Dr. BFS: Data Centric Breadth-First Search on FPGAs

Eric Finnerty Zachary Sherer Hang Liu Yan Luo
University of Massachusetts Lowell

ABSTRACT

The flexible architectures of Field Programmable Gate Arrays (FPGAs) lend themselves to an array of data analytical applications, among which Breadth-First Search (BFS), due to its vital importance, draws particular attention. Recent attempts that offload BFS on FPGAs either simply imitate the existing CPU- or Graphics Processing Units (GPU)- based mechanisms or suffer from scalability issues. To this end, we introduce a novel data centric design which extensively extracts the potential of FPGAs for BFS with the following two techniques. First, we advocate to partition and compress the BFS algorithmic metadata in order to buffer them in fast on-chip memory and circumvent the expensive metadata access. Second, we propose a hierarchical coalescing method to improve the throughput of graph data access. Taken together, our evaluation demonstrates that the proposed design achieves, on average, 1.6× and 2.2× speedups over the state-of-the-art FPGA designs TorusBFS and Umuroglu, respectively, across a collection of graph datasets.

ACM Reference Format:
Eric Finnerty Zachary Sherer Hang Liu Yan Luo. 2019. Dr. BFS: Data Centric Breadth-First Search on FPGAs. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317802

1 INTRODUCTION

BFS[1] is a building block to analyze graphs which have a wide range of applications, including Blockchain networks [1], social and computer networks [2], and chemical compound design graphs [3]. The significance of BFS is further manifested by the fact that Graph 500 [4], a supercomputer ranking organization, exploits BFS [5, 6] – to rank the world most powerful supercomputers, e.g., Sunway TaihuLight, MilkWay-2 and K computer.

It is commonly recognized that BFS is a data intensive application which spends the majority of the time accessing **algorithmic metadata** (i.e., vertex states) and the rest of the time retrieving **graph data** (i.e., neighbor list). In particular, BFS conducts the following three tasks **i)**. loading the neighbors of an active vertex, **ii)**. checking the statuses of each neighbor and **iii)**. marking the unvisited neighbors as current level vertices. Among them, **i)** is about graph data access while the rest belongs to metadata. Obviously, data access is at the core of graph traversal thus motivates this design of *data centric graph traversal on FPGAs*.

[1]This paper uses graph traversal and BFS interchangeably.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317802

FPGAs are becoming an increasingly attractive platform to deploy data analytical applications thanks to the flexible and customizable on-chip resources (such as processing logics, on-chip memory, interconnects, etc.). In this context, one can easily tailor FPGA fabricate toward specific needs exhibited by various workloads and extract parallelism from algorithms that would otherwise impossible on CPUs and GPUs. It is important to mention that the low power consumption of FPGAs is also tempting for various applications.

Attracted by these advantageous features, an array of endeavors has surged to offload BFS on FPGAs, which, however, falls short in two ways. On one hand, the work in [7, 8] simply extends CPU/GPU designs on FPGAs without exploiting the potential of FPGAs, such as hardware reconfigurability). On the other hand, we also observe efforts [9, 10] which are more tailored to the strengths of the FPGA platform, but fall short at metadata access and accommodating large graphs.

To this end, we introduce a novel data centric design which extensively extracts the potential of FPGA for BFS. In particular, this design virtually eliminates the expensive metadata access from external memory, as well as largely enhances the graph data retrieval. Taken together, our design is 1.6× and 2.2× faster than the state-of-the-art projects, such as, TorusBFS [9] and Umuroglu et al. [10]. In particular, this project comes with the following two contributions.

First, we advocate to partition and compress the BFS algorithmic metadata in order to buffer them in fast on-chip memory and circumvent the expensive metadata access. In particular, our initial intent is to load the entire metadata into on-chip memory before the computation so that the random metadata access is cached on-chip. However, this design is limited by the size of on-chip memory. To combat this scalability issue, we exploit the "unpopular" 1-D vertical partition to divide the graph by destination vertices so that each partition only accesses a specific (i.e. smaller) subrange of the metadata. Further, we compress the metadata into a bitwise status array in order to fit more metadata on-chip thanks to the bit-addressable nature of FPGA memories.

