Exploring FPGA Optimizations in OpenCL for Breadth-First Search on Sparse Graph Datasets

Atharva Gondhalekar  
Department of ECE  
Virginia Tech  
Blacksburg, VA, USA  
athanv@vt.edu

Wu-chun Feng  
Department of CS and ECE  
Virginia Tech  
Blacksburg, VA, USA  
feng@cs.vt.edu

Abstract—Breadth-first search (BFS) is a fundamental building block in many graph-based applications, but it is difficult to optimize due to its irregular memory-access patterns. Prior work, based on hardware description languages and high-level synthesis, address the memory-access bottleneck by using techniques such as edge-centric traversal, data alignment, and compute-unit replication. While these optimizations work well for dense graphs, they are ineffective for sparse graphs due to kernel launch overhead and poor workload distribution across processing elements. Thus, to complement this prior work, we present and evaluate optimizations in OpenCL for BFS on sparse graphs. Specifically, we explore application-specific and architecture-aware optimizations aimed at mitigating the irregular global-memory access bottleneck in sparse graphs. Our kernel design considers factors such as data structure (i.e., queue vs. array), number of memory banks, and kernel launch configuration. Across a diverse set of sparse graphs, our optimizations deliver a 5.7× to 22.3× speedup on Stratix 10 FPGA when compared to a state-of-the-art OpenCL implementation for FPGA.

Index Terms—FPGA, BFS, OpenCL, HLS, sparse graphs, dense graphs, graph traversal, irregular applications.

I. INTRODUCTION

Breadth-first search (BFS) is a graph traversal algorithm that is 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. A level-synchronous BFS can be described using a filter-apply-expand abstraction, as implemented in OpenDwarfs [1] and Gswitch [2]. The filter stage identifies the “vertices in a given level” to process. This set of vertices to be processed in a given level is known as the active set, and the vertices in that level 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. This process continues until the size of the active set reaches zero.

Recent work to accelerate the graph traversal algorithm has spanned a diverse set of fixed [3]–[6] and reconfigurable architectures [7]–[10] at different levels of programming abstraction. With the advent of OpenCL [11], [12], many graph optimization strategies have been explored [1], [8], [13], [14]. Some OpenCL implementations for FPGA even achieve performance that is comparable to the HDL-based implementations [15]. Many of the OpenCL-based solutions identify the irregular memory-access pattern of graph traversal as a major performance bottleneck and propose strategies such as data alignment, SIMD vectorization, and compute unit replication to improve the performance [8], [13]–[15]. Such optimizations perform well for relatively dense graphs with low graph diameter but not for sparse graphs. Optimizing BFS for sparse graphs remains a challenging task due to factors such as kernel launch overhead and poor workload distribution among processing elements. To address these shortcomings, we present optimization strategies to improve the performance of BFS for sparse graphs, resulting in the following contributions.

• A set of optimizations for vertex-centric BFS that targets the FPGA processing of sparse graph datasets, including kernel fusion, architecture-aware optimizations, and other application-specific optimizations.

• A queue-based implementation of BFS in OpenCL for FPGA that incorporates the aforementioned optimizations and delivers a speedup of 5.7× to 22.3× over an OpenCL-based state-of-the-art FPGA implementation.

II. RELATED WORK

A. Graph Processing on GPUs

With the emergence of general-purpose computing on GPUs, many BFS implementations have been proposed. Gunrock [3] and Enterprise [5] are high-performance graph-processing frameworks on NVIDIA GPUs. Rodinia [4] and Pannotta [16] are OpenCL-based benchmark suites with BFS and other graph algorithms. More recent advancements include Gswitch [2] and Sep-graph [17], both of which propose runtime adaptation between multiple strategies to optimize graph-based applications. While GPUs provide significant performance gains, many FPGA-based solutions for graph algorithms and other applications have been presented that achieve comparable performance and significantly higher performance per watt [7], [18]. See below for additional details.
B. Graph Processing on FPGAs

