Exploring BFS Optimization Using High Level Synthesis Tool

ABSTRACT

Recent years have seen a lot of successful demonstrations of using FPGAs for data intensive applications such as neural network, data compression and graph processing. The increasing popularity of FPGAs motivates the cloud vendors to have FPGAs integrated in the cloud and provide FPGAs as a service. At the same time, to both take advantage of the powerful FPGAs equipped in the cloud and improve the design productivity of the designers, the vendors mostly recommend high level synthesis tools or integrated design environments to program FPGAs using C/C++ or OepnCL. However, exploring large data intensive applications using high level design tools on the FPGAs for higher performance remains challenging when compared to those developed on top of conventional computing infrastructures i.e. CPU and GPU.

In this work, we take breadth first search (BFS) on large skewed graphs as an example exploring data intensive application acceleration on the FPGAs using Xilinx SDAccel which is the standard design environment for Xilinx PCIE based FPGAs. BFS is one of the most important building blocks of graph processing and it is notoriously difficult for parallel computing architectures due to the skewed data structure and irregular memory access. With the high level design tools, we can rapidly identify the performance bottleneck and implement corresponding optimization strategies such as pipeling and caching for the BFS. According to the experiments on real-world skewed graph, the optimized design achieves up to 20X performance speedup compared to the baseline design. Although the performance remains xxx slower than the optimized multi-core implementations, the energy efficiency is xxx times higher.

KEYWORDS

BFS, FPGA Cloud, High Level Synthesis, Pipelining, Caching

ACM Reference Format:

. 2017. Exploring BFS Optimization Using High Level Synthesis Tool. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

FPGAs have been increasingly utilized for accelerating data intensive applications such as neural network, data compression and graph processing and gain increasing popularity over the years. This encourages more and more cloud vendors to integrate FPGAs in the cloud and provide FPGAs as a service. At the same time, high level design tools and integrated environments are typically provided to make the FPGAs accessible to high-level application designers and improve the design productivity of the designers as well. However, optimizing the high level design for better performance on cloud FPGAs remains challenging. To that end, we take BFS as an example exploring the high level data intensive application optimization on the cloud FPGA using state-of-art high level design tools.

Breadth-first search (BFS) is the basic building component of graph processing and is thus of vital importance to high-performance graph processing. Nevertheless, it is notoriously difficult for accelerating on parallel computing architectures because of the irregular memory access and the low computation-to-memory ratio. At the same time, BFS on large graphs also involves tremendous parallelisms which indicates great potential for acceleration. With both the challenge and the parallelization potential, BFS has attracted a number of researchers exploring the acceleration on FPGAs [1–5].

Previous work have shown that BFS accelerators can provide significant performance speedup with superior energy efficiency on FPGAs when given comparable memory bandwidth. However, these work mostly optimize BFS with dedicated register transfer level (RTL) circuit design. The RTL based design does save resource consumption and is beneficial to the performance, but it usually takes long time for development, upgrade and maintenance. While FPGAs in the cloud may have parts of different generations or types, RTL based design thus suffers considerable portability cost when applied to the cloud. Worse still, computing tasks in the cloud may need to be migrated for the sake of performance and power consumption, RTL based design can also be a barrier for the task migration and scheduling in the cloud.

In this work, we particularly focus on BFS optimization using high level design tools targeting the cloud FPGAs. As the BFS is known to be memory bandwidth bound, we mainly centers the memory access optimization using the high level design tools. First of all, we convert the baseline BFS algorithm to a stream processing such that intermediate data can be handled on FPGA while sequential memory access can be easily identified and optimized by the high level design tools. Secondly, we add customized cache to avoid redundant random memory access. In particular, we also take advantage of the power-law distribution of the vertex degree and have the high degree vertex related data cached with higher priority. Finally, we further explore the data path duplication strategy to make full use of the memory bandwidth. According to the experiments on real-world graphs, the optimized high level BFS accelerator obtains xxx times performance speedup when compared to the baseline design. Despite the performance penalty caused by the high level design, the power efficiency is still xxx higher compared to the CPU system.

