2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search

Osama G. Attia Tyler Johnson Kevin Townsend Philip Jones Joseph Zambreno
Department Electrical and Computer Engineering
Iowa State University
Ames, IA 50011 USA
{ogamal, tyler07, ktown, phjones, zambreno}@iastate.edu

***Abstract*—Large-scale graph structures are considered as a keystone for many emerging high-performance computing applications in which Breadth-First Search (BFS) is an important building block. For such graph structures, BFS operations tends to be memory-bound rather than compute-bound. In this paper, we present an efficient reconfigurable architecture for parallel BFS that adopts new optimizations for utilizing memory bandwidth. Our architecture adopts a custom graph representation based on compressed-sparse raw format (CSR), as well as a restructuring of the conventional BFS algorithm. By taking maximum advantage of available memory bandwidth, our architecture continuously keeps our processing elements active. Using a commercial high-performance reconfigurable computing system (the Convey HC-2), our results demonstrate a 5× speedup over previously published FPGA-based implementations.**

***Index Terms*—Reconfigurable Computing, Breadth-First Search, Graphs, FPGA, Convey HC-2.**

I. Introduction

Many of today's large data-intensive scientific problems are solvable through graph theoretical approaches. Examples can be found in diverse domains including bioinformatics [1], artificial intelligence, and social networking [2]. For example, several methods of short read genome assembly rely on solving overlap layout consensus graphs or De Bruijn graphs [3]. It has been shown in the research literature that FPGA-based platforms can perform efficiently in solving such large-scale problems [4], [5].

Many previous researchers have investigated the goal of improving graph processing, with the most common theme being parallelization in some form of software design. Approaches using MPI, OpenMPI and massively multi-threaded architectures such as Cray have all been used to improve performance [6]. However, many of these approaches do not attack the problem at the source, which is the memory-bound nature of processing large graphs. Approaches using distributed and shared memory models already suffer due to relatively long memory latencies, but also can not easily amortize latency costs with random access patterns. Graph representation plays an important role in overall performance, with finding ways to provide bounds on the memory usage being one important aspect. Being able to easily and efficiently proceed to the next vertex is also important, since if this is not done properly, any improvements with regards to memory latency will be lost if time is wasted on collecting the next vertex's information.

One important building block of graph algorithms is the Breadth-First Search (BFS). BFS is used in various scientific problems that require high-performance approaches to traverse large-scale graphs. In creating a parallel approach for BFS, one challenge is that the individual operations will need to be synchronized among themselves, which could potentially cause pipeline stalls and other implementation inefficiencies. Thus, it is imperative that design tactics are taken to provide parallelization with as few data dependencies as possible. In the case of a multi-FPGA system (such as the Convey HC-2 [5] we target in this paper), this parallelization can be spread across a single FPGA and also simultaneously spread across multiple FPGAs. As such, inter-device synchronization mechanisms are also needed, which as a trade-off adds some latency.

In this paper, we introduce CyGraph, a reconfigurable architecture for parallel BFS. The main contributions for our approach are the following:

- We introduce an application-specific graph representation based on the Compressed-Sparse Row format (CSR) that appropriately matches the BFS algorithm.
- We develop a restructured and optimized version of BFS algorithm that makes use of this new graph representation. Theoretically, these optimizations cuts down the number of external memory requests by least 50%.
- Our CyGraph architecture is highly scalable when mapped to FPGA-based systems.
- We describe how our architecture scales across multiple FPGAs, and in doing so is performance competitive with an existing state-of-the-art BFS implementation on the same platform.

The remainder of this paper is organized as follows. In Section II, we introduce related work. Section III introduces background about breadth-first search algorithms and graph representation. In Section IV, we show the building block for paralleling BFS (the CyGraph kernel). Afterwards, we propose the hardware implementation of CyGraph in Section V. Performance results and insights are presented and discussed in Section VI. Finally, conclusions are drawn and potential directions for future work are pointed out in Section VII.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.30

II. Related Work

A huge need for higher performance and power efficient graph algorithms has developed with the growth of scientific problems that require dealing with large datasets [6]. Many attempts in the literature have aimed to accelerate graph algorithms through innovative programming models and dedicated hardware platforms. Attempted implementations include the usage of commodity processors, multi-core systems, GPUs, and FPGAs.

For multi-core implementations, a parallelized BFS algorithm for multi-core architectures was described by Agarwal et al. [7]. They use a bitmap to mark nodes that have been visited, and demonstrated a speedup over previous work. For implementations using large distributed memory machines, Beamer et al. have shown significant speedup over other implementations [8], [9]. Note that such implementations are not directly comparable to ours, given the high cost and power consumption of conventional supercomputers.