Second, we propose a hierarchical coalescing method to improve the throughput of graph data access. Particularly, we decouple the processors for moving data from external memory in core (i.e., **data access processor**) and the follow-up on-chip status checks/updates (i.e., **computing processor**). That is, we confine the amount of data access processors to equate the number of the memory channels in order to avoid contentions while maximize the amount of parallelism to in turn maximize the computing speed. In short, this design coalesces the memory accesses to be accessed from a single, parallel accessor module. Second, we coalesce the memory access from different requests into a single transaction. Third, we always load the memory bus amount of graph data on-chip to simplify the memory transaction issuance thus improve the throughput.

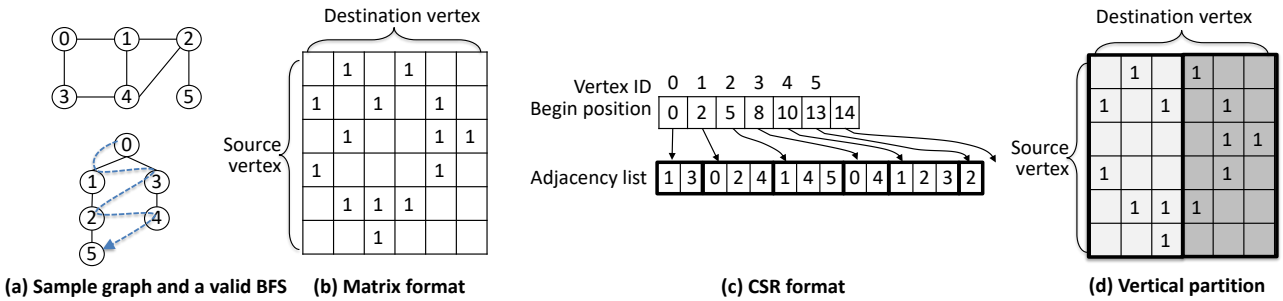

Figure 1: A (a) Sample graph, and its (b) Matrix format, as well as (c) CSR format. And (d) Vertical partition is adopted by this work for better cache designs.

While all these data centric designs are probably achievable atop general purpose CPU and GPUs, we find this effort would involve overwhelming amount of software control-flows (i.e., overhead) [11**?**]. In contrast, our work simply fabricates all these designs in hardware circuits on FPGAs thus enjoys exceptional performance.

The rest of paper is organized as follows. Section 2 describes background and related work. Section 3 presents the design principles of this work. Section 4 discusses the techniques proposed in this paper. The experiments and results are presented in Section 5. Section 6 concludes.

2 BACKGROUND AND RELATED WORK

This section presents the essential background knowledge for this work, namely BFS and graph partitioning.

2.1 Graph and Graph Representation

An unweighted graph can be represented as $G = (V, E)$, where V and E are the vertex and edge sets of the graph, respectively. Since mainstream FPGAs address memory components in a linear fashion, one can hardly store a graph in the geographical manner (as shown in Figure 1(a)). Instead, a sparse matrix storage format such as compressed sparse row (CSR), compressed sparse column (CSC), edge list and etc, is chosen.

Figure 1(b) plots the matrix format representation of the sample graph from Figure 1(a). In particular, the matrix format assumes each row and column represents a specific source and destination vertex, respectively. Each '1' stands for an edge in sample graph. For example, the '1' at 0-th row and 1-st column stands for edge (0, 1). Since Figure 1(a) is an undirected graph, one should also notice the presence of edge (1, 0) in Figure 1(b).

Figure 1(c) presents the CSR format representation which is also adopted by mainstream accelerator-based BFS attempts, such as Gunrock [12], Enterprise [6] and TorusBFS [9], for better locality and space consumption. The CSR format stores all the destination vertex IDs of each edge in the adjacency list array and uses a begin position array to denote the beginning index of the destination vertices for each source vertex. Therefore, the adjacency list and begin position arrays consume the space of $|E|$ and $|V| + 1$, respectively.

