Accelerating All-Pairs Shortest Path Using A Message-Passing Reconfigurable Architecture

Osama G. Attia, Alex Grieve, Kevin R. Townsend, Phillip Jones, and Joseph Zambreno
Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, USA 50010
Email: {ogamal, argrieve, ktown, phjones, zambreno}@iastate.edu

***Abstract*—In this paper, we study the design and implementation of a reconfigurable architecture for graph processing algorithms. The architecture uses a message-passing model targeting shared-memory multi-FPGA platforms. We take advantage of our architecture to showcase a parallel implementation of the all-pairs shortest path algorithm (APSP) for unweighted directed graphs. Our APSP implementation adopts a fine-grain processing methodology while attempting to minimize communication and synchronization overhead. Our design utilizes 64 parallel kernels for vertex-centric processing. We evaluate a prototype of our system on a Convey HC-2 high performance computing platform, in which our performance results demonstrates an average speedup of $7.9\times$ over the sequential APSP algorithm and an average speedup of $2.38\times$ over a multi-core implementation.**

***Index Terms*— all-pairs shortest path, Convey HC-2, FPGA applications, graphs, reconfigurable computing**

I. Introduction

Graph data structures have been used widely by researchers from many fields to facilitate representing, analyzing, and drawing conclusions in many real-world scientific applications [1]. However, with the on-going increase in graph sizes, the existing graph processing approaches have become challenged. There is a need for novel approaches to process these large-scale graphs while attaining a full utilization of the available computation resources.

In the last decade, a significant amount of research has been conducted to investigate graph processing frameworks using commodity multi-core processors, distributed systems, and GPGPUs. Due to the poor locality and memory-bound nature of graph processing algorithms, software implementations tend to suffer from high memory latency that is difficult to avoid [2]. Furthermore, a significant overhead for communication and synchronization is imposed in order to parallelize these algorithms. Previous research has shown that FPGA-based acceleration can provide low power and highly efficient solutions for a wide range of application domains, while also decreasing the amount of synchronization and communication overhead [3], [4].

An important and fundamental algorithm in graph analysis is the All-Pairs Shortest Path algorithm (APSP), which has many real-world applications including VLSI routing, bio-informatics, social network analysis, and communications [5]. Given a directed weighted graph, APSP is the problem of finding the shortest distance between every two nodes in the graph. This problem reduces to multiple breadth-first search (BFS) traversals in the case of an unweighted directed graph.

In this paper, we present a message-passing reconfigurable architecture that is suitable for vertex-centric graph algorithms. The main contributions for this paper are the following:

- We describe a scalable message-passing architecture inspired by the bulk-synchronous parallel model.
- We present a novel implementation for arranging processing elements in multi-FPGA platforms and allowing synchronization without degradation in performance.
- We implement a multi-step All-Pairs Shortest Path (APSP) algorithm for unweighted directed graphs based on level synchronous breadth-first search (BFS).
- We introduce a hardware implementation for the APSP kernel illustrating specific optimizations to hide the large memory footprint and amortize memory latency.
- We showcase a hardware prototype of our architecture and APSP implementation using a commercial high performance computing platform (Convey HC-2).
- We evaluate the competitiveness of our implementation against single-core sequential, and multi-core parallel implementations.

The remainder of this paper is organized as follows. We survey related work in Section II. In Section III, we introduce our reconfigurable architecture, and provide underlying system assumptions. Section IV goes through the baseline APSP algorithm and introduces our multi-step parallel APSP algorithm. Then, we show the implementation of the APSP kernels and optimization techniques 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.

II. Related Work

There have been numerous successful attempts in the literature to design graph libraries targeting single-core commodity machines. With the increased difficulty of speeding up the operational frequency of general-purpose processors and the non-stop expansion in data size [6], many researchers started investigating alternative parallel solutions. In [7], Malewicz et al. introduced Pregel, a distributed message-passing system for graph processing using a vertex-centric computation model inspired by the Bulk-Synchronous Parallel model (BSP). Apache Giraph [8] and GPS [9] followed, providing alternative open source projects to Pregel. In [10], the authors presented GraphLab, an asynchronous parallel graph processing engine

978-1-4673-9406-2/15/$31.00

for shared memory systems and targeting common patterns in machine learning problems.

