BFS Accelerator

Cheng Liu

April 4, 2017

1 Motivation

In the era of "big data", there has been growing interest in developing graph analytics applications to gain new insights and solutions. While Breadth-First Search (BFS) algorithm is the foundation for many graph processing applications and analytics workloads, many BFS acceleration designs have been proposed targeting multi-core processors, distributed systems, GPUs and FPGAs over the years as listed in Table 1. FPGA which allows intensive application-specific customization has been well-known to be energy efficient compared to general purposed computing devices and is usually able to provide unique performance-energy trade-offs. However, the best performance of BFS accelerator reported in literature is only around 1 to 2 billion traverses per second and is far behind that on the counterpart computing platforms which goes up to 120 billion traverse per second in spite of the memory bandwidth gap.

Table 1: Brief BFS Acceleration Review

Name	Year	Platform	Memory Bandwidth	Performance
Enterprise [6]	2015	GPU	300GB/s 600GB/s	76 GTEPS 122 GTEPS
ASAP12 [2]	2012	FPGA	80GB/s	1.6 GTEPS
IPDPSW14 [1]	2014	FPGA	80GB/s	1.9 GTEPS
FPL15 [8]	2015	FPGA	3.2GB/s	0.172 GTEPS
GraphOps [7]	2016	FPGA	38GB/s	0.8 GTEPS ?
Graphicionado [4]	2016	ASIC	68GB/s	–?
GraVF [3]	2016	FPGA	– ?	3.8GTEPS

When comparing the BFS accelerators on FPGA and GPU, we observe that existing FPGA accelerators mainly fall short on two aspects. First of all, the BFS accelerators on FPGAs have lower memory utilization which is critical to the resulting BFS performance because BFS is a memory-bound computing task due to the irregular memory access. Although the memory utilization may be influenced by many different factors such as the memory hierarchy and processing power difference, one of the primary factors is the amount of memory access needed for the same BFS task. The BFS accelerators on FPGA have few efficient BFS algorithmic strategies supported which are considered to be effective for alleviating the memory bandwidth bottleneck. For example, the adaptive top-down

and bottom-up BFS strategy infers the BFS frontier from the visited vertices at the beginning of BFS while doing so from the unvisited vertices when the BFS frontier gets large. Basically, this strategy helps to reduce the amount of memory access while detecting the BFS frontier. Another example is the high degree vertex caching. Vertices with high degree will be cached locally, which can be used to avoid accessing main memory during the traverse and thus also helps alleviate memory bandwidth pressure. With appropriately applying the algorithmic BFS strategies, the memory utilization of the BFS accelerators can be enhanced and thus the performance can be improved eventually.

Secondly, the practical graphs are typically scale-free and the vertex degree varies in a large range. As a result, the exploration of different type of vertices may require diverse exploration time. When they are assigned to the processing engines of BFS, the parallel processing engines allocated with low-degree vertices may be under utilized. In addition, inspecting low-degree vertices requires scattered short read over the CSR data stored in main memory while inspecting high-degree vertices mostly has continuous burst memory access. To approach this problem, unlike previous work which usually have homogeneous processing engines implemented on FPGA, we may classify the vertices in the frontier based on the degree of the vertices and allocate them to specifically customized processing engines accordingly. Meanwhile, with appropriate resource allocation, we may statically (or dynamically) adjusting the number of processing engines for different vertices such that the utilization of the processing engines can be balanced.

2 Proposed BFS Accelerator

In this section, the proposed BFS accelerator will be illustrated. It follows the classical BSP model. Basically it will traverse the vertices iteratively from the starting vertex. It will not proceed to the next iteration before all the active vertices (frontier) are explored in current iteration.

2.1 Background

Before illustrating the proposed BFS accelerator, we will explain the basic concept of BFS with a basic BFS implementation shown in List 1. Suppose $depth[v]$

represents the depth of vertex v in a graph G. s stands for the starting vertex of the Graph G. Then the sequential BFS algorithm is described as follows.

