OBFS: OpenCL Based BFS Optimizations on Software Programmable FPGAs

Cheng Liu*, Xinyu Chen†, Bingsheng He†, Xiaofei Liao‡, Ying Wang* and Lei Zhang*
*Institute of Computing Technology, Chinese Academy of Sciences
†National University of Singapore
‡Huazhong University of Science and Technology
liucheng@ict.ac.cn {xinyuc, hebs}@comp.nus.edu.sg xfliao@hust.edu.cn, {wangying2009, zlei}@ict.ac.cn

***Abstract*—Breadth First Search (BFS) is a key building block of graph processing and there have been considerable efforts devoted to accelerating BFS on FPGAs for both performance and energy efficiency. Prior work typically built the BFS accelerator through handcrafted circuit design using hardware description language (HDL). Despite the relatively good performance, the HDL based design leads to extremely low design productivity, and incurs high portability and maintenance cost. While high level synthesis (HLS) tools make it convenient to create a functionally correct BFS accelerator, the performance can be much lower the handcrafted design with HDL.**

To obtain both the near handcrafted design performance and better software-like features such as portability and maintenance, we propose OBFS, an OpenCL based BFS accelerator on software programmable FPGAs. With the observation that OpenCL based FPGA design is rather inefficient on irregular memory accesses, we propose approaches including data alignment, graph reordering and batching to ensure coalesced memory accesses. In addition, we take advantage of the on-chip buffer to mitigate the inefficient random DDR accesses. Finally, we shift the random level update in BFS out from the main processing pipeline and have it overlapped with the following BFS processing task. According to the experiments, OBFS achieves 9.5X and 5.5X performance speedup on average compared to a vertex-centric implementation and an edge-centric implementation respectively on Intel Harp-v2. When compared to prior handcrafted designs, it achieves comparable or even better performance.

I. Introduction

Previous works have shown that BFS accelerators on FPGAs can provide competitive performance and superior energy efficiency given comparable memory bandwidth (e.g., [6], [10], [12]). However, they typically develop and optimize BFS with dedicated circuits using hardware description language (HDL). The HDL based designs allow fine-grained control on resource consumption and are beneficial to the performance, but they usually take long time for development, upgrade, maintenance and for porting to a different FPGA device, which are all important concerns from the perspective of system designers.

Because of these design productivity problems, high-level synthesis (HLS) tools advance rapidly and become attractive recently. They are increasingly adopted in both industry and academia for fast prototyping and application acceleration [9], [11]. Although HLS tools improve the design productivity, the performance of HLS based designs may still be far from that of the corresponding handcrafted designs particularly for irregular applications [14], [7]. For instance, the authors in [14] showed that HLS based design of DNA sequencing can be 27X-80X slower than their handcrafted design. BFS is a typical application with inherent irregular memory accesses, and the performance of BFS reference implementation in [7] is much lower than that of the RTL implementations [5], [6].

To address the severe memory access bottleneck in BFS, we investigate aggressive memory access optimizations for OpenCL-based BFS on FPGAs. Firstly, we propose to reorder the graph layout and make the data aligned and batched. With the graph reordering, batching, and data alignment, a great number of memory accesses can be coalesced. Meanwhile, we assign the continuous data to different memory banks such that they can be accessed and processed in parallel without conflicts. According to the experiments, the proposed OpenCL-based BFS accelerator named OBFS achieves 9.5X and 5.5X performance speedup when compared to a vertex-centric implementation [7] and an edge-centric implementation [4] in OpenCL respectively. Moreover, OBFS achieves comparable performance to many handcrafted designs [5], [6], [16]. The code is open-sourced on github[1].

II. Pipelined OpenCL BFS optimization

In this work, we assume the graph is stored with compressed sparse row (CSR) format which includes a row pointer array (RPA) and a column index array (CIA). RPA contains the starting index of each vertex in CIA array while CIA consists of the incoming/outgoing neighbors.

To achieve high-throughput BFS, we develop a pipelined design for BFS, which enables direct on-chip communication between different pipeline stages without going through the shared DRAM. Essentially, BFS is a nested loop and each layer of BFS loop can be converted to a pipeline stage. The throughput of the pipeline is determined by the inner most loop kernel initially. A natural optimization is to duplicate the inner most loop kernel spatially. However, it is challenging to fulfill the massive parallel memory requests when they are issued directly. Therefore, BFS requires both pipelining optimizations and memory access optimizations.