The research in graph processing on FPGAs is extensive. Hitgraph is an HDL-based, edge-centric, graph-processing framework [7]. Hitgraph uses 20× less power than Gunrock on an NVIDIA K40 GPU; it also performs as well as (or better than) the GPU-based Gunrock even though the GPU has 4.8× higher memory bandwidth than the Xilinx Virtex UltraScale+ FPGA used in Hitgraph. Zhou et al. also present an HDL-based, edge-centric BFS [19]. CyGraph [20] is a reconfigurable architecture for parallel graph processing using queues. HyVE [21] implement a hybrid vertex-edge memory hierarchy for energy-efficient graph processing. GraphOps presents a modular hardware library for energy-efficient processing of graph-analytics algorithms [10]. Luo et al. [22] evaluate the irregular memory accesses in a Monte Carlo simulation and explore techniques to hide memory latency. ForeGraph [9] is a graph-processing framework for multi-FPGA systems. Finally, Umuroglu et al. present a hybrid CPU-FPGA implementation [23]. None of the above, however, specifically addresses the issues that arise when processing sparse graphs with BFS, namely kernel launch overhead and poor workload distribution across processing elements.

C. OpenCL-based BFS on FPGA

Several implementations of BFS have been proposed for FPGA. OpenDwarfs was one of the first OpenCL-based implementations for FPGA [1]. It is a vertex-centric and architecture-agnostic BFS kernel that is functionally portable across CPU, GPU, APU, and FPGA. In the BFS implementation from Spector [13], architecture-aware optimizations for FPGA are applied to the OpenDwarfs BFS kernel. These optimizations include compute-unit (CU) replication, loop unrolling, and SIMD work-item execution. Both OpenDwarfs and Spector 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 a 5× improvement over the architecture-agnostic OpenDwarfs kernel on synthetic graphs [14]. Chen et al. propose an edge-centric BFS with multiple processing elements (PEs) as well as a novel data-shuffling technique to handle run-time dependencies caused due to data dispatch across multiple PEs [8]. OBFS [15] makes use of techniques such as graph re-ordering, data alignment, and overlapping the apply stage with other tasks.

Our approach differs from prior work in three ways. First, we avoid expensive global-memory accesses by maintaining the data structures in local memory. Second, we explore a queue-based implementation in addition to an array-based implementation. Third, we merge the three stages (filter-apply-expand) in a single kernel and avoid kernel launch overhead.

III. APPROACH

In this work, we leverage the filter-apply-expand approach and use 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. The active set can be represented either using a queue or an array.

1) Queue-based BFS: In a typical queue-based implementation, vertices are added and removed using push and pop operations, respectively, as shown in Algorithm 1. These push and pop operations cannot be implemented in OpenCL since dynamic memory management is not supported by OpenCL. However, we can mimic the behavior of the queue by using arrays — one array to represent the queues for storing active vertices and the other to represent their neighbor set. Instead of performing push and pop operations, we use 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.

Algorithm 1: BFS implementation using queues

\begin{algorithm}
\begin{algorithmic}
\State \textbf{Input:} graph \( G(V, E) \), source \( S \), vertices \( n \)
\State \textbf{Output:} cost\([n]\]
\State \textbf{Data:} cqueue\([n]\), nqueue\([n]\), visited\([n]\]
\State \textbf{cost}[S] \leftarrow 0
\State \textbf{cqueue}[0] \leftarrow \textbf{s}
\While {\text{sizeof}(\textbf{cqueue}) \neq 0}
\ForEach {vertex \textbf{v} \in \textbf{cqueue}}
\If {\textbf{visited}[\textbf{v}] \neq 0}
\textbf{visited}[\textbf{v}] \leftarrow 0
\textbf{cost}[\textbf{v}] \leftarrow \textbf{cost}[\textbf{v}] + 1;
\textbf{nqueue}.push(v2);
\textbf{cqueue.pop}(\textbf{v})
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

2) Array-based BFS: Active set vertices can be represented using arrays. Typically, arrays with the size equal to the number of vertices are initialized. A vertex \( v \) is active if the value at the index \( v \) of array is 1. It is possible to use a bitmap instead of array. That is, one element of the integer array is 4 bytes long and therefore can hold information on up to 32 vertices. The active set can be filtered from the bitmap by performing bit-level shift and logical XOR, AND, and NOT operations. Hassan et al. [14], [15] make use of a bitmap whereas Krommydas et al. [1] and Gautier et al. [13] use arrays. Algorithm 2 shows array-based graph traversal.

IV. OPTIMIZATIONS

Here we describe the optimizations used to accelerate BFS. We categorize the optimizations into three groups: (1) OpenCL framework-specific optimizations, (2) architecture-aware optimizations, and (3) application-specific optimizations.