Many researchers have attempted to accelerate individual graph algorithms using reconfigurable platforms. For instance, Betkaoui et al. investigated the design of large-scale graph processing algorithms on multi-FPGA systems including BFS [11], APSP [12], and Graphlets counting [13]. Bondhugula et al. showed a parallel FPGA-based design for weighted APSP algorithm in [14]. The authors in [15] presented a parallel FPGA-based implementation for a frequent subgraph mining algorithm using a Convey HC-1 machine. In our previous work [16], we proposed an architecture for parallel BFS traversals on multi-FPGA systems.

For hardware accelerated graph processing architectures, the authors in [17], [18] presented GraphGen, a compiler for single FPGA-based vertex centric algorithms. However, the authors did not discuss the scalability of their platform across multiple FPGAs with shared memory subsystem. Ceriani et al. proposed a distributed multi-core architecutre using FPGAs for irregular memory access patterns in [19]. The architecture uses 4 FPGAs to implement multiple Xilinx MicroBlaze softcore general-purpose processors. These previous attempts show the benefits of FPGA-based architectures in tackling graph algorithms and motivates the investigation of new approaches to design a scalable reconfigurable computing platform for graph processing.

III. Graph Processing Architecture

We consider multi-FPGA platforms with a shared memory system where each FPGA has access to m memory controller ports. Also, we assume that the FPGAs are connected in a ring topology through I/O ports. The goal of our architecture is to provide a parallel processing elements arrangement such that we maximally saturate either of the computation resources or the memory bandwidth resources of each FPGA.

A. Computational Model

The computational model of our architecture is similar to that introduced in the Pregel framework for distributed systems [7], which is based on the *Bulk Synchronous Parallel (BSP)* model [20]. A typical BSP model splits computations among a sequence of *supersteps* separated by synchronization barriers. In each superstep, every processing element is responsible for processing allocated tasks and sending messages to other processing elements. The algorithm terminates when no tasks remain to process.

Our architecture follows a vertex-centric computational model using a message-passing approach. In each superstep, *1)* we process messages sent between vertices, *2)* execute a user-defined function, and *3)* send new messages to be processed in the next superstep. The model maintains two queues of messages: *1) current messages queue* Q_c, which contains the messages to be processed at the current superstep and *2) next messages queue* Q_n, which contains the messages to be processed in the next superstep. When the current superstep terminates, we swap the empty Q_c with Q_n.

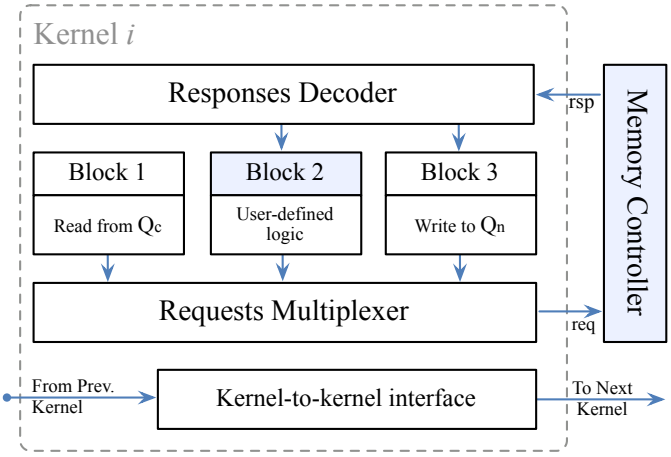

Fig. 1. A standard kernel template in our architecture reads from the current messages queue Q_c, processes messages, and writes back to the next messages queue Q_n. Kernels are arranged in a ring configuration to collaborate writing to Q_n.

Since our architecture targets shared memory multi-FPGA systems, we allocate the current and next queues in memory. Intuitively, the processing elements (hereafter named *kernels*) could share in processing Q_c by splitting the number of messages by the number of processing elements, or by just interleaving reading from Q_c. In order to avoid the problem of gathering messages at the end of each superstep and writing them to Q_n, our architecture utilizes kernel-to-kernel communication to help kernels write to Q_n. Message size and content are algorithm dependent and are left for the user to define.