[1]https://github.com/Liu-Cheng/bfs_with_Intel_OpenCL.git

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00056

A. Pipelining

Typically, deep fine-grained pipelining can be beneficial to the processing efficiency and implementation frequency on FPGAs. Thus, we split the inner most loop and divide BFS into five pipeline stages. The pseudo code of the BFS with fine-grained pipelining is shown in Algorithm 1. Channels implemented with FIFOs can be used to connect the different pipeline stages. With channels, stages form a producer-consumer pipeline. With the fine-grained pipelining, there are no direct dependent memory accesses in each pipeline stage. In Stage 1, it reads frontier vertices from memory. In Stage 2, RPA of the frontier vertices are read to determine their locations in CIA. In Stage 3, it reads CIA array to get the indices of the outgoing neighbors. In Stage 4, each neighbor is inspected and unvisited neighbors are considered as frontier in the next BFS. In Stage 5, the new frontier vertices are streamed to memory and the level of the frontier vertices is updated.

The five-stage pipeline implementation is used as the *baseline* for further optimizations. In the following, we develop a series of memory access optimizations to improve the efficiency of the baseline implementation.

Algorithm 1 Pseudo code of pipelined BFS algorithm

```
1: procedure BFS
2:     level[v] ← −1 where v ∈ V
3:     level[v_s] ← 0
4:     current_level ← 0
5:     frontier ← v_s
6:     l ← 0
7:     while !frontier.empty() do
8:         traverseFrontier() //Stage 1
9:         inspectFrontierRPA() //Stage 2
10:        inspectFrontierCIA() //Stage 3
11:        checkNgbVisitStatus() //Stage 4
12:        updateFrontier(l) //Stage 5
13:        exchange frontier and next_frontier
14:        l ← l + 1
15: procedure traverseFrontier
16:    for v ∈ frontier do
17:        frontier_channel.write(v)
18: procedure inspectFrontierRPA
19:    v ← frontier_channel.read()
20:    rpa.start ← RPA[v]
21:    rpa.end ← RPA[v + 1]
22:    rpa_channel.write(rpa)
23: procedure inspectFrontierCIA
24:    rpa ← rpa_channel.read()
25:    for idx ← rpa.start to rpa.end do
26:        v_out ← CIA[idx]
27:        ngb_channel.write(v_out)
28: procedure checkNgbVisitStatus
29:    v_out ← ngb_channel.read()
30:    if (level[v_out] == −1) then
31:        next_frontier_channel.write(v_out)
32:        level[v_out] ← l + 1
33: procedure updateFrontier
34:    v_out ← next_frontier_channel.read()
35:    next_frontier.write(v_out)
```

B. Memory coalescing

In BFS, there are many inefficient memory accesses, mainly caused by small data width, short burst accesses, and false dependent memory accesses. To address those issues, we try to coalesce the memory accesses with the following approaches.

First, we have both RPA and CIA arrays aligned. For the RPA array, it is changed to the new sequence (RPA[0], RPA[1] - RPA[0]), (RPA[1], RPA[2] - RPA[1]), ..., ($RPA[i]$, $RPA[i+1]$ - $RPA[i]$), ... where i is the vertex index. In addition, the two 32-bit random RPA read operations are combined to a single memory read aligned to 64-bit. For CIA array, it is typically read from $CIA[RPA[i]]$ to $CIA[RPA[i+1]$ sequentially when traversing the neighbors of vertex i. To ensure aligned neighbor traverse, we have both ends aligned to $B \times 32$-bit where B refers to the number of batched data. Basically, B vertices batched together can be read each time in $inspectFrontierCIA()$. Additional padding i.e. '-1' may need to be added to ensure that CIA range of each vertex can be fully divided by B. With the data alignment and batching, the overall memory access efficiency improves despite the additional padding overhead.

Second, B neighbors must be processed in parallel in $checkNgbVisitStatus()$ to guarantee balanced pipelining, and thus B also determines the number of parallel processing units. As B neighbors are independent, it is natural to distribute them to the B processing units. Nevertheless, B parallel processing units of $checkNgbVisitStatus()$ may still be stalled due to the memory conflicts when the B data to be accessed are located in the same memory bank.