A. OpenCL Framework-specific Optimizations

OpenCL allows programmers to invoke the kernel in one of the two configurations: NDRange and single task. With NDRange, kernel execution occurs using a group of work items that execute the kernel in a pipelined manner. In a single-task kernel launch, only a single work item is used; this configuration allows pipelined execution within the kernel loops.
The single-task configuration allows us to implement the set as opposed to the pipelining of work items from NDRange. The single-task configuration allows us to implement the set of optimizations discussed below.

1) **Kernel fusion:** While the NDRange variants require synchronization during the filter and expand stages, a single-task variant does not require synchronization between the two stages due to the serial execution of a single work item. This allows all the BFS stages to be merged into one kernel. Thus, kernel fusion reduces the number of synchronization calls made using clFinish between the BFS stages.

2) **Elimination of host-based synchronization:** To avoid host-based synchronization altogether, we insert an outermost while loop in the merged kernel. The loop terminates when the number of active vertices is zero.

### B. Architecture-aware Optimizations

OpenCL facilitates a “write once, run anywhere” programming model. However, extracting optimal performance from an OpenCL kernel when running on a given hardware architecture necessitates architecture-aware optimizations. Below we describe our architecture-aware optimizations to achieve high performance on an Intel Stratix 10 FPGA.

1) **Local queue:** In the array-based version, active vertices are found by searching all the elements of an array. In the queue-based version, we iterate over the size of the queue. While the queue eliminates the need for redundant comparisons, it still needs expensive irregular reads and writes to global memory. We mitigate the irregular global accesses by initializing the queue in FPGA local memory (BRAM). Though this optimization limits the graph sizes that can be processed, we show in §IV-C how to alleviate this limitation. We also realize a local array-based BFS from Algorithm 2 and compare the performance of the two versions in §V.

#### C. Application-specific Optimizations

The architecture-agnostic kernels in OpenDwarfs and their optimized FPGA variants in Spector provide unoptimized and optimized baselines for FPGA performance, respectively. However, there still remains room for application-specific optimizations to further improve performance, as noted below.

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 its neighbor, we use a local variable to keep track of the current level. We assign this level to the neighbors of active vertices. This optimization improves performance in two ways. First, it avoids the expensive global read of the dependent variable in the loop that stalls the pipeline in the filter stage. As a result, our initiation interval (i.e., II) for the loop in the filter stage takes only one (1) clock cycle, as shown in the Table I. Second, it avoids redundant cost updates that take place in Algorithm 2 at line 10 when two vertices from the active set have shared neighbors.

2) **Elimination of duplicate entries:** The array visited in Algorithm 1 serves the purpose of eliminating the introduction of duplicate entries in the queue; but using an array limits the size of the queue as local memory is limited. By replacing the array with a bitmap, we allocate more space for the queue and, in turn, can process graphs with up to \(2^{21}\) vertices (or 2,097,152 vertices) in our optimized FPGA implementation.

![Algorithm 2: BFS implementation using arrays](image)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>II (# clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-task OpenDwarfs with array</td>
<td>237</td>
</tr>
<tr>
<td>Single-task OpenDwarfs with bitmap</td>
<td>225</td>
</tr>
<tr>
<td>This paper (Using local copy of level)</td>
<td>1</td>
</tr>
</tbody>
</table>

1The initiation interval (II, for short) is the number of clock cycles that the pipeline must stall before it can process the next loop iteration.
TABLE II: Sparse Graph Datasets

<table>
<thead>
<tr>
<th>Graph</th>
<th>Description</th>
<th># Vertices</th>
<th># Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luxembourg OSM</td>
<td>Open street map</td>
<td>114.8K</td>
<td>159.7K</td>
</tr>
<tr>
<td>roadNet-CA</td>
<td>Road network</td>
<td>2.0M</td>
<td>2.8M</td>
</tr>
<tr>
<td>roadNet-PA</td>
<td>Road network</td>
<td>1.1M</td>
<td>1.5M</td>
</tr>
<tr>
<td>Belgium OSM</td>
<td>Open street map</td>
<td>1.4M</td>
<td>1.5M</td>
</tr>
<tr>
<td>G3-circuit</td>
<td>Circuit simulation problem</td>
<td>1.6M</td>
<td>3.0M</td>
</tr>
<tr>
<td>Ecology1</td>
<td>Landscape ecology problem</td>
<td>1.0M</td>
<td>1.9M</td>
</tr>
</tbody>
</table>