B. Kernel Template

Figure 1 shows the proposed template design for our kernels. The kernels are arranged in a ring configuration, depicted in Figure 1 as the kernel-to-kernel interface, and discussed in the next subsection. At the initialization of the platform, all parameters are passed to the kernels in a pipelined fashion. Each kernel contains a memory multiplexer that is responsible for tagging memory requests and sending them to the memory controller. The response decoder is responsible for receiving memory responses, checking the enclosed tag, and forwarding the response to the appropriate function block. Block 2 hosts user-defined logic for application-specific node processing. In the case of compute-bound applications, multiple kernels can be structured to share a single memory controller port.

C. Kernel-to-kernel interface

In our architecture, the kernel-to-kernel configuration allows multiple kernels to collaborate writing to Q_n. An alternative solution is to use a more complex gather/scatter, model which comes at the cost of resources and design complexity. We extend the kernel-to-kernel interface to span multiple FPGAs through I/O ports. The interface behaves very similar to the familiar token ring LAN protocol as follows: a token circulates continuously through the kernels every clock cycle. At the

beginning of a superstep, the token is initialized by the first kernel to zero and sent to the next kernel. When a kernel receives the token, it increments its value by the number of slots it is going to write to Q_n. An on-chip FIFO is used to cache messages until a kernel receives the token and reserves space in Q_n. By the end of a superstep, the token value indicates how many messages reside in Q_n that need to be processed in the next superstep. If the token value is zero, then no tasks remain to be processed and execution terminates.

IV. All-pairs shortest path

In this section we provide a background review on the All-Pairs Shortest Path (APSP) algorithm. Then, we discuss our proposed parallel algorithm for APSP.

A. Baseline Sequential APSP

Computing shortest paths in a graph is considered a very common and fundamental graph algorithm in a wide range of real-world applications. The problem of all-pairs shortest path is described as follows: Given a directed graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$ mapping edges to real-valued weights, we want to find for every pair of nodes $u, v \in V$ the path with least-weight [21].

The APSP problem in unweighted directed graphs reduces to the problem of finding the path with shortest distance (i.e. least number of edges). Thus, running the breadth-first-search algorithm (BFS) from every node in the graph is an all-pairs shortest path algorithm that works on unweighted graphs. Algorithm 1 shows the baseline APSP for unweighted graphs using BFS.

Algorithm 1: Baseline APSP

Input: graph vertices v, and edges E
Output: Array d with distances

```
1 for u ∈ V do
      // traverse from u and update d(u, v)
2     BFS(V, E, u)
```

B. Multi-Step APSP

One valid strategy to parallelize APSP is to run multiple instances of BFS in parallel. However, this method may lead to wasted computational resources when processing large-scale sparse graphs. Thus, we propose a parallel multi-step APSP algorithm inspired by the level synchronous BFS [22]. In summary, the algorithm runs parallel threads of BFS traversal starting from all graph nodes. All the BFS threads use the same memory queues. Each computation that belongs to a BFS traversal is tagged with the source node that initiated the BFS traversal.

Graph representation: We consider an input graph represented using *compressed-sparse row format* (CSR) [23]. The CSR representation has been used widely due to its space-efficiency in representing sparse matrices and graphs. For a graph of n nodes and m edges, CSR consumes $\mathcal{O}(n+m)$ space compared to $\mathcal{O}(n^2)$ in the case of dense matrix representation. The CSR representation consists of two vectors: *i)* Column-indices (C) which is a vector storing the node's adjacency list and *ii)* Row-offset (R) which is a vector of pointers indicating where the adjacency list of each node starts. Given the values of $R[i]$ and $R[i+1]$, we can find the position and size of node i's adjacency list in C.

Algorithm: Our algorithm maintains three data structures: *i)* current queue (Q_c), which contains messages to be processed in the current step, *ii)* next queue (Q_n), which contains messages to be processed in the next step, and *iii)* distance array (d), an output $n \times n$ matrix containing the distance between every two nodes in the graph. Each message in Q_c and Q_n represents a traversal task. A message is a tuple with the first element indicating the source node of the traversal task, the second element is a pointer to the start of an adjacency list in C, and the last element is the number of elements in the adjacency list.