This paper is organized as follows. In section 2, we briefly introduces the background of BFS and FPGA cloud and then present memory access analysis of BFS to motivate the optimization proposed in this work. In section 3, we present the overview of the BFS accelerator design in a typical cloud FPGA environment. In section

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

4 and 5, we detail the pipeline design and cache strategy of the BFS accelerator using the high level design tools. In section 6, we present comprehensive experiments of the BFS accelerator. Finally, we briefly reviewed previous BFS acceleration work in Section 7 and concludes the paper in Section 8.

2 BACKGROUND AND MOTIVATION

In this section, we will briefly introduce the FPGA cloud and the state-of-art high level design environments for development. Then we will evaluate the memory access efficiency for different memory access patterns and present insight analysis of the BFS performance bottleneck on top of the memory efficiency analysis.

2.1 FPGA Cloud

In the past few years, FPGAs are becoming pervasive in data centers for accelerating various data intensive applications such as software defined network, nueral network, and database. With the increasing demand of FPGAs in the data center, more and more cloud vendors start to provide FPGA-as-a-service. However, the complexity of programming FPGAs using low level hardware description language has been a major obstacle that hinders FPGA cloud from widespread adoption.

To address this problem, the cloud vendors thus start to offer high level programming options such as C/C++ and OpenCL to program the FPGAs in the cloud. Xilinx Nimbix already have OpenCL and high level synthesis supported. Amazon is also moving forward to this direction. Although the high level design tools are not as mature as the conventional HDL design flow, it is clear that they are critical to make the FPGAs accessible to more designers and spread FPGAs to more applications and will become the major design methods for FPGA programming in FPGA cloud in future.

2.2 BFS Algorithm

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.

A widely used frontier based BFS algorithm implementation is named as level synchronized BFS. The details of the algorithm are presented in Algorithm 1. The basic idea is to traverse the frontier vertices and inspect the neighbors of the current frontier vertices to obtain the frontiers in next BFS iteration. Then the algorithm can start a new iteration with a simple switch of current frontier queue and next frontier queu. The algorithm ends when the frontier queue is empty.