GPUs have been adopted for speeding up computations in a variety of applications, including graph processing. Hong et al. [10], [11] proposed a level-synchronous BFS kernel for GPUs. They showed improved performance over previous work. Merril et al. [12] used a prefix sum approach for cooperative allocation, leading to improved performance. In [13], the authors present a GPU programming framework for improving GPU-based graph processing algorithms.

For FPGA-implementations, previous work accommodated on-chip memory in order to store graphs [14]. However, this is not suitable for large-scale problems since these designs do not consider the high-latency of off-chip memory. Recent FPGA-based platforms, such as the Convey HC-1/HC-2 [5], [15], are known for their high memory bandwidth and potential for application acceleration. For example, the Convey HC-1/HC-2 platforms have been shown to be beneficial in speeding up various computation-intensive and memory-bound applications [16]–[19].

The closest previous approach to our work can be found in Betkaoui et al. [20], which introduces a parallel graph exploration algorithm using the Convey HC-1 platform. The authors achieved a $2\times$ improvement over the state-of-the-art GPU and CPU implementations. This revealed a capability of reconfigurable computing platforms to outperform these of GPU/CPU. In Section VI, we show that through a novel restructuring and optimization of the BFS algorithm, our approach obtains higher memory utilization than in [20] resulting in a $5\times$ relative speedup.

More recent BFS implementations by Convey using their latest HC-2ex and MX-100 systems [21] illustrate the importance of this application kernel to the Graph500 supercomputer rankings [22].

III. Background

BFS is a graph traversal algorithm that visits all the connected nodes in a specific graph starting at given root node. The algorithm visits the nodes that have smaller hop distance from the root node first. The nodes could be processed while traversing the graph, or the order of visiting nodes could be saved for post-processing. BFS can be used to solve many problems that utilize a graph theoretical approach.

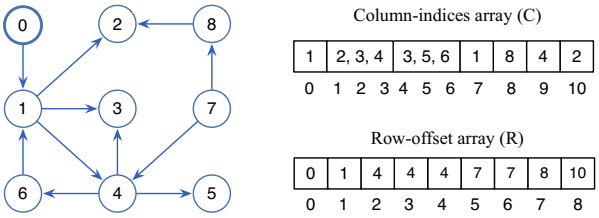


Fig. 1: Simple example for CSR graph representation format

A. Graph Representation

One simple way to represent a graph of n nodes is by using an adjacency matrix of size $n \times n$ whose rows are the adjacency list of graph nodes. As this can be inefficient, most of the state-of-the-art BFS implementations use the Compressed Sparse Row (CSR) format for representing graphs. Since it only stores the non-zero values of the adjacency matrix, CSR is commonly used to represent sparse matrices/graphs. Figure 1 shows a simple example of the CSR representation, which consists of two vectors:

- Column Indices Array (C): contains the nodes adjacency list, with size is bounded by the number of non-zero edges.
- Row Offset Array (R): contains the indices at which the adjacency list of each node starts.

B. Level-Synchronous BFS

One commonly used approach towards a parallel breadth-first search is level-synchronous BFS. The level-synchronous BFS algorithm uses three sets of nodes as follows:

- Set of current nodes (Q_c)
- Set of next nodes (Q_n)
- Visited nodes (V)

The algorithm starts with the root node in the current set. Iteratively, the algorithm visits all the nodes in Q_c, sets their level as the current level and collects the next-level set Q_n. In the next iteration, Q_c is populated with values from Q_n, and Q_n is cleared for the new level.

This approach is illustrated in Algorithm 1. Q_c and Q_n refer to FIFO queues in memory. The vector *Levels* stores the final result of the algorithm and is initialized with zero. Instead of using the V set, the algorithm checks the *Levels* value of a specific node (i.e. node i is not visited if *Levels[i]* equals 0).

A major challenge in designing large-scale graph processing algorithms is utilizing memory bandwidth. Increasing system throughput requires that data is kept available for immediate processing. In the previous algorithm, there are six memory read requests (lines 6, 7, 10, 11, 13 and 14) and two memory write requests (lines 9, and 16). In the following section, we

propose a custom graph representation and optimization that results in fewer memory requests.