Algorithm 2: Multi-step APSP

Input: R, C vectors representing graph data in CSR
Result: d array of distances between every two nodes

```
1  Algorithm ParallelAPSP(R, C)
2      level ← 1
3      InitAPSP(R, C, d, Q_n)
4      while not Q_n.empty() do
5          swap(Q_c, Q_n)
6          APSPLevel(Q_c, Q_n, d, C, level)
7          level ← level + 1

1  Procedure InitAPSP(R, C, d, Q_n)
2      for i ∈ [0, N − 1]] do
3          for j ∈ [0, N − 1] do
4              visit_flag ← 0
5              neigh_count ← R[j + 1] − R[j]
6              d[i, j] ← ⟨R[j], neigh_count, visit_flag⟩
7          Q_n.push(⟨i, d[i, i]⟩)
8          d[i, i] ← ⟨0, 1⟩

1  Procedure APSPLevel(d, C, Q_c, Q_n, level)
2      while not Q_c.empty() do        // in parallel
3          ⟨u, d_uv⟩ ← Q_c.pop()
4          start ← d_uv[63..32]
5          size ← d_uv[31..1]
6          for i ∈ [start, size − 1] do
7              w ← C[i]
8              d_uw ← d[u, w]
9              if d_uw[0] = 0 then
10                 Q_n.push(⟨u, d_uw⟩)
11                 d[u, w] ← ⟨level, 1⟩
```

Algorithm 2 shows the pseudocode for our parallel APSP. The algorithm starts by issuing traversal tasks from all the

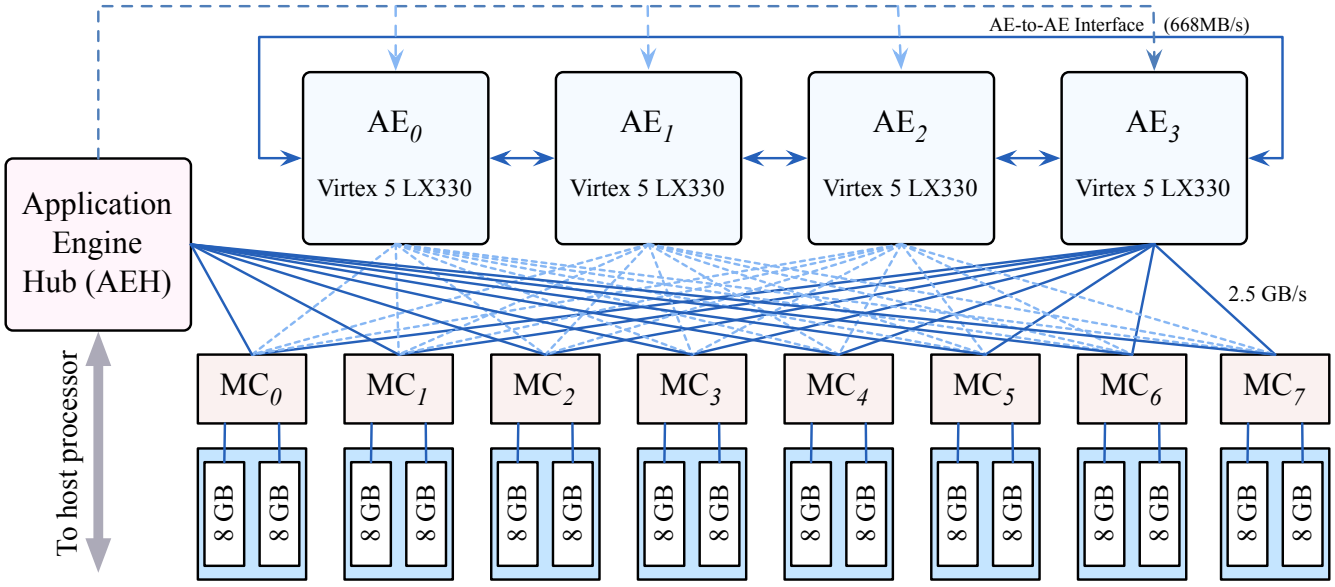

Fig. 2. The Convey HC-2 coprocessor board. The board consists of 4 programmable FPGAs connected to 8 memory controllers through a full crossbar connection.

nodes in the graph (i.e. for each node u, push the address of u's adjacency list to Q_n). Then, in every step we swap the empty Q_c and Q_n, execute all BFS traversals in Q_c and increase the current graph traversal level by one. We initialize the distance $d(u, v)$ with the tuple $\langle start, count, visit\ flag \rangle$. The $visit\ flag$ indicates if node v has been traversed from the source node u. The $start$ value is a pointer to the adjacency list of v in C. The $count$ value tells the length of the adjacency list. Therefore, for every traversal task, we read the distance between the source node u and every neighbor of an arbitrary node v. If the visited flag equals 0, we set it to 1 and update the distance (i.e. $d(u, w) = \langle level, 1 \rangle$, where w is a neighbor of node v). Then, we push a new task to traverse the adjacency list of node w from the source node u.