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
```

Table 1: Real-World Graphs

Name	# of vertex	# of edge	Type
Youtube	1157828	2987624	Undirectional
Live Journal	4847571	68993773	Directional
Pokec	1632804	30622564	Directional
rmat-19-32	524288	16777216	Directional
rmat-21-128	2097152	268435456	Directional

2.3 Motivation

BFS on large skewed graph is known to be a memory bandwidth bound task. Therefore understanding the BFS memory access pattern and providing suitable memory access optimization strategies is critical to make best use of the memory bandwidth and achieve higher performance of BFS. To that end, we analyzed the memory access pattern of a baseline BFS on a set of real-world graphs selected from SNAP benchmark. Details of graphs are listed in Table 1.

Figure **??** presents the memory access pattern of a sequential bfs assuming the graph is stored as compressed sparse row (CSR) format. From this figure, it can be found that bfs memory access pattern is rather complex. It involves both considerable sequential memory access with various length as well as random memory access. For the sake of performance, both sequential memory access and random memory access needs to be optimized in the bfs accelerator.

For sequential memory access, the optimization in high level design tools is therefore to ensure the tools to generate streaming or burst memory operations. For random memory access, it is relatively challenging. We notice that most of the random memory access in the BFS algorithm is from checking frontier neighbor status as indicated in previous BFS algorithm. As the vertex status is stored sequentially, we thus analyzed the redundancy of the frontier neighbor vertices to get insight of the random access. Figure 2 shows the amount of frontier neighbor vertices that need to be accessed in each BFS iteration. In some of the BFS iterations, the amount is even large than the total amount of the vertices in the graph. It indicates that there are many redundant vertices in it.

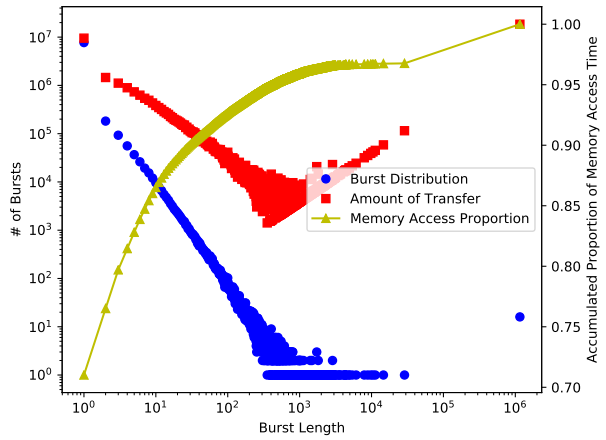

Figure 1: Burst Memory Operation Distribution in BFS on Youtube Graph. There are large amount of random memory access and short sequential memory access. There are few large sequential memory operations, but the total amount of memory operations are not negligible.

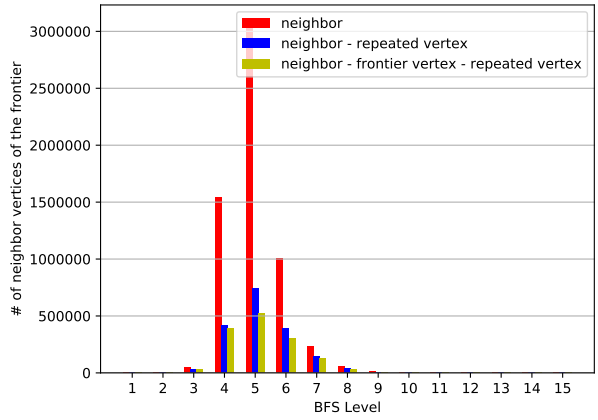

Figure 2: Repeated Neighbors of Frontier Vertices in BFS

When we get rid of the redundant vertices, the amount of vertices drops up to 70% as shown in the figure. In addition, we also analyzed the vertices that are visited in previous BFS iteration. It turns out that another big proportion of the neighbor vertices can be ignored and there is no need to read its status in the following part of the BFS algorithm. With this observation, we thus try to buffer the traversed frontier vertices and check the buffer before fetching from external memory to get rid of the redundant memory access in the BFS accelerator.

The real-world graphs usually have large amount of vertices and edges while the FPGA on chip memory is far from enough to buffering even the redundant vertices and their information. Hereby, we can only have a sub set of the vertices to reside in on-chip memory. Apparently, high degree vertices are more likely to be referenced