Third, we reorder the outgoing neighbors of each vertex to ensure no conflicts in the B parallel update units. Algorithm 2 details the reordering together with the data alignment. The original CSR graph consists of RPA array and CIA array and the reordered new CSR graph is stored in $newRPA$ array and $newCIA$ array respectively. The reordering roughly involves two steps. 1) Neighbors of each vertex are assigned to B queues based on the modulo result of neighbor index and B. 2) We get one data from each queue to construct a batch of B data. Repeat the process until all queues are empty. All the batched data are put into $newCIA$ array. '-1' will be used as padding to construct the batched data when the corresponding queue is empty earlier. $newRPA$ is updated accordingly to ensure the CSR storage format.

With the data alignment, graph batching and reordering, memory accesses in $inspectFrontierRPA()$, $inspectFrontierCIA()$ and $checkNgbVisitStatus()$ are coalesced for parallel processing. Note that graph reordering and data alignment are considered as graph pre-processing, which are performed on the CPU.

C. On-chip bitmap buffering

After the graph reordering and data alignment, there are still many random accesses in $checkNgbVisitStatus()$, i.e., $level[v\_out]$. Although B independent memory access requests to DRAM can be issued in parallel, the random memory accesses remain inefficient. Instead of using $level$ to determine the visiting status, we utilize bitmap to represent the visiting status directly, which is also used in some of the previous BFS accelerator designs [7]. Each vertex needs only one bit to determine whether it is visited. The latest FPGAs with up

Algorithm 2 Data alignment and reordering of a CSR graph

```
1: queue[B]
2: eid ← 0
3: for v ∈ V do
4:     newRPA[2v] ← eid
5:     for (i ← RPA[v] to RPA[v + 1]) do
6:         j ← mod(CIA[i], B)
7:         queue[j].write(CIA[i])
8:     D ← 0
9:     while (there are still non-empty queues in queue[B]) do
10:        D ← D + 1
11:        for (i ← 0 to B) do
12:            eid ← eid + 1
13:            if (!queue[i].empty()) then
14:                newCIA[eid] ← queue[i].read()
15:            else
16:                newCIA[eid] ← −1
17:    newRPA[2v + 1] ← D × B
```

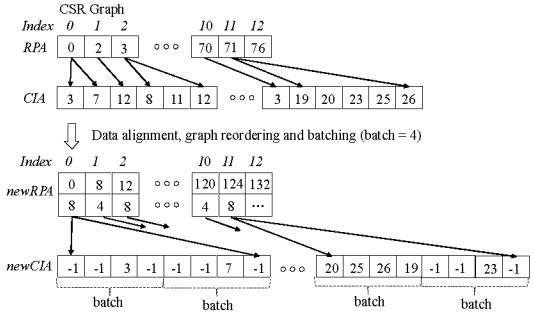

Fig. 1: CSR layout after the data alignment, graph reordering and batching

to 500Mb on-chip buffer can accommodate graphs with 500M vertices in theory, which fulfills the requirements of many realistic graphs such as twitter2010 (42M vertices). Meanwhile, we need to split the visiting status bitmap into B banks to fit the parallel processing of $checkNgbVisitStatus()$. The visiting status of a vertex v will be put into the ith memory bank where $i = mod(id, B)$. The bitmap layout is consistent with the graph reordering and batching.

D. Level update shifting

To ensure level query of any vertex in the graph after BFS, we need to update level information to array $level[]$. Since the level update i.e. $level[c\_out] \leftarrow l+1$ includes massive random writes to DRAM, it is slow and can affect the main BFS pipelining when it is put in the $inspectNgbVisitStatus()$. Nevertheless, we notice that it does not affect the next BFS iteration thanks to the on-chip bitmap. Since the frontier vertices are already stored in memory in $updateFrontier()$, we can always perform the level update based on just the stored frontier. Therefore, we can safely shift level update processing out from the $checkNgbVisitStatus()$ and simplify the main BFS pipelining. Meanwhile, we can postpone the level update without blocking the BFS processing. Therefore, we separate the level update from the main BFS pipeline and overlap it with the following BFS processing task (consider a task implemented by multiple BFS operations). Alternatively, it can also be processed on the attached host CPU.