V. Hardware Implementation

In this section, we show how to transform our proposed multi-step APSP algorithm discussed in the previous section into a hardware kernel for our proposed graph processing architecture. Given n kernels in a design, each kernel is responsible for reading $|Q_c|/n$ messages from Q_c, completing traversal tasks, and writing new messages to Q_n.

Figure 3 shows our hardware implementation of the APSP kernel based on Algorithm 2. As discussed in Section III, the kernel uses a memory requests multiplexer and memory response decoder to maximally utilize the attached memory controller (MC) port. The APSP kernel consists of four processes running concurrently. The processes use on-chip FIFOs to push requests to the memory controller (MC). Then, the Requests Multiplexer reads the content of the FIFOs, tags the requests, and pushes the requests to the MC port. When the memory Response Decoder receives a response from the MC port, it checks the associated tag and pushes the response to the appropriate process block.

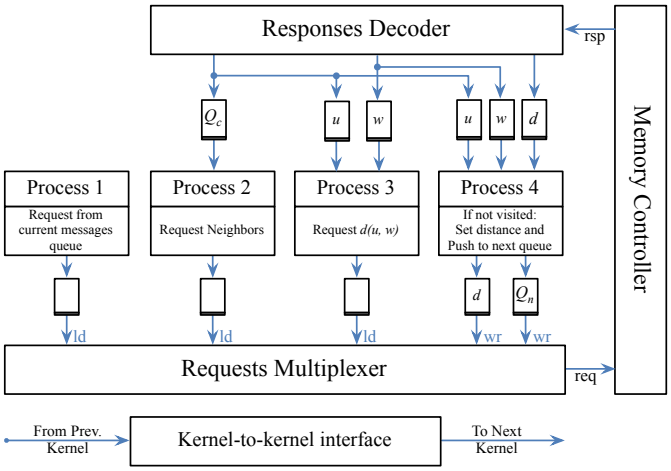

Fig. 3. The hardware architecture of the APSP kernel. Each kernel is designed to utilize one memory controller port.

The kernel processes work as follows: Process 1 is responsible for making read requests from Q_c. Kernels split Q_c among themselves. A message in Q_c is a tuple of the following data: source node (u), pointer to the adjacency list of an arbitrary node (v), and the size of the list. When a Process 1 response arrives from the MC, the message is pushed to Process 2. Also, Processes 3 and 4 receive information about the source node that issued the traversal task. Process 2 is responsible for reading the neighbor nodes in the adjacency list contained in the message. Process 3 receives the neighbor nodes and requests the distance between the source node and the neighbors. Given the source node u, an arbitrary node w, and the distance $d(u, w)$, Process 4 checks the visited flag in $d(u, w)$. If the node w has not been visited from the source