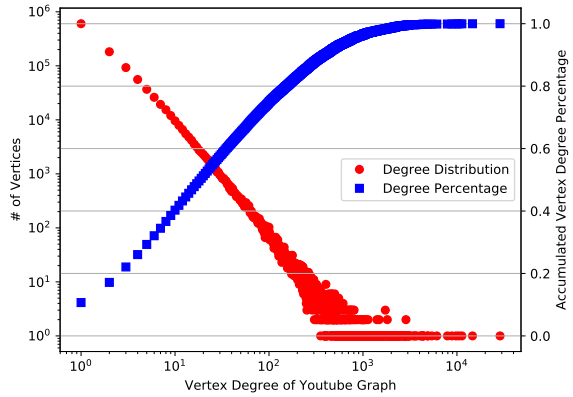

Figure 3: Vertex Degree Distribution. A small amount of high degree vertices take up a large proportion of the connections in the graph

in the BFS iterations and are the major source of the frontier neighbor redundancy. With limited on chip buffer, we thus try to buffer high degree vertices in a higher priority. To further investigate the potential of buffering the high degree vertices, we analyzed the vertex degree distribution of Youtube graph. The vertex degree distribution is shown in 3. It can be found that a small amount of high degree vertices may have a large proportion of the connections of the graph. It indicates that buffering a small amount of high degree vertices will help remove the redundant frontier neighbors and thus making best use of the limited on-chip memory.

3 BFS ACCELERATOR OVERVIEW

Our BFS accelerator specially targets small-world in-memory graphs on a PCIE based FPGA card which is a classic setup in the FPGA cloud. It has the graph stored with the CSR format on the FPGA DDR for the traverse. It gets BFS requests from host CPU. When the BFS is done, it transfers the vertex status to host memory. In addition, the BFS accelerator follows the BSP model as most of the previous parallel BFS implementations. Basically the algorithm must be synchronized before moving forward to the next iteration.

The BFS algorithm is critical to the BFS accelerator and thus we explored the existing level synchronous BFS algorithm in detail. We notice that the level synchronous BFS algorithm may have redundant vertices pushed to the frontier queue in a prallel architecture especially when the frontier grows larger. The main reason roots in the large amount of overlapped neighbor vertices among the frontier vertices as mentioned in previous section. When the frontier neighbors are inspected in parallel for faster BFS traverse, these overalpped vertices may be considered as frontiers independently and inserted into the next frontier queue. As it is rather difficult to remove the redundant vertices from the frontier, the redundant vertices in BFS frontier will soon lead to large amount of redundant traverses though it will not cause any mistakes.

To address this problem, we opt to inspect the frontier from the vertex status in each BFS iteration to completely cut down the propogation of the repeated frontier vertices as proposed in prior GPU based BFS acceleration. The modified algorithm is detailed

in 2. Instead of starting from the frontier analysis, it starts with a complete vertex status analysis and inspects the frontier in each BFS iteration. Although it seems the additional frontier inspection stage brings more memory access, the inspection processing are complete sequential memory access and can be done efficiently. It is still worth for the overhead when compared to the cost caused by the redundant frotnier vertices.

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.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
```