TABLE I: Graph Benchmark

Name	# of vertex	# of edge	Type
YouTube (YT)	1,157,828	2,987,624	Undirected
Live Journal (LJ)	4,847,571	68,993,773	Directed
Pokec (PK)	1,632,804	30,622,564	Directed
R19	524,288	16,777,216	Directed
R21	2,097,152	67,108,864	Directed

TABLE II: Performance comparison to existing OpenCL based BFS

Benchmark	YT	LJ	PK	R19	R21
OBFS	79.4	475.7	557.8	787.7	934.2
Spector	12.01	-	-	64	-
Speedup	6.6X	-	-	12.3X	-
Work[4]	25.9	63.9	81.3	183.5	157.8
Speedup	3.1X	7.4X	6.8X	4.3X	5.9X

III. Experiments

In this section, we measure the overall performance of the optimized OpenCL-based BFS accelerator on Intel Harp-v2 [8] with a set of representative graphs. Then we have an end-to-end comparison to the existing OpenCL based BFS implementations and summarize the performance of prior BFS implementations on FPGAs.

A. Experiment Setup

The graphs used for evaluation includes three real-world graphs and two synthetic graphs as listed in Table I. The real-world graphs are from social network (SN) while the synthetic graphs are generated with R-MAT model. For R-MAT graphs, we follow the Graph 500 benchmark parameters ($A = 0.59, B = 0.19, C = 0.19$). The size of a R-MAT graph is determined by the scale factor S and the edge factor E, which means the graph has 2^S nodes and $E \times 2^S$ edges. The configurations for the two graphs R19 and R21 are ($S = 19, E = 32$) and ($S = 21, E = 32$), respectively. In the experiments, the OpenCL compiler is based on Intel OpenCL SDK 16.0.2 for FPGA. The FPGAs used in Harp-v2 is Arria 10 GX1150[8]. The performance metric is million traversed edges per second (MTEPS). To ensure fair comparison, we tested 64 non-trivial BFS and averaged the the performance.

B. Performance evaluation

The performance of the proposed BFS accelerator OBFS as presented in Table II achieves up to 934.2 MTEPS on the R21 and 567.0 MTEPS on average when the batch size (B) is 16. Compared to the OpenCL based vertex-centric BFS implementation from Spector [7], OBFS shows 9.5X performance speedup, though Spector fails on the other three graphs due to memory access stall. When compared to the OpenCL based edge-centric BFS implementation from Chen's work [4], OBFS achieves 5.5X performance speedup on average. The comparison reveals that OBFS achieves significant performance speedup over existing OpenCL based BFS implementations.

On top of the end-to-end comparison with OpenCL-based approaches on Harp-v2, we also compare OBFS to prior

TABLE III: Performance of BFS accelerators on FPGA-DRAM platforms

System	Dataset Type	Bandwidth (GB/s)	Hardware Platform	Design Method	MTEPS /FPGA
Work[13]	SN	0.1	Virtex-5	S&RTL	160-790
Work[3]	fMRI	80	HC-2	M&RTL	62.5-650
CyGraph[1]	R-MAT	80	HC-2	M&RTL	420-550
Work[12]	R-MAT	3.2	Zedboard	M&RTL	90-255
FPGP[5]	SN	12.8	VC707	M&RTL	122
ForeGraph[6]	SN	19.2	VCU110	S&RTL	364-1069
Work[16]	R-MAT	12.8	Harp-v1	M&RTL	330-670
Work[15]	SN	19.2	Ultrascale+	S&RTL	1500-3500
OBFS	R-MAT	16	Harp-v2	M&OCL	861
OBFS	SN	16	Harp-v2	M&OCL	371

RTL-based BFS accelerators. These BFS accelerators may vary on many different aspects such as graph types, memory bandwidth, hardware platforms, and evaluation methods. A summary of these BFS acceleration works is presented in Table III. Most of the works used either social network (SN) graphs which are the same or similar to our benchmark sets or R-MAT graphs. High-performance multi-FPGA platforms such as Convey HC-2 [2] have higher memory bandwidth while the single-FPGA platforms typically have relatively limited DRAM bandwidth. These works also differ on evaluation methods. Some of them obtained the performance via measurement on realistic hardware while others rely on simulation. We use 'M' and 'S' to represent real hardware measurements and simulations, respectively.