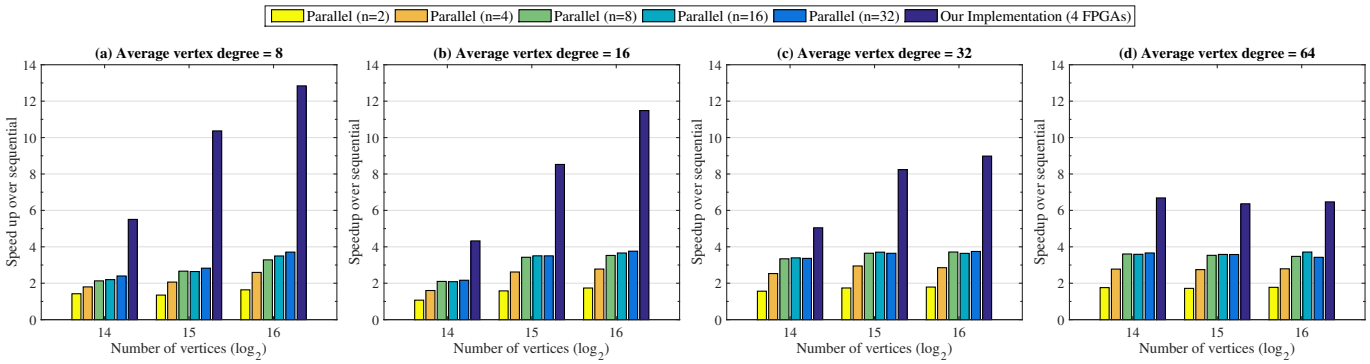

Fig. 4. Experimental results normalized to a sequential, single core software APSP implementation. A parallel software implementation with n number of processes, and our hardware APSP implementation are presented.

node u, Process 4 pushes the tuple $\langle u, d(u,w)\rangle$ to Q_n and updates the distance to be the current level (i.e. the number of the current superstep).

A kernel buffers messages to be written to Q_n in an on-chip FIFO until it obtains the communicaiton token. Once the kernel-to-kernel interface receives the token, it increments its value with the number of messages to be written in Q_n and passes the token to the next kernel (i.e. we reserve a space in Q_n). Then, the memory Requests Multiplexer can begin writing buffered messages into the reserved space.

VI. System Evaluation

For evaluating our architecture, we implemented our APSP hardware architecture using a commercial high performance reconfigurable computing platform, the Convey HC-2 [24], [25]. The Convey HC-2 is a heterogeneous computing platform that consists of an Intel Xeon E5-2609 CPU and a coprocessor board consisting of four programmable Xilinx Virtex5-LX330 FPGAs referred to as Application Engines (AEs). Figure 2 illustrates the components of the Convey HC-2 platform, which is convenient for prototyping our architecture. As shown in Figure 2, each AE has access to eight two-port on-board memory controllers in a full crossbar configuration, simultaneously providing each AE with memory access speeds of 20 GB/s. Our architecture employs 16 kernels per AE with each kernel harnessing exactly one of the sixteen memory controller ports. AEs communicate among themselves using the HC-2's full-duplex 668 MB/s ring connection called the AE-to-AE interface.

For the software implementations, we implemented a sequential APSP algorithm utilizing the Boost Graph Library (BGL version 1.57) [26]. We use the sequential implementation as a baseline to calculate speedup. Furthermore, we parallelized the software APSP algorithm using Open MPI for more direct comparison with our architecture. Both software versions executed on an Intel Core i7-860 with CPU frequency at 2.8 GHz and 16 GB of DDR3 RAM. The parallel implementation runs varying number of processes (2, 4, 8, 16, and 32 processes).

In our experiments, we used GTgraph [27], a synthetic graph generator suite, to generate R-MAT graphs with 2^{14}, 2^{15}, and 2^{16} nodes and an average vertex out-degree of 8, 16, 32, and 64. R-MAT is a fast recursive model for generating graphs with patterns very similar to those found in real-world graphs [28].

Table I. Memory bandwidth utilization for graph of size 2^{14} and average vertex degrees varying from 8 to 64

Num. of edges	Avg. degree	Throughput	Utilization
131,072	8	61.39 GB/s	76.73%
262,144	16	72.70 GB/s	90.87%
524,288	32	74.68 GB/s	93.35%
1,048,576	64	75.63 GB/s	94.55%

Figure 4 illustrates the performance of our FPGA compared to the parallel implementation. The y-axis shows the speedup against the sequential APSP algorithm utilizing BGL, and the log scale x-axis shows the number of nodes. The results show that for the parallel implementation, increasing the number of processes resulted in a modest speedup for each set of vertices and degrees. This trend is due to a better utilization of the available CPU cores as we increase the number of processes. However, the increase in performance began to saturate or diminish as the number of processes approached 8 regardless of graph size and average vertex degree which is attributed to our CPU having only 4 physical cores, each capable of running two threads concurrently.

The figure shows that our architecture achieves a speedup of $4.32\times$ to $12.84\times$ over the sequential software implementation as the number of nodes in the graph increases. Specifically, graphs with lower average vertex degrees result in a higher speedup over the parallel and sequential software implementations. This behaviour is due to the poor performance of commodity processors with irregular memory access patterns. However, as the graph's average vertex degree increases, the speedup of our architecture decreases. As the average out degree increases, there is a higher chance of having duplicates in the next queue (Q_n), consequently leading to race conditions and a degradation in performance.

