in Table 4.8. The NDRange-based solutions do not work well for the sparser graphs because of the kernel launch overhead and low bandwidth utilization, as shown in Table 4.5. In the case of sparser graphs with low GI and low average degree, the average number of vertices processed per iteration is not large enough to effectively utilize the available bandwidth, resulting in low bandwidth utilization. In such cases, optimizations from the approach I (§4.1.2) deliver better performance compared to the NDRange-based optimizations as expensive global memory operations with low bandwidth get replaced with local memory operations with low latency. As the graphs get denser, indicated by either a high average degree or a high Gini index, implementation with compute-unit replication from approach II (§4.1.4) outperforms approach I. In the case of denser graphs, the bandwidth utilization is considerably higher in comparison with sparser graphs, as shown in Table 4.5.

Table 4.9: Comparison of the best perfoming FPGA implementations with an equivalent OpenMP implementation for CPU

Graph	Avg degree	Gini index	Runtime (Seconds)					
			Implementation					
			(ST + Q + BM + 8B)	(NDR + REQ + CU4)	CPU: Single-threaded implementation	CPU: OpenMP with 2 threads	CPU: OpenMP with 4 threads	CPU: OpenMP with 8 threads
Luxembourg-OSM	2	0.068	**0.078**	0.436	0.334	0.246	0.131	**0.074**
Inf-Roadnet-PA	2	0.187	**1.008**	1.473	1.744	1.244	0.659	**0.345**
Ecology1	2	0.0009	**0.946**	4.845	5.783	4.248	2.172	**1.146**
Roadnet-CA	3	0.1844	**1.873**	2.557	3.393	2.301	1.21	**0.632**
G3-Circuit	2	0.077	**1.441**	1.954	2.541	1.733	0.894	**0.484**
Sc-msdoor	45	0.142	0.697	**0.237**	0.226	0.17	0.094	**0.055**
Sc-ldoor	43	0.144	1.561	**0.587**	0.657	0.462	0.256	**0.144**
soc-youtube	7	0.7274	0.618	**0.096**	0.063	0.056	0.04	**0.028**
Citation-DBLP	56	0.544	1.046	**0.252**	0.159	0.129	0.082	**0.05**
Hollywood-2009	105	0.7346	2.818	**1.041**	0.567	0.394	0.271	**0.191**

4.3 Comparison with a CPU implementation

This work has shown the efficacy of sparsity-aware optimization paths for BFS. In this section, the best performing implementations on Stratix 10 SX 2800 FPGA from this work are compared against an OpenMP-based parallel implementation of BFS from the Rodinia benchmark suite [4], as shown in Table 4.9. The CPU is Intel(R) Xeon(R) Gold, the same as the host CPU for the OpenCL program. The CPU operates at 3.4GHz. For graphs with GI < 0.2, the single-task implementation with local memory optimization offers competitive performance within an order of magnitude of CPU performance. This result is encouraging given that the CPU operates at a frequency approximately 22$\times$ faster than the operating frequency for the single-task implementation. For denser graphs, OpenMP implementation with eight threads considerably outperforms the best-performing FPGA implementation. This result is expected because the denser graphs provide sufficient work that can be effectively parallelized across eight CPU cores.

Chapter 5

Summary

5.1 Conclusion

This work has presented two sets of optimization strategies in OpenCL for BFS, one for near-hypersparse graphs and the other for sparse to moderately denser graphs. This work evaluates the impact of the proposed optimizations on Intel Stratix 10 FPGA. A diverse set of graphs with variations in the number of vertices, the number of edges, and degree distribution is used to evaluate the performance of proposed optimizations. Compared to the reference implementation, this work achieves 5.7×-22.3× speedup for the sparser graphs. The optimization approach for denser graphs with a high average degree achieves 1.1×-2× speedup over the reference implementation.

5.2 Future Work

Graphs present a problem that is dynamic in nature, and static mechanisms cannot address it entirely. This work shows that optimizing sparser and denser graphs requires a different set of optimizations. A possible subject of future study is to extend this work to a framework similar to the auto-tuning framework of Gswitch [22] on GPUs. Furthermore, it is possible to explore the efficacy of a hybrid approach where the choice of the appropriate design, i.e., queue-based single-task vs. NDRange for a given iteration, is made at run-time depending

on the sparsity of the vertices in the active set.

Bibliography

[1] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330077. doi: 10.1145/2664666.2664670. URL https://doi.org/10.1145/2664666.2664670.