Algorithm 1: Level-Synchronous BFS Algorithm

Input: Arrays R, C holding graph data, and *root* node
Output: Array *Levels* holding traversal order of nodes

```
1  Q_n.push(root)
2  current_level ← 1
3  while not Q_n.empty() do
4      Q_c ← Q_n
5      while not Q_c.empty() do
6          v ← Q_c.pop()
7          level ← Levels[v]
8          if level = 0 then
9              Levels[v] ← current_level
10             i ← R[v]
11             j ← R[v + 1]
12             for i < j; i ← i + 1 do
13                 u ← C[i]
14                 level ← Levels[u]
15                 if level = 0 then
16                     Q_n.push(u)
17                 end
18             end
19         end
20     end
21     current_level ← current_level + 1
22 end
```

IV. BFS Optimization

Our CyGraph approach is based on the previously-described level-synchronous BFS algorithm, and targets high performance reconfigurable computing platforms with relatively large amounts of memory bandwidth. Specifically, we propose a custom graph representation that we use later in our BFS implementation to utilize memory bandwidth. We also explore efficiency optimizations to the BFS algorithm that reduces the number of memory requests and hence increases overall system throughput.

A. Custom Graph Representation

From Algorithm 1, we notice that for each node i in the current set, we have to read the level and check if the node is visited or not. Then, if the node is not visited we have to read the row offset array ($R[i]$ and $R[i+1]$) in order to find the start and end addresses of this node's adjacency list. Also, it is clear that we need the *Row Index Array (R)* only once per node. Hence, after reading and visiting we can use its row index value for storing the level data. We show later how this optimization will lower the number of memory requests in the algorithm.

The Convey HC-2 platform, as with many state-of-the-art computing platforms, is capable of fetching 64-bit values per memory request. Consequently, we modified the width of the row offset array (R) in the CSR representation to be an array of elements of 64-bit width. For each element, $R[i]$, we will use the least significant bit of the row index as a *visited flag* (*i.e.* node i is not visited, if the LSB of $R[i]$ is 0). The remaining bits of $R[i]$ are used as follows:

- If node i is visited, the LSB of $R[i]$ will equal 0 and the remaining bits will be used for node level.
- If node i is not visited, the MSB bits from 63 to 32 will be used as row index and the bits from 31 to 1 as neighbors count.

Figure 3 illustrates two entries of the new offset row array in case of visited and unvisited nodes.

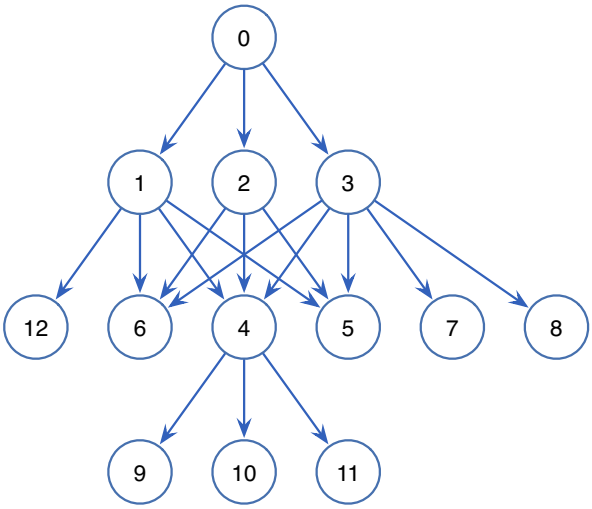


Fig. 2: Graph example, on which Algorithm 1 performs poorly.

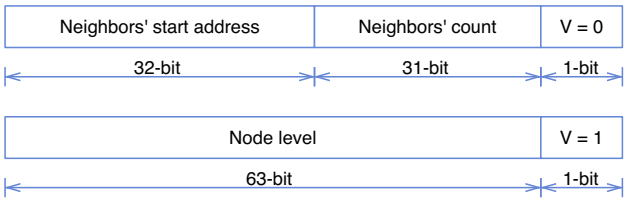


Fig. 3: Custom CSR format

B. Algorithm Optimizations

We consider the following optimizations to the traditional level-synchronous BFS algorithm:

- **Current/next queues of row indices:** In every iteration, we just need the row index of a specific node. Consequently, a better decision is to push that row index of node $R[i]$ instead of node ID to next queue. Hence, we will make use of the data we fetched in a previous iteration.
- **Rearranging instructions:** A parallel version of Algorithm 1 will perform poorly in the graph shown in

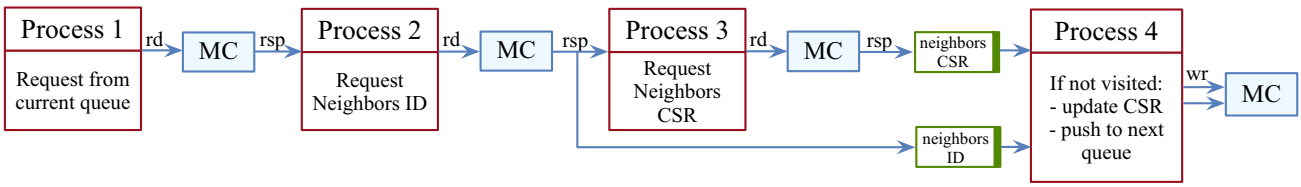


Fig. 4: Simple pipelined version of CyGraph kernel

Fig. 2. After the third level, we will have duplicates of nodes $4, 5$, and 6 in the next queue. The main idea of the optimization is that we first visit the neighbors of the current node and push them to the next queue (the previous approach was to visit the current node and push its neigbors to the next queue). Furthermore, even if duplicate nodes are pushed to Q_n, the possibility to have duplicate of their neighbors in the future is very low (parallel kernels have to be in a perfect synchronization which is not possible with random memory access).