2.2 Breadth-First Search (BFS)

BFS starts from a root vertex and subsequently traverses through a graph level by level. Note, the next level traversal only starts once current level work is finished, which is sketched (dotted, blue line) in Figure 1(a). The level information of each vertex is stored in a metadata structure – status array – which has size $|V|$.

At each iteration, BFS conducts two tasks – expansion and inspection – concerning both source and destination vertices. In particular, expansion loads the adjacency lists (i.e., destination vertex) of vertices from prior iteration. Inspection will check the status of adjacent vertices (i.e., destination vertex) and mark those unvisited ones as current level vertices.

As the traversal proceeds, the updated destination vertices from the current inspection will become the source vertices for the expansion of the next iteration. In our design, we choose to use a CSR-formatted graph, with the vertex status array serving as metadata for the traversal.

2.3 Field Programmable Gate Arrays (FPGAs)

An FPGA is a reconfigurable computing platform that allows designers to implement custom hardware architectures without the overhead in time and production cost associated with Application-Specific Integrated Circuit (ASIC) production. Many resources are included on chip to aid in the production of these designs like adaptive logic modules (ALMs), memory blocks (BRAM), embedded multipliers for digital signal processing, and other peripheral interfaces like external memory (DDR, HMC, etc.) and Peripheral Component Interconnect - express (PCIe) interfaces.

Today, many FPGA platforms are geared towards the production of hardware accelerators. The flexible routing architecture and parallel resources allow for many types of designs that would be impossible in hardened silicon or in software, the main advantage being deep pipelining and the simultaneous operation of replicated custom structures. Coupled with high level tools for moving data over network interfaces or PCIe, data can be moved quickly on or off the device en masse.

Some FPGAs are also packaged as a System on Chip (SoC) with a CPU on die, usually ARM-based. Intel's Cyclone, Arria, and Stratix platforms all have ARM SoC options, with the idea that they could be implemented in a standalone system with custom hardware. The FPGA and ARM CPU are coupled by an AXI bridge that allows for communication between the hardware and software, allowing for hardware-software codesign that would be impossible even on an FPGA mounted on PCIe.

2.4 Related Work

Recent years have witnessed BFS acceleration on a variety of processors, such as, multi-core CPU [13**?**] GPUs [5, 6, 12], Xeon Phi [14, 15], and FPGA [7–10, 16]. In this section, we compare our work against the most relevant FPGA platforms.

There exist FPGA-based implementations of both hybrid vertex- [17] and edge- [18] centric paradigms in pursuit of better performance: vertex centric designs do well when there are few updates per iteration, and edge centric designs do well in the opposite case. Successful designs have been proposed with both, including GraphGen [**?**] for vertex-centric processing and [8] for edge-centric processing. Hybrid designs have also been proposed that attempt to switch between the two paradigms such that the strengths of each paradigm can be used on the iterations where they are needed [7]. We use a vertex-centric design because it fits best with the partitioning scheme we are using.

To alleviate vertex status access pressure, TorusBFS [9] caches the entire status array in on chip memory. Since various PEs possesses different on-chip memory, a toroidal message-passing network is exploited to maintain coherence. Clearly, the torus connection can rapidly drain up the FPGA resources and this design cannot scale to large number of PEs thus experiencing low performance. We notice that Zhou et al. [19]'s multi-ported block RAMs can, in a small scale, address this issue. However, with multi-ported block RAMs comes the need to control simultaneous writes and reads, which can increase design complexity and consume valuable device resources.

Many designs use multiple processing elements when accessing external memory, but these designs have high memory bandwidth [20] or use exotic memory technologies like hybrid memory cube or HBM [21, 22]. Not all accelerators have access to these technologies; in fact most FPGA acceleration boards today use DDR4 DRAM as their external memory, some with only a single interface to access it. These boards are therefore bound by the shortcomings of this memory technology, namely poor random access performance and low maximum bandwidth, with the advantage being low cost per unit capacity. Our proposed design is optimized for use with DRAM, and seeks to address its shortcomings with an architecture designed around tightly controlled memory accesses to a single memory interface, and extracts parallelism from the width of the bus, not the bandwidth.