TABLE III: Resource Usage Summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Data Structures for Active Set</th>
<th>Area Utilization</th>
<th>Frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spector</td>
<td>Global arrays</td>
<td>90/193 (16%)</td>
<td>1912 (16%)</td>
</tr>
<tr>
<td>This Paper</td>
<td>Local arrays</td>
<td>23147/3 (12%)</td>
<td>108/108</td>
</tr>
<tr>
<td></td>
<td>Local queues with bitmap</td>
<td>226834 (12%)</td>
<td>108/108</td>
</tr>
<tr>
<td></td>
<td>Local queues with memory banks</td>
<td>226834 (12%)</td>
<td>108/108</td>
</tr>
</tbody>
</table>

V. PERFORMANCE EVALUATION

A. Experimental Setup

We perform our experiments on an Intel® Stratix 10 SX 2800 FPGA with an Intel® Xeon® Gold 6128 CPU as the OpenCL host. To evaluate the efficacy of our approach (vs. the state-of-the-art Spector BFS), we use a workload of undirected sparse graphs from “The Network Data Repository with Interactive Graph Analytics and Visualization” [26], as shown in Table II. We draw the reference implementation of BFS from [27] with commit SHA b536909.

B. Analysis of Resource Utilization

Table III shows the area utilization and frequency of Spector BFS and our BFS implementations on the Stratix 10 FPGA. Kernels in the global array-based Spector and our local array-based BFS implementation are scheduled at higher frequencies than the kernels in the queue-based implementations. In our local memory-based implementations, the RAM usage is considerably higher than in Spector. RAM usage is 77% for the queue-based kernels compared to 16% in the Spector. There are 4% more ALUTs and FFs required in Spector compared to the queue-based version. This is expected as Spector makes use of optimizations such as SIMD work-item execution.

C. Performance Improvement

Fig. 1 shows the normalized speedup of our BFS over Spector’s BFS [13]. The speedup comparisons are with respect to the optimized BFS implementation in Spector. While Spector’s BFS outperforms our local array-based kernel by 3.18× on the Ecology1 graph and 2.35× on the Luxembourg-osm graph, our queue-based implementations significantly outperform Spector’s. For our queue-based kernel with a local array, we observe 12.80× and 13.15× speedup with the Luxembourg-osm and Ecology1 graphs, respectively. Using a bitmap, instead of an array, in the same kernel further improves performance by a factor of 1.29.

1) Impact of banking: For the graphs in Fig. 1, the queue-based kernel with bitmap outperforms Spector by a factor of 16.9. Using two memory banks in the same kernel causes an average 1.3× further increase in the performance. We do not observe any further increase in the performance by increasing the number of banks to 4 and 8. The operating frequency for the queue-based version with two banks is 148 MHz (vs. 108 MHz without banking). For all the graphs in Table II, we observe a similar trend where the queue-based implementation with bitmap and two banks shows the most improvement.

2) Impact of sparsity: Table IV shows the speedup for our best performing implementation (queue + bitmap + 2 memory banks) over Spector for the sparse graphs. For the same implementation, we report the number of edges traversed per second in Table IV. Graphs with a much higher number of BFS iterations benefit the most from our optimizations. Sparse graphs process a small number of vertices in each iteration. A queue-based traversal is preferable over an array-based traversal as it avoids the need to iterate over the entire array to find active vertices. Our set of optimizations is applicable to sparse datasets, and we show 5.7× to 22.3× improvement over the reference implementation for such graphs.

VI. CONCLUSION

This paper presents OpenCL optimizations for BFS on sparse graphs. Irregular global memory access bottleneck for sparse graphs is mitigated by using architecture-aware and application-specific optimizations. We evaluate the impact of the proposed optimizations on an Intel Stratix 10 SX 2800 FPGA. Compared to the reference implementation, we achieve 5.7× to 22.3× speedup for sparse graphs. This work is meant to serve as a complement to the state-of-the-art parallel BFS designs that improve the performance for dense graphs using replication of kernel logic. As a subject of future study, we intend to explore the efficacy of a hybrid approach where the choice of the appropriate design, i.e., queue-based single PE vs. multiple PEs for a given iteration is made at run-time depending on the sparsity of the vertices in the active set.
REFERENCES