- **Custom graph representation:** As mentioned in the previous subsection, the custom graph representation will allow us to find out if a node is visited or not and the indices of its adjacency in just one memory request. Thus, cutting down the memory requests to the half.

Algorithm 2 shows the CyGraph BFS after optimizations are applied. We notice that we decreased the number of memory requests to five requests instead of eight (three read requests at lines 8, 12, 13 and two write requests at lines 15, 16). Also, the optimization reduces the possibility of having duplicates in the current/next queue which increases system throughput.

Algorithm 2: BFS using custom CSR

Input: Arrays R, C holding graph data, and *root* node
Output: Array *Levels* holding traversal order of nodes

```
1  csr ← R[root]
2  Q_n.push(csr)
3  R[root] ← b'11
4  current_level = 1
5  while not Q_n.empty() do
6      Q_c ← Q_n
7      while not Q_c.empty() do
8          v_csr ← Q_c.pop()
9          i ← v_csr[63..32]
10         j ← v_csr[31..1]
11         for i < j; i ← i + 1 do
12             u ← C[i]
13             u_csr ← R[u]
14             if u_csr[0] = 0 then
15                 Q_n.push(u_csr)
16                 R[i] ← (current_level + 1) & b'1
17             end
18         end
19     end
20     current_level ← current_level + 1
21 end
```

V. Our Approach

In this section, we present the methodology that we used to implement the CyGraph kernel and parallelize it. Then, we show how the kernels will manage to share access to the same memory space (current/next queues) efficiently.

A. Kernel Architecture

The CyGraph kernel serves as a building block for our implementation. As shown in Fig. 4, we designed the CyGraph kernel as a pipeline of four stages. Pipeline stages are separated by a FIFO queues. However, in order to optimize the memory controllers and keep them busy with requests, we are multiplexing all kernels requests to a single memory controller. Figure 5 shows the optimized kernel architecture.

The main target of our system is to make the best effort to utilize memory bandwidth and hence making data available for the high computational power of the FPGA. So, the design methodology we used to approach the problem is splitting the algorithm into small processes/stages that are separated by memory requests. Each process will be continuously reading previous process memory response and making new requests to the memory controller.

In our kernel design, process 1 is responsible for dequeuing nodes' information from the current queue. Each kernel reads from the current queue with an offset plus *kernel_id* (i.e. kernels interleave reading from the current queue). Process 1's response is pushed to the FIFO queue q_1. Concurrently, process 2 is responsible for reading the current queue nodes that are fetched by process 1 and requests their neighbors' ID. The response of process 2 is available immediately to process 3 and a copy to process 4 gets pushed through the FIFO queue q_2. Process 3 gets the neighbors' ID and requests their CSR information which will be available to process 4 through q_3. Finally, process 4 checks the CSR information of each neighbor. If a neighbor is not visited, process 4 issues two requests:

- Enqueue neighbor's information (CSR) to the next queue.
- Update the neighbor's information with the current level.

It is important to note that all kernel processes run in parallel. The requests multiplexer is designed to exploit the memory controller bandwidth by keeping it busy every clock cycle either for writing or reading. The requests multiplexer uses a priority scheduling algorithm to check the queues. Process 4's requests are the highest priority and process 1 is