Table I presents memory bandwidth utilization for the number of nodes fixed to 2^{14} and varying average vertex

degree. As shown in the table, despite the irregular memory access patterns of the APSP algorithm, our design maintians an increase in memory bandwidth utilization as the graph's average vertex degree increases. On average our implementation utilizes 88.87% of the 80 GB/s Convey HC-2 memory bandwidth.

Resource Utilization: Table II shows the device resource utilization of our hardware implementation using the Convey HC-2 platform. Our architectural template design has plenty of unused resources available for application specific computation or custom logic. We note that the overhead of the Convey HC-2 platform reserves about 10% of each FPGA's logic and 25% of the Block RAMs for dispatch and MC interfaces. Our implementation is clocked at a frequency of 150 MHz and uses FIFOs with a depth of 512.

Table II. Resource utilization for our architecture template and the APSP implementation

		Utilization	
Resource	Available	Template	APSP
Block RAM/FIFOs	288	25%	79%
Slice Registers	207,360	47%	67%
Slice LUTs	207,360	37%	56%
Slice LUT-FF pairs	207,360	57%	80%

VII. Conclusion

In this paper, we have described a novel scalable parallel multi-FPGA architecture utilizng a message-passing model for accelerating large-scale graph processing algorithms. We showcased our architecture with a parallel implementation of APSP using unweighted directed graphs on the Convey HC-2 platform. Using 4 FPGAs, our APSP implementation outperforms sequential and parallel software by up to a factor of $12.84\times$ and $3.67\times$, respectively. As future work, we will employ our architecture to implement different vertex-centric graph algorithms with real world applications. Also, we will explore graph partitioning approaches to scale our architecture between different nodes not sharing the same memory.

Acknowledgment

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

References

[1] N. Deo, *Graph theory with applications to engineering and computer science*. PHI Learning Pvt. Ltd., 2004.

[2] 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.

[3] M. Pohl, M. Schaeferling, and G. Kiefer, "An efficient FPGA-based hardware framework for natural feature extraction and related computer vision tasks," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[4] A. Agarwal, H. Hassanieh, O. Abari, E. Hamed, D. Katabi, and Arvind, "High-throughput implementation of a million-point sparse Fourier transform," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[5] Y. He, J. Wang, L. Wang, Z. J. Chen, C. Yan, H. Yang, H. Tang, C. Zhu, Q. Gong, Y. Zang *et al.*, "Uncovering intrinsic modular organization of spontaneous brain activity in humans," *PloS one*, 2009.

[6] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[7] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.

[8] Apache Giraph. http://giraph.apache.org/.

[9] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proceedings of the International Conference on Scientific and Statistical Database Management*, 2013.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, pp. 716–727, 2012.

[11] B. Betkaoui, Y. Wang, D. B. 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.

[12] 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.

[13] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: graphlet counting case study," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2011.

[14] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[15] A. Stratikopoulos, G. Chrysos, I. Papaefstathiou, and A. Dollas, "HPC-gSpan: An FPGA-based parallel system for frequent subgraph mining," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[16] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno, "CyGraph: A reconfigurable architecture for parallel breadth-first search," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing Workshops (IPDPSW)*, May 2014, pp. 228–235.

[17] G. Weisz, E. Nurvitadhi, and J. Hoe, "GraphGen for CoRAM: Graph computation on FPGAs," *Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, 2013.

[18] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martínez, and C. Guestrin, "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation," in *Proceedings of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2014, pp. 25–28.

[19] M. Ceriani, S. Secchi, O. Villa, A. Tumeo, and G. Palermo, "Exploring efficient hardware support for applications with irregular memory patterns on multinode manycore architectures," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[20] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

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

[23] Y. Saad, *Iterative methods for sparse linear systems*. Siam, 2003.

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

[25] K. K. Nagar and J. D. Bakos, "A Sparse Matrix Personality for the Convey HC-1," in *IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011.

[26] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Pearson Education, 2001.

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

[28] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the SIAM International Conference on Data Mining*, 2004, pp. 442–446.