Listing 1: Sequential BFS Algorithm

```
// top-down BFS algorithm
depth[s] = 0;
level = 0;
// Overall iteration control
while(!bfsIsDone(level)){
    // Inspect depth data and find the vertices in the frontier
    for(v in G){
        if (depth[v] == level){
            // Expand the vertices in the frontier
            for (v' in v.successor()){
                // update v' depth  if it is unvisited
                if(depth[v'] == inf){
                    depth[v'] = level + 1;
                }
            }
        }
    }

    // Proceed to the next iteration
    level++;
}

// When the frontier is empty, the bfs completes.
bool bfsIsDone(level){
    int counter = 0;
    for(v in G){
        if (depth[v] == level)
            counter++;
    }
    if(counter == 0) return true;
```

```
    else return false;
}
```

Given the algorithm presented above, the BFS can be roughly divided into the following three phases:

- Phase 1: Get the frontier queue based on the status array (or the depth of the graph) and completes the BFS when the queue is empty.
- Phase 2: Traverse the successors of vertices in the frontier.
- Phase 3: If the successors are unvisited, update its depth information and mark it as visited. When all the vertices in the frontier are explored, move to the next iteration.

2.2 Proposed Parallel BFS

With the sequential BFS, we can further parallelize it. As shown in 2, we keep the same three-phase design and have each phase parallelized and optimized. The details can be found in the comments in the algorithm. On top of the optimizations presented in the algorithm, we still need to perform the adaptive top-down and bottom-up switching. It requires minor modifications on most of the basic processing in the algorithm and is omitted for simplicity.

Listing 2: Parallel BFS Algorithm

```
depth[s] = 0;
level = 0;
// Overall iteration control
while(!bfsIsDone(level)){
    // Phase 1: Inspect depth of all the vertices
    // for frontier vertices, classify the
    // frontier vertices into three types, and
    // have them stored in three different
    // queues respectively. Meanwhile, visited hub
    // vertices (vertices with high degree) will be
    // cached on chip and it will be used in the
    // following processing. In addition,
```

```
// the inspecting process can be divided into multiple
// independent tasks as well.
for(v in G){
    if (depth[v] == level){
        if(v.outDegree() < 32){
            smlQueue.push(v);
        }
        else if(v.outDegree() < 256){
            midQueue.push(v);
        }
        else{
            lrgQueue.push(v);
        }
    }
    // Store visited hub vertices into on chip cache.
    else if(depth[v] < level and v.degree() > 1024){
        HubVertexCache(v);
    }
}

// Phase 2: Expand the vertices in the
// frontier. If the successors of
// the vertices haven't been visited yet,
// they will be send to a update queue
// corresponding to its depth location.
// Basically the vertices in
// different queues can update their depth
// in parallel. Vertices in the same queue
// may probably be reordered for potential
// consecutive write operation.
// Meanwhile, the expansion on each frontier
// queue can also be partitioned for parallel
// processing.
for (v in smlQueue){
```

```
        for (v' in v.successor()){
            // If v' is found in HubVertexCache,
            // it means it is visited and thus there is no
            // need reading depth[v'] from memory for
            // checking if it is visited.
            if(HubVertexCache[v'] hit){
               continue;
            }
            // update v' depth  if it is unvisited
            else if(depth[v'] == inf){
                updateQueue[mod(v', backNum)].push(v');
            }
        }
    }

    // Phase 3: Update depth of the vertices
    // in each updateQueue. As there may redundant
    // vertices in each updateQueue which may eventually
    // results in repeated memory write wasting
    // memory bandwidth. Thus we need to
    // remove the redundant vertices before
    // writing back, though there is no guarantee
    // squeezing all the redundant vertices.
    removeRedundantVertices(updateQueue);
    for (v in updateQueue){
        v.depth = level + 1;
    }

    // Proceed to the next iteration
    level++;
}
```

Here is summary of the optimization strategies of the proposed BFS algorithm targeting BFS accelerator design on FPGA or ASIC.

- Frontier queue inspection partition for parallel processing.
- Frontier vertices classification based on vertex degree.
- Hub vertices caching to avoid repeated memory access.
- Application specific processing logic for different types of vertices (i.e. vertices in different queues)
- Frontier queue partition for both parallel expansion and balanced expansion time.
- Reordering vertices to be updated to assist the parallel and consecutive depth update.
- Removing redundant vertices in each update queue compromising the hardware overhead and saving memory bandwidth.