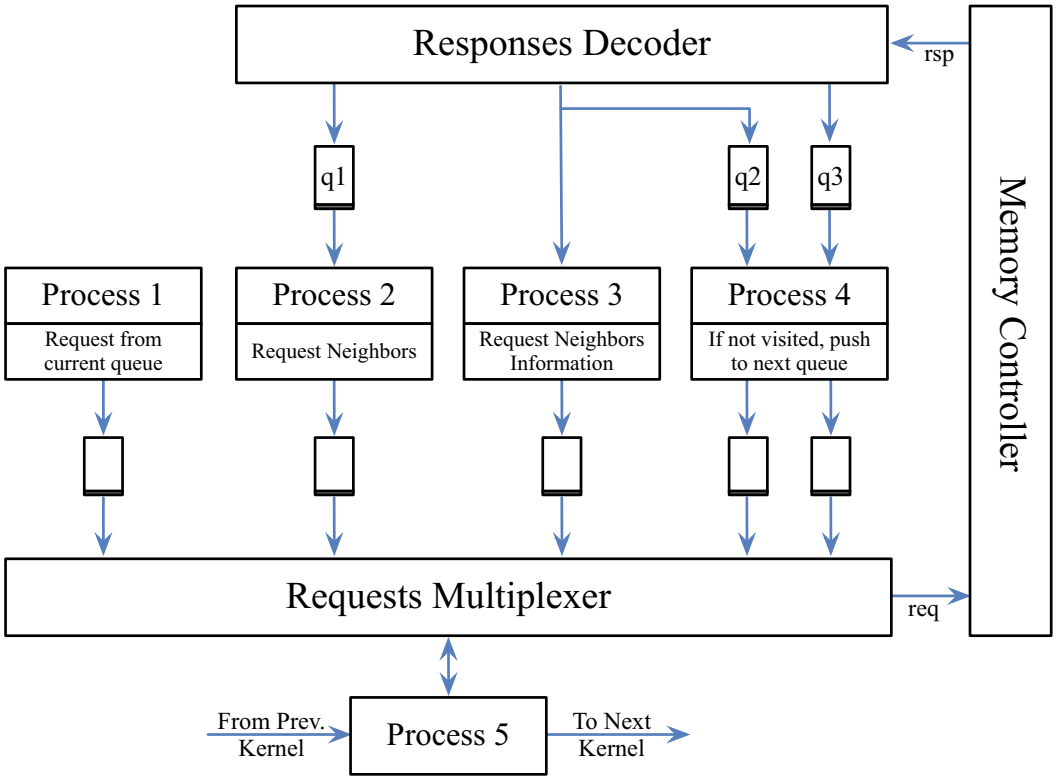


Fig. 5: CyGraph Kernel Architecture

the lowest priority. This heuristic is chosen based on the fact that resolving the requests of process 1 will lead to more data in the other queues. The multiplexer tags each read request with the owner process's ID. When the response decoder gets a new response, it will check the tag associated with it and consequently it will know to which process it does belong.

In order to find out when the kernel finishes processing, every process counts all items it reads and writes. Kernel processes share these counts among themselves. Each process is finished if the preceding process to it is done and when it processes as many items as the previous process. The kernel finishes when all of its processes are terminated successfully and all the FIFO queues are empty. Finally, a kernel must issue a flush request to the memory controller it is using in order to make sure that all the memory requests made are completed. The count of the nodes pushed to the next queue are carried out to next level.

B. Kernel-to-Kernel Interface

One problem that this design faces is to allow multiple reads and writes to the same memory-based queue. Different kernels have to read and write in parallel to the current and next queues. In the case of reading from the current queue, this problem is very intuitive and is solved by interleaving and splitting the current queue among different kernels. However, in the case of writing to the next queue, kernels don't know exactly how many neighbors will be written by each kernel.

In order to solve this problem, we designed our platform as a ring network as shown in Fig. 6. The kernel-to-kernel interface behaves very similar to the familiar token ring LAN protocol. A token circulates continuously through the kernels every clock cycle. The token tells each kernel how many neighbors the previous kernel will be writing to the next queue. The kernel-to-kernel interface works as follows:

- The master kernel initiate the interface by sending an empty token to the first kernel in the ring.
- When the token starts circulating, each kernel gets it for one cycle and has to pass it to the next kernel.
- Each kernel keeps counting how many nodes that needs to write them to next queue. We call this the *demand*.
- If a kernel gets the token, it should reserve its demand, pass the token including information about what previous kernel reserved, and reset the demand count.
- If a kernel does not need to reserve any more space when it gets the token, it should pass the exact same token as the previous kernel.
- When a level is done, the master kernel will be able to know how many items were written to the next queue from the last token.
- The token is reset at the beginning of every level.

