Strategetic Optimization for BFS Acceleration on FPGAs

Cheng Liu

July 17, 2017

1 Introduction

Graph is a common data representation in a broad domains of applications such as social networks, bioinformatics, metabolic networks and computer networks. As the graph grows in scale in the big data era, graph processing becomes increasingly costly and accelerating the widely used building components such as Breadth-first search (BFS) is of vital importance for high-performance graph processing.

BFS is notoriously difficult for accelerating on parallel computing architectures because of the irregular memory access and the low computation-to-memory ratio. These characteristics essentially make BFS a memory bandwidth bounded task and challenging for accelerating. To approach this challenge, a number of research have been performed targeting on various platforms including multicore processors, GPUs [4], FPGAs [1–3, 5, 6], ASICs [**?**] and distributed systems over the past few years.

With the increased interest on energy efficient computing systems, FPGAs gains increasing popularity and is rapidly moving forward to main-stream computing system and gets adopted in more data centers. Many previous work also demonstrated that building BFS accelerators on FPGAs can provide competitive performance over multi-core CPUs on top of the energy efficiency. In this work, we thus try to explore BFS accelerating on FPGAs and we argue that FPGA is suitable for BFS acceleration mainly for two reasons. First of all, FPGA allows specific memory access infrastructures to be built to fit for various memory access patterns including customized caching, streaming and random memory access. Secondly, BFS is memory bound and suffers relatively long external memory access latency. As a result, it turns to be a good fit to FPGAs with ample parallelism but relatively slow clock speed.

BFS will be briefly explained here. Some concepts such as frontier will be introduced here as well.

Previous BFS acceleration on FPGA work mainly focused on developing parallel hardware architectures to improve the resulting performance. However, the inherent memory bound characteristic of BFS dramatically limits the benefits brought by more parallel processing logic. In this work, we focus on strategetic optimizations which aims to reduce the amount of memory access and improve the memory bandwidth utilization to directly alleviate the bottleneck of the BFS

accelerators on FPGAs and thus enhance the resulting performance. Four BFS optimization strategies are developed to improve the memory bandwidth utilization of the BFS accelerator.

- The BFS is divided into two phases. In the first phase, the frontier is inspected from the vertex level information with complete sequential memory access. In the second phase, it traverses the frontier just as the conventional level based BFS. Compared to the conventional level based BFS, the frontier is no longer a shared writing queue among the paralell traverse processing logic and it ensures no duplicated vertices in the frontier queue. This strategy thus avoids either complex unique processing of the frontier or redudant frontier vertices processing. Although additional inspection stage is needed, it brings just sequential memory access and can be very efficient. Essentially this strategy improves the memory bandwidth utilization.

- The frontier vertices are classified into four groups based on its vertex degree and each group of the frontier vertices will be processed with specially customized logic. Basically frontier vertices with higher degree will be handled by processing elements with burst memory access support, while low-degree vertices will be passed to process elements with just simple random memory access. This strategy helps to improve the memory bandwidth utilization as well as balancing the processing time of different vertices.

- A typical top-down based BFS is efficient when the frontier size is small. However, the vertices in the same frontier may have the same neighbors. When they are processed in parallel, a large amount of redundant traverse will be incurred at BFS level with larger frontier size. This problem gets worse for the graphs with small-world property whose BFS frontier size scales with power law. A bottom-up BFS can be more efficient for BFS iterations with larger frontier size. Thus a hybrid top-down and bottom-up BFS is implemented. With a relatively BFS mode switching metric, this hybrid BFS can reduce the amount of memory access significantly.

- Real-world graphs have small world property and a small portion of the vertices have high degree. These high-degree vertices may be referenced many times during the traverse. With this observation, the high-degree

vertex traversal status is cached on FPGA. Whenever a status of vertex needs to referenced during the traverse, the processing elements will check the cache first. High degree vertex status cached on-chip will thus help to avoid the repeated memory access and improves the memory bandwidth utilization eventually.

The organization of the paper goes as follows. Section 2 presents the background of the BFS algorithms and related work on BFS acceleration. Section 3 gives the overview of the proposed BFS accelerator. Section 4 details the optimization strategies and the corresponding implementations. Section 5 shows the experimental results and analysis. Section 6 concludes this work.

2 Background and Related Work

This section will firstly introduce the background of the BFS algorithms and then review previous BFS acceleration work.

2.1 Background

BFS is a widely used graph traversal algorithm in many applications. It traverses the graph by processing all vertices with the same distance from the source vertex iteratively. The set of vertices which have the same distance from the source is defined as frontier. The frontier that is under analysis in the BFS iteration is named as current frontier while the frontier that is inspected from current frontier is called next frontier. By inspecting only the frontier, BFS can be implemented efficiently and thus the frontier concept is utilized in many BFS implementations.

BFS algorithm can be formalized as follows. Assume G is a graph with vertex set V and edge set E, BFS finds a shortest path from a source vertex $v_s \in V$ to all the other vertices in the graph G. For each vertex $v \in V$, BFS will output a level value l, indicating its distance from v_s (v can be accessed from v_s by traveling through $l-1$ edges).

Algorithm 1 presents a level synchronized BFS algorithm. It has been utilized in many previous BFS accelerators. More explanation will be added here.

Algorithm 1 Level Synchronized BFS Algorithm

```
1: procedure BFS
2:     level[v_k] = −1 where v_k ∈ V
3:     level[v_s] = 1
4:     current_frontier ← v_s
5:     current_level = 1
6:     while current_frontier not empty do
7:         for v ∈ current_frontier do
8:             S = n ∈ V|(v, n) ∈ E
9:             for n ∈ S do
10:                if level[n] == −1 then
11:                    level[n] ← current_level + 1
12:                    next_frontier ← n
13:        current_level = current_level + 1
14:        Swap current_frontier with next_frontier
```

2.2 Related Work

to be added.

3 BFS Optimization Strategies

The modified BFS algoirithm is presented in 2. BFS optimization strategies wll be detailed in this section.

4 Proposed BFS Accelerator

The BFS accelerator overview and hardware modules developed for the corresponding BFS optimization strategies will be detailed here.

Algorithm 2 Modified BFS Algorithm

```
1: procedure BFS
2:     level[v_k] = −1 where v_k ∈ V
3:     level[v_s] = 0
4:     current_level = 0
5:     frontier ← v_s
6:     while frontier not empty do
7:         for v ∈ V do
8:             if level[v] == current_level then
9:                 frontier ← v
10:        for v ∈ frontier do
11:            S = n ∈ V|(v, n) ∈ E
12:            for n ∈ S do
13:                if level[n] == −1 then
14:                    level[n] ← current_level + 1
15:        current_level = current_level + 1
```

5 Experiments

5.1 Experiment Setup

5.2 Graph Benchmark

5.3 Results

- performance improvements and resource overhead of the optimization strategies
- Reduced communication with the optimized strategies
- Runtime distribution analysis

6 Conclusion

to be added.

References

[1] Osama G Attia, Tyler Johnson, Kevin Townsend, Philip Jones, and Joseph Zambreno. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 228–235. IEEE, 2014.

[2] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 8–15. IEEE, 2012.

[3] Nina Engelhardt and Hayden Kwok-Hay So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. IEEE, 2016.

[4] Hang Liu and H Howie Huang. Enterprise: Breadth-first Graph Traversal on GPUs. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 68:1—-68:12, 2015.

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

[6] Yaman Umuroglu, Donn Morrison, and Magnus Jahre. Hybrid breadth-first search on a single-chip fpga-cpu heterogeneous platform. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015.