3 DESIGN PRINCIPLES

This section briefly presents the twin design principles for the proposed data-centric graph traversal on FPGAs. And the end of each subsection, we also point out the unique advantages of FPGAs that favor our designs.

#1. Buffering the partitioned and compressed metadata in on-chip memory.

We advocate to buffer metadata in fast on-chip memory stemming from the fact that BFS spends majority of the runtime checking and updating the metadata, as made evident by mainstream projects [5, 6, 23]. This phenomenon originates from the following two facts: first, accessing the graph data (i.e., neighbor lists) of each frontier is more or less sequential [6]. Second, metadata access is decided by the vertex IDs of neighbor lists where geographically close vertices are often assigned with random vertex IDs. Note, it is computationally intractable to assign continuous IDs to all nearby vertices [24].

Despite the fact that buffering all metadata in on-chip memory can potentially expedite the graph traversal, it also limits the size of a solvable graph to the budget of the on-chip memory (i.e., ∼10s of MBits). This evolves into a noticeable problem since the majority of nontrivial real-world graphs contain an amount of metadata that greatly exceeds the storage capacity of on-chip memories[23].

In this work, we implement a graph partitioning method to address this problem. As shown in Figure 1(d), if we vertically partition the graph and solve each partition at a time, the accelerator will only access and update the metadata of destination vertices in a particular range. In this context, one only needs to buffer that specific subrange of metadata in limited on-chip memory.

While both designs can be implemented on general purpose GPUs (given its manually controllable shared memory), FPGAs stand out with two advantages. First, FPGAs provision bit-addressable on-chip memory which allows us to buffer 8× more metadata. It is important to mention that FPGAs support bitwise atomic operations in on-chip memory which are absent from GPUs. Second, the Arria 10 FPGA features 44 Mbits on-chip memory, which is several times larger than state-of-the-art GPUs [25].

#2. Hierarchically coalescing graph data access.

Once metadata is buffered on-chip, graph data becomes the next bottleneck. Since real-world graphs can arrive at magnitude of GBs, we have to store the graph data in external memory and stream in for computation. Consequently, our second design contribution hierarchically combines graph data accesses in order to avoid congestions and maximize throughput.

First, at the higher level, we decouple the data access processor (definition in Section 1) and computing processor in order to maximize data throughput and processing power, respectively. In particular, we first restrict the amount of graph data loaders to be the same as the number of memory channels in order to avoid I/O contentions. Second, since more vertex processors will yield larger computing power thus faster processing speed, we fabricate much higher amount of processors for follow-up computing, i.e., checking and updating neighbor status. Simply put, we coalesce the memory access from multiple computing processors to avoid contentions at memory channel.

Second, toward data access processor design, we combine multiple data requests to minimize the amount of transactions, as well as always loading full memory bus bandwidth amount of data (i.e., 512 bits) in order to simplify the transaction issuance. Note, both designs can maximize the throughput of transactions. Because the first design is straightforward, we will explain the second one with a counter example as below. Reading a single integer value will still require reading a whole line of memory bus, and computing the offsets of a memory block that is smaller than bus bandwidth will delay the issuance of the request. Taken together, we propose to directly load that full bus bandwidth amount of data.

It is important to mention that all these designs, i.e., decoupling data and computing processors, coalescing and simplifying data

access transactions are naturally fabricated on FPGAs which will, otherwise, require sophisticated control-flows (i.e. overhead) to implement on general purpose accelerators [11].

4 HARDWARE IMPLEMENTATION

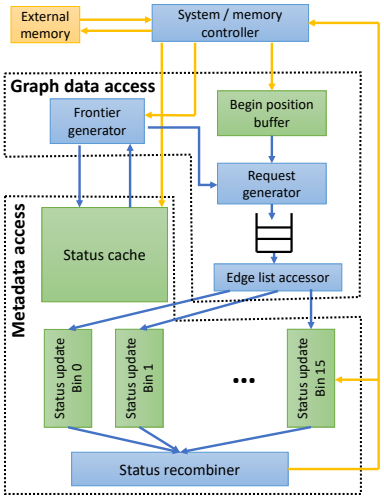

Figure 2: An overview of the accelerator design, where the green and yellow boxes are on-chip and external memory, respectively. The yellow and blue lines are external memory data and on-chip data flows, respectively.