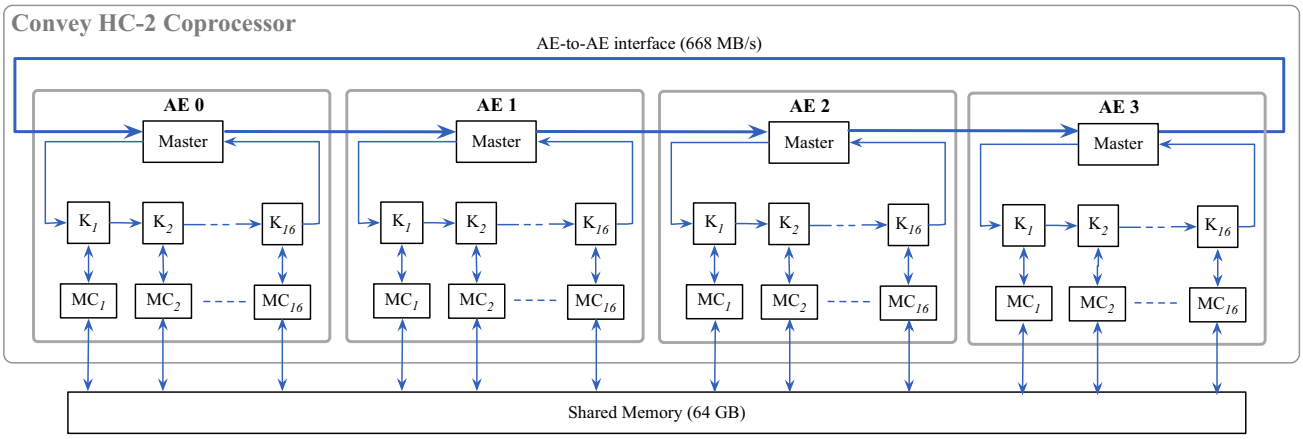


Fig. 6: CyGraph System Architecture

VI. Implementation Results

We ported our CyGraph BFS architecture to the Convey HC-2 platform [5]. The Convey HC-2 is a heterogeneous high performance reconfigurable computing platform that consists of a coprocessor board of four programmable Virtex5-LX330 FPGAs. The Convey HC-2 programmable FPGAs are called *Application Engines* (AEs) and the custom FPGA configurations are called *personalities*. Each of the four AEs is connected through a full crossbar to 8 on-board memory controllers, with each memory controller providing two request ports (i.e. 16 memory ports per AE). The full crossbar allows the four AEs to access the memory at a peak performance of 80 GB/s (i.e. each AE can access memory at peak bandwidth of 20 GB/s). Furthermore, the AEs are connected together through a full-duplex 668 MBps ring connection, referred to as the AE-to-AE interface, which we are utilizing to extend our kernel-to-kernel communication between multiple FPGAs. Also, it is important to state that the coprocessor memory is fully coherent with the host processor memory. The coprocessor board is connected with the host processor through a PCI Express channel.

Figure 6 shows our CyGraph system architecture for multiple FPGAs. The master processing element is responsible for triggering and managing the flow of processing inside each FPGA. Also, it is used to maintain the advancement of each level and other synchronization requirements. Our custom implementation on the Convey HC-2 uses 16 kernels per AE (i.e. 64 kernels in the whole design). Each kernel utilizes one memory port as mentioned in the previous section. We use the AE-to-AE interface to link the kernels among the different FPGAs as shown in Fig. 6. CyGraph starts from the host processor which will process and copy the graph data to memory and then it will invoke the CyGraph coprocessor via a custom instruction. Our implementation is fully written in VHDL code.

Experimental Results

As in previous related work, we have generated random and R-MAT graph data using GT-graph, a suite of synthetic graph generators [23]. Figures 8, 7, and 9 demonstrate how CyGraph performs against the BFS implementation from Betkaoui et al. [20]. Both of the implementations target the same platform, the Convey HC-1/HC-2. We compare our implementation using 1 and 4 Application Engines (AEs) against their implementation using 4 AEs. The comparison shows throughput in billions of edges per second and uses graphs with number of nodes that spans from 2^{20} to 2^{23} and average vertex degrees that spans from 8 to 64.

Figures 8a and 8b show that CyGraph maintains a speedup over the Betkaoui et al. implementation [20] for graphs of average vertex degree 8 and 16. However, the two systems show relatively close performance for higher average vertex degrees. The lower performance of [20] for smaller average vertex degrees is due to the fewer number of updates it will be performing on every iteration. However for larger vertex degrees, their implementation tends to loop over the large number of vertices sequentially, updating larger number of values per iteration and thus achieving higher eventual throughput. Figure 9 summarizes the previous results to show the effect of average vertex degree on the execution time. It shows that as graphs get more sparse, our implementation obtains higher speedups over [20] and comes quite close to its peak performance.

Resource Utilization

Table I shows the device resource utilization. We note that our CyGraph resource utilization does not increase significantly from 1 AE to 4 AEs since it replicates the same design for each FPGA, plus a few more gates to control the AE-to-AE interface. Also, our CyGraph kernel leaves relatively more resources available which could be employed to add more customizations and computations.