2.3 The Proposed BFS Accelerator

Here is the overview of the proposed BFS accelerator 1. The three phases of the BFS is implemented as three different pipeline stages. Basically, the figure visualizes parallel BFS algorithm. We will not repeat the details that have been explained in previous subsection. Here we will introduce the data storage in the accelerator. All the input data including CSR data and depth information will be stored on DDR directly. While the queues are needed to connect the pipeline stages and handle the mismatched processing speed at run-time. In this figure, we assume that the frontier queue between the first pipeline and the second pipeline is implemented on DDR while the queues between the second and the third pipeline utilizes the on-chip storage.

Using DDR as the queue allows the relevant pipeline stages to operate on a larger data set but consumes larger memory bandwidth. Using the on-chip storage for buffering the intermediate data may miss some of the optimized design options, but it will save the on-chip memory bandwidth in general. Here we just show both the possible design options. Probably, the second option may be better, but anyway we will see the experiments in future.

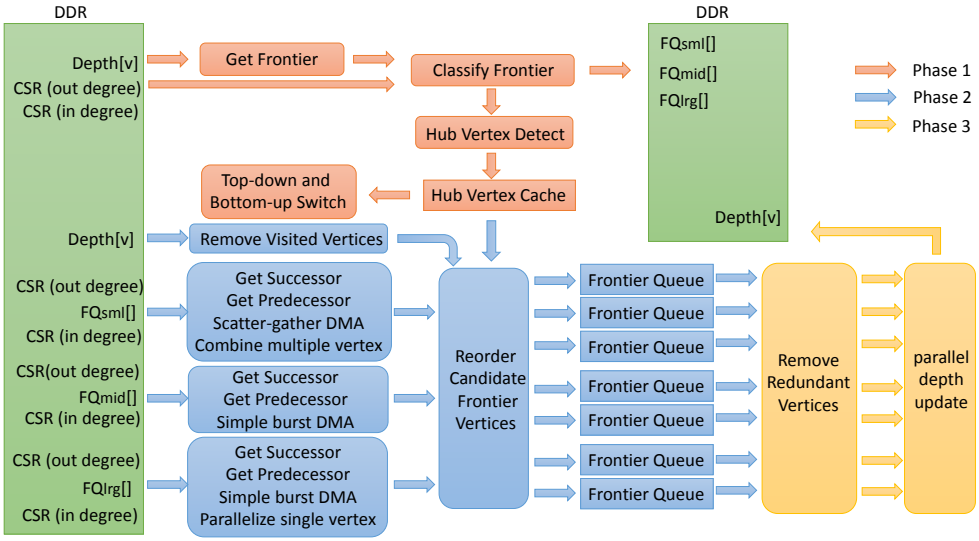

Figure 1: BFS Overview, it consists of three pipelined phases. As the design follows the BSP model, it will not move to the next iteration until Phase 3 completes.

As the top-down method needs to traverse the successors of the frontier vertices while the bottom-up method needs to traverse the predecessors of the frontier, we need to store both of them in CSR for the adaptive method.

3 About the Experiments

I am trying to develop a cycle-accurate simulator using SystemC first so that the design optimization techniques can be easily verified and evaluated. When the optimization techniques are decided, I can further build the real hardware design on the FPGAs with the most effective optimization techniques. I am not sure if it is a good idea to integrate all the optimization techniques in a single FPGA design. Maybe we can focus on one or two of them at a time for example the hub vertex caching and gradually extend the work.

On top of the BFS accelerator optimization, another thing that I want to explore is how the BFS accelerator scales with the memory bandwidth. As the BFS accelerator is sensitive to the memory bandwidth and different FPGA platforms may have various memory bandwidth available, it may be interesting to develop a bandwidth-aware BFS accelerator or an auto-tuning framework over the existing

BFS accelerator design for different FPGA platforms.

As there is no available SystemC DRAM model, I planned to use DRAMSim2 as the model first. Then I found a new simulator ramulator developed by Prof. Mutlu's group [5]. It supports many different memory architectures including DDR series, LPDDR series, HBM etc. So I decided to use it instead. Hopefully, we may further extend this work to general graph accelerator design with new memory systems such as 3D-DRAM etc.

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] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[5] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[6] 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.

[7] 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.

[8] 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.