[2] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno. CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 228–235, 2014.

[3] Sungmin Bae, Yong Cheol Cho, Sungho Park, Kevin Irick, Yongseok Jin, and Vijaykrishnan Narayanan. An FPGA Implementation of Information Theoretic Visual-Saliency System and Its Optimization. pages 41–48, 05 2011. doi: 10.1109/FCCM.2011.41.

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

[5] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. Pannotia: Understanding Irregular GPGPU Graph Applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 185–195, 2013.

[6] X. Chen, R. Bajaj, Y. Chen, J. He, B. He, W. Wong, and D. Chen. On-The-Fly Parallel Data Shuffling for Graph Processing on OpenCL-Based FPGAs. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 67–73, Sep. 2019. doi: 10.1109/FPL.2019.00020.

[7] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to High-Performance Hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534, 2012.

[8] 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*, FPGA '17, page 217–226, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450343541. doi: 10.1145/3020078.3021739. URL https://doi.org/10.1145/3020078.3021739.

[9] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. XBFS: Exploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, page 121–131, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366700. doi: 10.1145/3307681.3326606. URL https://doi.org/10.1145/3307681.3326606.

[10] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. Spector: An OpenCL FPGA Benchmark Suite. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 141–148, Dec 2016. doi: 10.1109/FPT.2016.7929519.

[11] M. W. Hassan, A. E. Helal, P. M. Athanas, W. Feng, and Y. Y. Hanafy. Exploring FPGA-specific Optimizations for Irregular OpenCL Applications. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2018. doi: 10.1109/RECONFIG.2018.8641699.

[12] T. Huang, G. Dai, Y. Wang, and H. Yang. HyVE: Hybrid Vertex-Edge Memory Hierar-

chy for Energy-Efficient Graph Processing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 973–978, 2018.

[13] *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*. Intel, . URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807516407.html.

[14] *Intel FPGA SDK for OpenCL Pro Edition: Programming Guide*. Intel, . URL https://www.intel.com/content/www/us/en/programmable/documentation/mwh1391807965224.html.

[15] Intel. Intel FPGA SDK for OpenCL, 2019. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf.

[16] *Spector BFS*. Kastner Research Group. URL https://github.com/KastnerRG/spector/tree/master/bfs.

[17] 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. doi: 10.1007/s11265-015-1051-z. URL https://doi.org/10.1007/s11265-015-1051-z.

[18] Jérôme Kunegis and Julia Preusse. Fairness on the Web: Alternatives to the Power Law. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, page 175–184, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312288. doi: 10.1145/2380718.2380741. URL https://doi.org/10.1145/2380718.2380741.

[19] C. Liu, X. Chen, B. He, X. Liao, Y. Wang, and L. Zhang. OBFS: OpenCL Based BFS Optimizations on Software Programmable FPGAs. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 315–318, Dec 2019. doi: 10.1109/ICFPT47387.2019.00056.

[20] H. Liu and H. H. Huang. Enterprise: Breadth-First Graph Traversal on GPUs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[21] 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 *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[22] 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*, PPoPP '19, page 201–213, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295716. URL https://doi.org/10.1145/3293883.3295716.

[23] A. Munshi. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314, 2009.

[24] 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*, FPGA '16, page 111–117, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450338561. doi: 10.1145/2847263.2847337. URL https://doi.org/10.1145/2847263.2847337.

[25] Ryan A. Rossi and Nesreen K. Ahmed. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*, 2015. URL http://networkrepository.com.

[26] Sabine Storandt. Algorithms for Vehicle Navigatio, Ph.D. thesis, University of Stuttgart. 2012.

[27] 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, 2015.

[28] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. SEP-Graph: Finding Shortest Execution Paths for Graph Processing under a Hybrid Framework on GPU. New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362252. doi: 10.1145/3293883.3295733. URL https://doi.org/10.1145/3293883.3295733.

[29] 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, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340922. doi: 10.1145/2851141.2851145. URL https://doi.org/10.1145/2851141.2851145.

[30] Xilinx. The Xilinx SDAccel Development Environment, 2015. URL https://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf.

[31] S. Zhou, R. Kannan, V. K. Prasanna, G. Seetharaman, and Q. Wu. HitGraph: High-throughput Graph Processing Framework on FPGA. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2249–2264, 2019.

[32] 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*, CF '18, page 69–77, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357616. doi: 10.1145/3203217.3203233. URL https://doi.org/10.1145/3203217.3203233.