In Convey HC-2 platform, about 10% of the FPGAs' logic and 25% of the block rams are occupied for the required

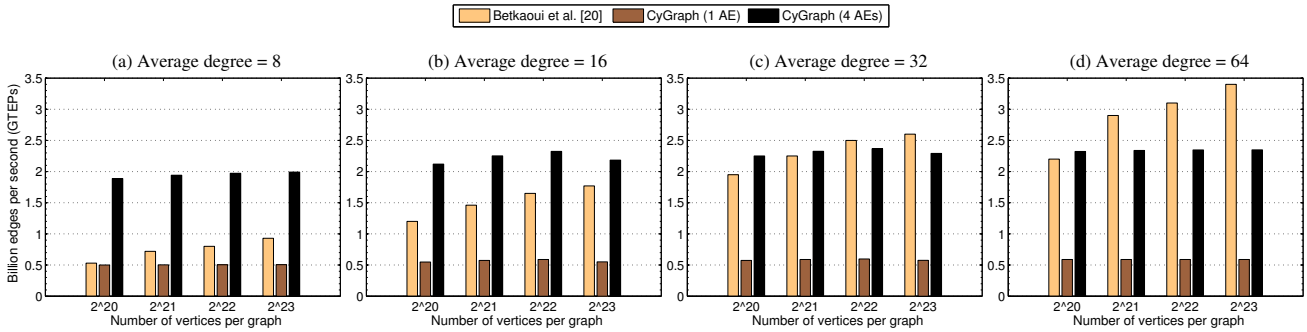


Fig. 7: BFS execution time for CyGraph against [20] using random graphs

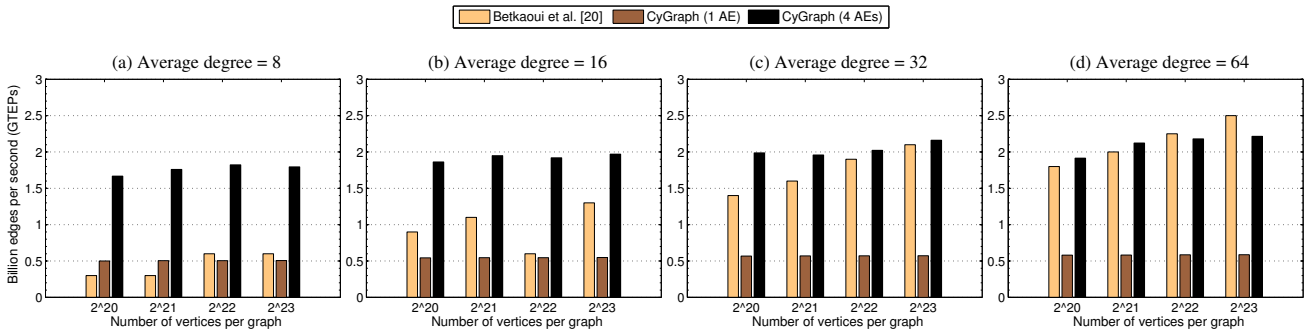


Fig. 8: BFS execution time for CyGraph against [20] using RMAT graphs

interfaces (e.g. dispatch interface, and MC interface). For our CyGraph kernel, the depth of the FIFOs used is set to 512.

	Slice LUTs	**BRAM**	**Slice LUT-FF**
CyGraph 1 AE	53%	55%	74%
CyGraph 4 AEs	55%	55%	74%
Betkaui et al. [20]	80%	64%	n/a

TABLE I: Resource Utilization

VII. Conclusion and Future Work

In this paper, we have proposed a novel FPGA-based graph traversal implementation, namely CyGraph. Our implementation outperformed a state-of-the-art FPGA implementation up to factor of 5. CyGraph performed better for graphs of medium average vertex degrees. We have shown that application-specific data structures significantly improve performance for the target platform. We have introduced a kernel-to-kernel interface that enables multiple processing elements to collaborate writing to the same memory data-structure. Finally, we have followed a design approach that could be carried out to design other hardware reconfigurable architectures for graph processing algorithms.

The suggested future work is twofold. The first direction is to enhance our implementation by load balancing among kernels. This will keep all CyGraph kernels busy for equal processing times, resulting in a better utilization of memory bandwidth. One idea is to split the large CSRs into multiple CSRs before pushing them to the next queue. In the next level, due to the fact that kernels interleave reading, kernels will have to process a relatively equal number of neighbors. The second direction is to develop a full framework for graph algorithms using the Convey HC-2. Other graph problems (e.g. shortest path, Eulerian path, maximum flow, maximum clique) are known to be difficult problems. Such a framework can serve as a base and useful tool for researchers to build upon it.

Acknowledgements

This work was supported in part by the National Science Foundation (NSF) under awards CNS-1116810 and CCF-1149539.