This section presents the implementation of the data and metadata access optimized accelerator. Figure 2 briefly depicts a high level diagram of the hardware implementation. It is designed to separate the metadata access from the data access, improving the processing speed as well as the utilization of the memory. Below, we will discuss these designs in detail.

4.1 Metadata Access Design

The metadata processing section of the design focuses on creating an optimized memory access pattern for metadata retrieval, as well as accommodate the graphs with nontrivial amount of vertices.

Partitioning is a preprocessing step in which the graph is vertically partitioned such that the edges with destination vertices in the range *pn* to *p(n+1)* are in partition *n*. Departing from the mainstream 1-D horizontal partitioning method used by [5, 6], which partitions the graph by source vertices, this work chooses the 1-D vertical partition hinging upon the "pushing" nature of typical BFS. That is, BFS always pushes updates from source to destination. In this case, the status check and potential status updates from each partition will always be restricted to a limited range.

In detail, this partition is achieved within three steps. Firstly, we will scan through the graph and classify various edges into different partitions based upon destination vertices. Assuming we have p partitions, the destination vertex range for partition i is $[\frac{|V|\cdot i}{p}, \frac{|V|\cdot(i+1)}{p}]$. For instance, if we have $p = 2$ partitions, partition 0 and 1 are account for the edges whose destination vertices falling in range of $[0, \frac{|V|}{2}]$ and $[\frac{|V|}{2}, |V|]$, respectively. Second, the corresponding metadata of each partition is also identical. Third step converts this edge list format into CSR format.

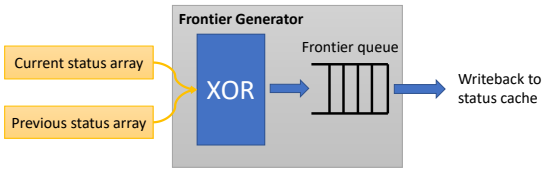

Figure 3: A detailed representation of the frontier finder module.

Once preprocessing is accomplished, we will load the metadata on-chip twice in each iteration. The first loading is for the **frontier generator** which highlights the frontier vertices in the current partition. Frontier highlighting is done by checking vertex status and finding those vertices that were visited at the previous level. To streamline this process, we represent the visited status of each vertex as a single bit, resulting in a 32x compression of status versus storing them as integers. The benefit of this is twofold: it allows for a reduced footprint for metadata in memory, and it allows for faster frontier highlighting by computing the bitwise XOR of the status of the current level with that of the previous level, leaving only those vertices that were only visited on the last level. The frontier generator writes these frontiers back in to the status cache so that at the end of this process the status cache will contain all of the frontiers for the partition on chip. The reading, frontier highlighting, and writeback steps are all pipelined in the frontier generator for optimal performance.

The second loading is for the edge list accessor to check and update the states of neighboring vertices in a pipelined fashion. As we will discuss shortly, edges are packed and loaded and extracted from 512 bits of data. In order to process them, 16 separate block rams are used to hold the status updates corresponding to relative offsets within a memory line (e.g. the first location in a memory line will always update only the first block RAM, the second will always update the second block RAM, etc). At the end of the partition traversal, the status updates in that partition will be striped across all of these block RAMs. Receiving a line of memory and updating the bins is a pipelined process, and the pipelined edge list accessor can update all bins in a single cycle, resulting in a maximum of 16 status updates per cycle.

Once a partition is solved, the copy step can begin. The status combiner is a separate module that bridges the gap between the updated status and current status. During the copy phase, the status combiner performs a simultaneous read from all of the update bins and combines the resulting data using a bitwise OR operation. This creates a single memory line that holds all of the updates that were previously striped across the update bins. The resulting line is then written back to the external memory. Once all of the updated status has been committed to the memory, the next partition can be fetched to the chip and processed. Once all partitions have been processed, and no frontiers were found, the traversal is complete. Otherwise, the accelerator will start the next level traversal.