According to the performance comparison in the Table III, OBFS achieves competitive performance to most prior RTL designs. This demonstrates the potential of utilizing OpenCL for BFS with inherent irregular memory accesses. We notice that the performance of OBFS is not as good as that reported in the simulation result [15]. There may be mainly two reasons for this. First, current OpenCL has many implementation constraints which may limit the adoption of some optimization strategies. For instance, Current Intel OpenCL compiler targeting Harp-v2 does not allow on-chip buffer sharing between different kernels (This will be resolved in compilers supporting OpenCL Spec2.0, which defines global shared on-chip buffer to enable the feature.), so it poses constraints on BFS pipelining and data path parallelization. Second, OpenCL based design runs at lower clock frequency due to the lack of low-level circuit control mechanisms. For example, RTL design in [15] runs at 250 MHz, but OBFS ranges from 160 MHz to 200 MHz. While the OpenCL tools keep evolving. it can be expected that the performance gap between OpenCL based designs and RTL designs might shrink in the near future.

IV. Conclusion

Handcrafted BFS accelerators with HDL usually suffer high portability and maintenance cost despite the relatively good performance. OpenCL-based BFS accelerators can greatly alleviate these problems, but it is challenging to achieve satisfactory performance due to the inherent irregular memory accesses. In this work, we propose a series of high-level optimization approaches to improve the irregular memory access efficiency and pipelining in BFS. Compared to reference BFS implementations with a vertex-centric framework [7] and an edge-centric framework [4], the proposed OBFS achieves 9.5X and 5.5X performance speedup on average respectively. When compared to the prior HDL-based BFS accelerators, OBFS also achieves competitive performance.

Acknowledgement

This work is supported in part by the National Natural Science Foundation of China under grant No. 61874124, No. 61929103, and No. 61902375. Thanks to Intel for access to the CPU+FPGA system through the Hardware Accelerator Research Program (HARP).

References

[1] O. G. Attia, T. Johnson, et al. Cygraph: A reconfigurable architecture for parallel breadth-first search. In *Parallel & Distributed Processing Symposium Workshops, 2014 IEEE International*, pages 228–235, 2014.

[2] J. D. Bakos. High-performance heterogeneous computing with the convey hc-1. *Computing in Science & Engineering*, 12(6):80, 2010.

[3] B. Betkaoui, Y. Wang, et al. A reconfigurable computing approach for efficient and scalable parallel graph exploration. In *Application-Specific Systems, Architectures and Processors, 2012 IEEE 23rd International Conference on*, pages 8–15, 2012.

[4] X. Chen, R. Bajaj, Y. Chen, et al. On-the-fly parallel data shuffling for graph processing on opencl based fpga. In *29th International Conference on Field Programmable Logic and Applications*, 2019.

[5] G. Dai, Y. Chi, et al. FPGP: graph processing framework on FPGA A case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 105–110, 2016.

[6] G. Dai, T. Huang, Y. Chi, et al. ForeGraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 217–226, 2017.

[7] Q. Gautier, A. Althoff, P. Meng, and R. Kastner. Spector: An opencl fpga benchmark suite. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 141–148. IEEE, 2016.

[8] P. Gupta. Accelerating datacenter workloads. In *26th International Conference on Field Programmable Logic and Applications*, 2016.

[9] D. Koch, F. Hannig, and D. Ziener. *FPGAs for Software Programmers*. Springer, 2016.

[10] X. Ma, D. Zhang, and D. Chiou. Fpga-accelerated transactional execution of graph workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 227–236. ACM, 2017.

[11] R. Nane, V. Sima, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, Oct 2016.

[12] Y. Umuroglu, D. Morrison, and M. 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.

[13] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna. A message-passing multi-softcore architecture on fpga for breadth-first search. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 70–77. IEEE, 2010.

[14] L. Wu, D. Bruns-Smith, Nothaft, et al. Fpga accelerated indel realignment in the cloud. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 277–290, 2019.

[15] P. Yao, L. Zheng, X. Liao, H. Jin, and B. He. An efficient graph accelerator with parallel data conflict management. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, page 8. ACM, 2018.

[16] S. Zhou and V. K. Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *the 29th International Symposium on Computer Architecture and High Performance Computing*, pages 137–144, 2017.