4 BFS PIPELINING

As motivated in previous section, there are considerable memory access in the BFS algorithm. In order to ensure the sequential memory access processed efficiently, these memory access should be synthesized as burst memory operation or streaming operation. However, a baseline BFS algorithm on top of CSR data is a complex nested loop which is difficult for the high level design tools to infer the sequential memory access with basic optimization pragama. To address this problem, we transform the algorithm into a series of dependent fine-grained sub functions and the sequential memory access in each function can be explicitly expressed as streaming memory access. These dependent functions can further form a high-level pipeline which fits well with the data flow optimiation in Xilinx SDAccel.

The streamed BFS algorithm is presented in Figure **??**. To convert all the sequential memory access to stream operations using Xilinx SDAccel, the BFS algorithm is divided into six sub functions which are also considered as pipeline stages when optimized using the data flow model provided by Xilinx SDAccel. Details of each sub function is presented in **??**.

The last two stages i.e. S5 and S6 which don't include clear sequential memory access can be merged as one stage with slight performance degreadation with more memory reading ports. This design trade-off will be discussed in the following part of the paper.

Here is a theorem:

Theorem 4.1. *Let f be continuous on $[a, b]$. If G is an antiderivative for f on $[a, b]$, then*

$$\int_a^b f(t)\,dt = G(b) - G(a).$$

Algorithm 3 Streamed BFS Algorithm

```
1:  procedure BFS
2:      frontier_size ← 1
3:      level ← 0
4:      while (frontier_size > 0) do
5:          stage1(depth, depth_stream)
6:          stage2(depth_stream, frontier_stream, level, frontier_size)
7:          stage3(frontier_stream, CSR.RPA, RPA_stream)
8:          stage4(RPA_stream, CSR.CIA, CIA_stream)
9:          stage5(CIA_stream, depth, next_frontier_stream)
10:         stage6(depth, next_frontier_stream, level)
11:         level ← level + 1
12:
13: procedure STAGE1(depth, depth_stream)
14:     for v ∈ V do
15:         depth_stream << depth[v]
16: procedure STAGE2(depth_stream, frontier_stream, level, frontier_size)
17:     frontier_size = 0
18:     for v ∈ V do
19:         d[v] ← depth_stream.read()
20:         if (d[v] == level) then
21:             frontier_stream << v
22:             frontier_size + +
23: procedure STAGE3(frontier_stream, CSR.RPA, RPA_stream)
24:     while (!frontier_stream.empty()) do
25:         v ← frontier_stream.read()
26:         RPA_stream << [CSR.RPA[v], CSR.RPA[v + 1]]
27: procedure STAGE4(RPA_stream, CSR.CIA, CIA_stream)
28:     while (!RPA_stream.empty()) do
29:         [begin, end] ← RPA_stream.read()
30:         for v ∈ CSR.CIA(begin, end) do
31:             CIA_stream << v
32: procedure STAGE5(CIA_stream, depth, next_frontier_stream)
33:     while (!CIA_stream.empty()) do
34:         v ← CIA_stream.read()
35:         if (depth[v] == −1) then
36:             next_frontier_stream << v
37: procedure STAGE6(depth, next_frontier_stream, level)
38:     while (!next_frontier_stream.empty()) do
39:         v ← next_frontier_stream.read()
40:         depth[v] ← level + 1
```

Here is a definition:

Definition 4.2. If z is irrational, then by e^z we mean the unique number that has logarithm z:

$$\log e^z = z.$$

The pre-defined theorem-like constructs are **theorem**, **conjecture**, **proposition**, **lemma** and **corollary**. The pre-defined definition-like constructs are **example** and **definition**. You can add your own constructs using the *amsthm* interface [**?**]. The styles used in the `\theoremstyle` command are **acmplain** and **acmdefinition**.

Another construct is **proof**, for example,

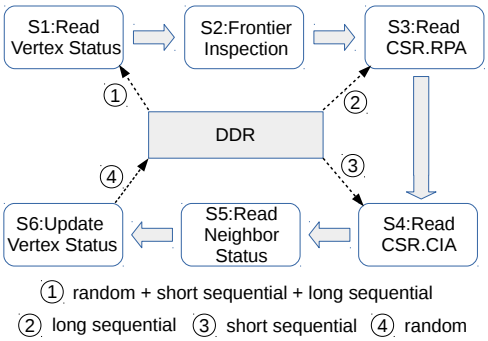

Figure 4: Streamed BFS Algorithm

Proof. Suppose on the contrary there exists a real number L such that

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = L.$$

Then

$$l = \lim_{x\to c} f(x) = \lim_{x\to c}\left[gx \cdot \frac{f(x)}{g(x)}\right] = \lim_{x\to c} g(x) \cdot \lim_{x\to c} \frac{f(x)}{g(x)} = 0 \cdot L = 0,$$

which contradicts our assumption that $l \neq 0$. □

5 BFS CACHING

6 EXPERIMENTS

6.1 Experiment Setup

6.2 Performance Comparison

6.3 Pipelining Analysis

6.4 Caching Analysis

7 RELATED WORK

8 CONCLUSIONS

This paragraph will end the body of this sample document. Remember that you might still have Acknowledgments or Appendices; brief samples of these follow. There is still the Bibliography to deal with; and we will make a disclaimer about that here: with the exception of the reference to the LaTeX book, the citations in this paper are to articles which have nothing to do with the present subject and are used as examples only. this equation: $\lim_{n\to\infty} x = 0$, set here in in-line math style, looks slightly different when set in display style. (See next section).

A ACKNOWLEDGEMENT

ACKNOWLEDGMENTS

The authors would like to thank Dr. Yuhua Li for providing the matlab code of the *BEPS* method.

REFERENCES

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

[2] Brahim Betkaoui, Yu Wang, David B Thomas, and Wayne Luk. 2012. 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*. IEEE, 8–15.

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

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

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