4.2 Data Access Design

Graph data access is simply related to the processing of CSR data (i.e., begin position and edge list), which consists of two steps – begin position generation and edge list loading/filtering.

The **request generator** takes the frontiers created by the frontier generator and creates memory requests for the neighbors of

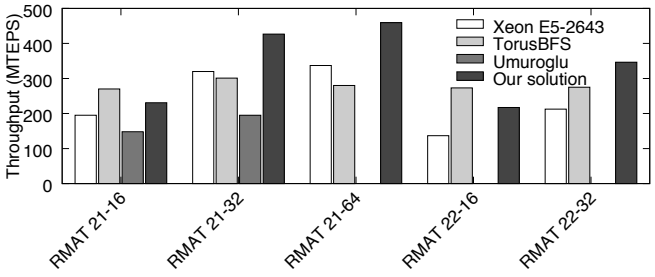

Figure 4: Comparing our design against state-of-the-art projects, i.e., TorusBFS [9] and Umuroglu [10].

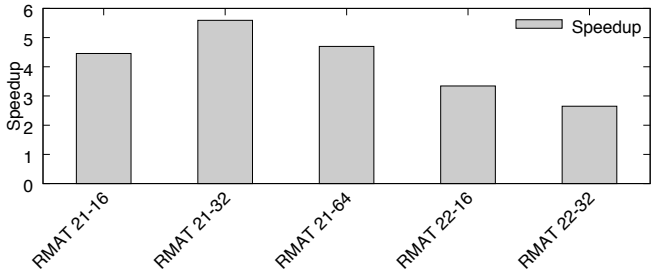

Figure 5: Speedup of using a bigger cache.

Resource	Absolute Utilization	% of Available
Logic (ALMs)	68,224/251,680 ALMs	27%
Block RAMs	19,258,360/43,642,880 Mb	44%
PLLs	54/96	56%

Table 1: The device utilization of our design on our target platform.

those vertices to be issued during the data access step. The request generator fetches begin position data from memory and matches it with frontiers generated in the metadata access step. The begin position data is read in sequential mode for optimal memory bandwidth utilization. In this step, it is beneficial to coalesce adjacent memory requests into single transactions to reduce the total transaction count. Consider vertices 1 and 2 in figure 1(c). If these vertices are both valid on an iteration, then two requests would be formed containing $\{0, 2, 4\}$ and $\{1, 4, 5\}$. Since these are sequential in memory, it would be optimal to combine them into one request containing $\{0, 2, 4, 1, 4, 5\}$. The request generator does this using a pipelined combination network to minimize the number of total memory accesses and to promote burst operation. It also allows the request generator to produce a new request on each cycle.

To increase memory performance, the request generator also includes a memory buffer to allow for the pre-fetching of begin position data. The memory buffer fetches memory lines from the external memory and enqueues them on a begin position queue. Once the queue is nearly empty, the memory buffer process wakes up and refills itself. Memory buffer reads are done in sequential burst mode, increasing memory bandwidth utilization.

Valid edge extraction. When utilizing full memory lines to access edges, it is likely that a line may contain invalid edge data. For the accelerator to be aware of which edges in a line are valid, it generates bitmaps that correspond to the first and last lines of a memory request. The bitmaps are 16 bits wide, corresponding to the number of edges we contain in a memory line. A 1-bit in the bitmap denotes a valid edge for that request in that line. Bitmaps are not required for lines between the first and last in a request because all of the edges in these lines are known to be valid due to the CSR graph representation. These bitmaps are pushed onto the request queue with the memory requests that they correspond to, and are generated as part of the memory request pipeline.

5 EVALUATION AND RESULTS

We implemented this system in 7,034 lines of Verilog code targeting the Arria 10 SoC development kit, clocked at 220MHz for the 1Mb cache design and 200MHz for the 2Mb design. Table 1 shows the design's utilization on this platform. We have chosen RMAT graphs for our testing because of their use in evaluation of other works [10, 20]. We use graphs of scale 21, and 22 with average degrees of 16, 32, and 64 for each. The scale 22 with degree 64 could not be tested on our platform due to external memory size limitations, but unlike the internal memory restrictions of [9] external memory can be more easily expanded, allowing for scaling beyond scale 22. Graphs were generated using the Graph500 graph generator at MGHPCC using the default parameters (A=0.57, B=0.19, C=0.19).