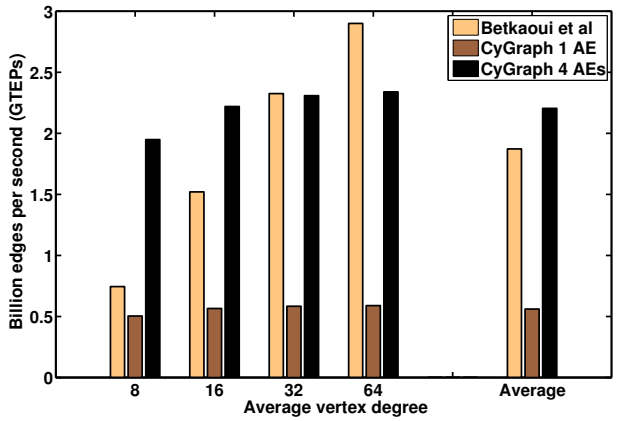
References

[1] O. Mason and M. Verwoerd, "Graph theory and networks in biology," in *IET Systems Biology*, vol. 1, no. 2, 2007, pp. 89–119.

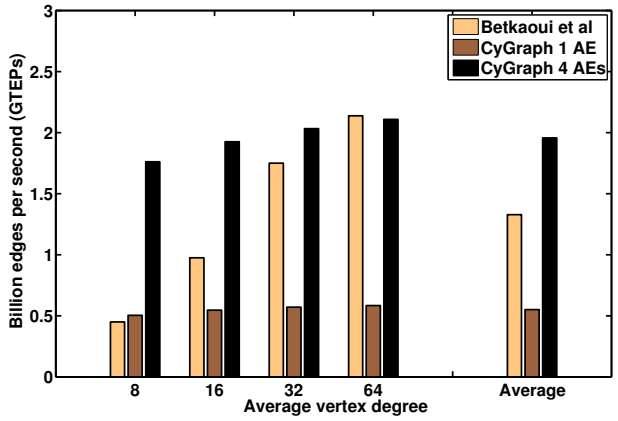
[2] M. Gjoka, M. Kurant, C. T. Butts, and A. Markopoulou, "Walking in Facebook: A case study of unbiased sampling of OSNs," in *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2010.

[3] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," in *Genome Research*, vol. 18, no. 5, 2008, pp. 821–829.

[4] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and W. L. Ruzzo, "Hardware acceleration of short read mapping," in *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2012, pp. 161–168.



(a) Random Graphs



(b) R-MAT Graphs

Fig. 9: Average vertex degree effect on CyGraph execution

[5] J. D. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1," in *Computing in Science & Engineering*, vol. 12, no. 6, Dec. 2010, pp. 80–87.

[6] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," in *Parallel Processing Letters*, vol. 17, no. 01, 2007, pp. 5–20.

[7] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable Graph Exploration on Multicore Processors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[8] S. Beamer, A. Buluç, K. Asanovic, and D. Patterson, "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search," *Technical Report UCB/EECS-2013-2, University of California*, 2013.

[9] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[10] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 267–276.

[11] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core CPU and GPU," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 78–88.

[12] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU Graph Traversal," in *Proceedings of the Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, pp. 117–128.

[13] J. Zhong and B. He, "Medusa: Simplified Graph Processing on GPUs," in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 99, 2013.

[14] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multi-softcore architecture on FPGA for Breadth-first Search," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, Dec. 2010, pp. 70–77.

[15] T. M. Brewer, "Instruction set innovations for the Convey HC-1 computer," *IEEE Micro*, vol. 30, no. 2, pp. 70–79, 2010.

[16] K. K. Nagar and J. D. Bakos, "A Sparse Matrix Personality for the Convey HC-1," in *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011.

[17] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "Parallel FPGA-based all pairs shortest paths for sparse networks: A human brain connectome case study," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 99–104.

[18] K. Townsend and J. Zambreno, "Reduce, Reuse, Recycle (R^3): a Design Methodology for Sparse Matrix Vector Multiplication on Reconfigurable Platforms," in *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jun. 2013.

[19] W. Augustin, J. Weiss, and V. Heuveline, "Convey HC-1 hybrid core computerthe potential of FPGAs in numerical simulation," in *Proc. Int. Workshop on New Frontiers in High Performance and Hardware-aware Computing (HipHaC)*, 2011.

[20] B. Betkaoui, Y. Wang, D. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, 2012, pp. 8–15.

[21] K. Wadleigh, J. Amelio, K. Collins, and G. Edwards, "Hybrid Breadth First Search Implementation for Hybrid-Core Computers," in *SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, 2012, pp. 1354–1354.

[22] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," in *Cray Users Group (CUG)*, 2010.

[23] D. Bader A. and K. Madduri, "GTgraph: A suite of synthetic graph generators," 2006. [Online]. Available: www.cse.psu.edu/~madduri/software/GTgraph