5.1 Comparison with state-of-the-art

Figure 4 demonstrates the effectiveness of the proposed design over Umuroglu [10] and TorusBFS [9], and an Intel Xeon E5-2643 CPU. We show a 4.1× speedup on average over the Xeon CPU, obtained as a geometric mean of the speedups on each graph. Over state of the art FPGA designs, we achieve up to a 1.6× improvement over TorusBFS [9] and up to a 2.1× improvement over Umoroglu [10]. It is also obvious that because of keeping scalability in mind, our system can dramatically outperform the other projects when the size of the graph soars in Figure 4. The design in [10] reports a figure for MTEPs/GB/s to measure the usage of the memory interface. Our design does not outperform [10] in this regard, because we do not choose to switch to an edge-centric paradigm at the dense levels. While it decreases the amount of burst mode accesses to the memory, thus reducing overall bandwidth utilization, our solution eliminates redundant work, therefore improving the throughput. The design in [10] also suffers from scalability issues due to the fact that it stores the status array in on-chip memory.

In addition to the performance gain, we found that we also surpass the Xeon-powered server in total system efficiency. We measured the power consumption each full system from mains power at the power supplies. On the FPGA platform we were able to achieve 15.3 MTEPS/watt peak efficiency, while on the server we could only achieve 1.5 MTEPS/watt, resulting in an 10x increase in efficiency for our platform. For our platform, we show a power consumption of 16 W with a 2 Mb vertex cache, and 14.5 W with a 1 Mb vertex cache. Through this, we also observe that using a smaller vertex cache can result in lower power consumption due to the reduced memory cost.

Resource utilization. Table 1 investigates the resource utilizations of our design. We use a large percentage of the available block RAM on chip (44%), mostly taken up by our status cache.

5.2 Performance impacts of optimizations

Figure 7 demonstrates the memory reduction yielded by coalesced memory access by comparing against our design with memory

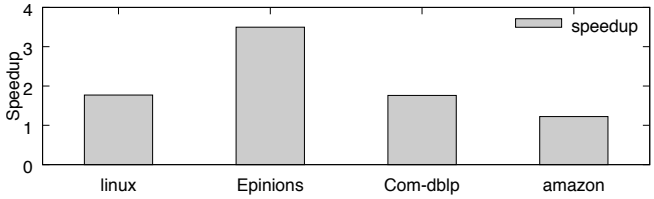

Figure 6: Speedup of decoupling data access and computing.

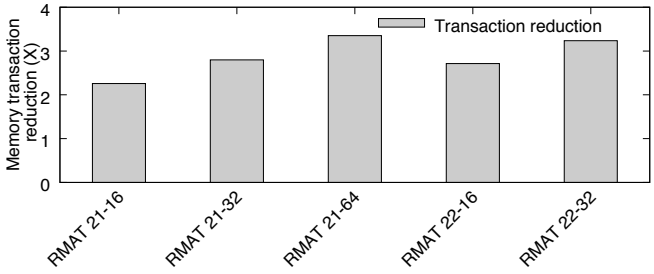

Figure 7: Reduction in memory transaction from coalesced graph data access.

coalescing disabled. Coalescing memory accesses results in up to a 3.4× reduction in total memory requests, 2.3× at minimum, and 2.9× on average. Fewer memory transactions means that the load on the memory interface is reduced, and that more transactions can occur in the sequential burst mode. This means that coalescing the requests for neighbors of adjacent frontiers can yield better performance in the metadata update step.

Figure 5 studies the benefits of using a bigger on-chip memory over a smaller one, resembling the benefits of bit over byte addressable mechanism. We synthesized our design with a vertex cache of two sizes, 1Mb and 2Mb, and observed that the 2Mb cache always yielded higher performance. We see a 4.1× speedup on average using the larger 2Mb cache, with 5.6× maximum and 2.6× minimum. We find that expanding the size of our cache yields better performance in all graphs, because the ability to hold more of the graph on chip at a given time means that we avoid having to use high-latency external memory to fetch other partitions.

Figure 6 depicts the speedup of the design with separate metadata and data with a design that does not separate data and metadata. To test this, we use one of our own designs that can only support smaller graphs, so we use a suite of smaller graphs to compare. Our current design achieves up to a 3.5× speedup, 1.2× at minimum, and 2.0× on average. This is largely due to the lower number of memory accesses in our current design. Note, these datasets are downloaded from https://snap.stanford.edu/data/.

6 CONCLUSION

In this paper, we have proposed a novel data-centric hardware accelerator for BFS that uses an off-chip status array and tight external memory access control to reduce contention in its status updates. Compared with CPU designs and state-of-the-art FPGA designs, we show that our hardware-software codesigned model can achieve 4.1× speedup over a CPU design, up to 2.1× over [10] and up to 1.6× over [9].

ACKNOWLEDGMENT

We thank the anonymous reviewers for their comments. This work is supported in part by the US National Science Foundation No. 1547428, No. 1541434, No. 1738965, No. 1450996 and CRII Award No. 1850274.

REFERENCES

[1] Anil Gaihre, et al. Do bitcoin users really care about anonymity? an analysis of the bitcoin transaction graph. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1198–1207. IEEE, 2018.
[2] Brahim Betkaoui, et al. A framework for FPGA acceleration of large graph problems: Graphlet counting case study. In *2011 International Conference on Field-Programmable Technology*, pages 1–8. IEEE.
[3] Nenad Trinajstic. *Chemical graph theory*. Routledge, 2018.
[4] Richard C Murphy, et al. Introducing the graph 500. *Cray User's Group (CUG)*, 19:45–74, 2010.
[5] Duane Merrill, et al. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.
[6] H. Liu et al. Enterprise: breadth-first graph traversal on GPUs. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12.
[7] Shijie Zhou et al. Accelerating graph analytics on CPU-FPGA heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE.
[8] Shijie Zhou, et al. An FPGA framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers - CF '18*, pages 69–77. ACM Press.
[9] Guoqing LEI, et al. TorusBFS: A novel message-passing parallel breadth-first search architecture on FPGAs. *IRACST International Journal*, 5(5):6.
[10] Y. Umuroglu, et al. Hybrid breadth-first search on a single-chip FPGA-CPU heterogeneous platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8.
[11] Michael Bauer, et al. Cudadma: optimizing gpu memory bandwidth via warp specialization. In *SC*, 2011.
[12] Yangzihao Wang, et al. Gunrock: Gpu graph analytics. *ACM Transactions on Parallel Computing (TOPC)*, 4(1):3, 2017.
[13] Reynold S Xin, et al. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.
[14] Alexander Frolov, et al. Performance evaluation of breadth-first search on intel xeon phi. page 12.
[15] Mireya Paredes, et al. Breadth first search vectorization on the intel xeon phi. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 1–10. ACM, 2016.
[16] Shijie Zhou, et al. High-throughput and energy-efficient graph processing on FPGA. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 103–110. IEEE.
[17] Grzegorz Malewicz, et al. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
[18] Amitabha Roy, et al. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.
[19] Charles Eric LaForest et al. Efficient multi-ported memories for FPGAs. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '10*, page 41. ACM Press.
[20] O. G. Attia, et al. CyGraph: A reconfigurable architecture for parallel breadth-first search. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 228–235.
[21] Jialiang Zhang, et al. Boosting the performance of FPGA-based graph processor using hybrid memory cube: A case for breadth first search.
[22] Maohua Zhu, et al. Performance evaluation and optimization of hbm-enabled gpu for data-intensive applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(5):831–840, 2018.
[23] Hang Liu et al. Graphene: fine-grained io management for graph computing. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, pages 285–299. USENIX Association, 2017.
[24] Hao Wei, et al. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828. ACM, 2016.
[25] NVIDIA TESLA P100 GPU, https://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-pcie-datasheet.pdf.