A Reconfigurable Architecture for the Detection of Strongly Connected Components

OSAMA G. ATTIA, KEVIN R. TOWNSEND, PHILLIP H. JONES,
and JOSEPH ZAMBRENO, Department of Electrical and Computer Engineering,
Iowa State University

The Strongly Connected Components (SCCs) detection algorithm serves as a keystone for many graph analysis applications. The SCC execution time for large-scale graphs, as with many other graph algorithms, is dominated by memory latency. In this article, we investigate the design of a parallel hardware architecture for the detection of SCCs in directed graphs. We propose a design methodology that alleviates memory latency and problems with irregular memory access. The design is composed of 16 processing elements dedicated to parallel Breadth-First Search (BFS) and eight processing elements dedicated to finding intersection in parallel. Processing elements are organized to reuse resources and utilize memory bandwidth efficiently. We demonstrate a prototype of our design using the Convey HC-2 system, a commercial high-performance reconfigurable computing coprocessor. Our experimental results show a speedup of as much as 17× for detecting SCCs in large-scale graphs when compared to a conventional sequential software implementation.

Categories and Subject Descriptors: C.1.4.2 [**Computer Systems Organization**]: Reconfigurable Computing

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Breadth-first search, strongly connected components, convey HC-2, FPGA applications, graphs, reconfigurable computing

ACM Reference Format:
Osama G. Attia, Kevin R. Townsend, Phillip H. Jones, and Joseph Zambreno. 2015. A reconfigurable architecture for the detection of strongly connected components. ACM Trans. Reconfigurable Technol. Syst. 9, 2, Article 16 (December 2015), 19 pages.
DOI: http://dx.doi.org/10.1145/2807700

1. INTRODUCTION

Large-scale graph processing plays a fundamental role in many scientific computing applications. Examples can be found in diverse domains including bioinformatics [Mason and Verwoerd 2007], artificial intelligence [Dean et al. 2012], and social networking [Gjoka et al. 2010]. These applications have a strong tendency to be computationally intensive.

Commodity solutions in the market have been trying to meet the huge demand for performance of these applications by increasing the computation speed and capacity. Breakthroughs in the semiconductor industry have continued to result in an

This work was supported in part by the National Science Foundation (NSF), under awards CNS-1116810 and CCF-1149539.
Authors' addresses: O. G. Attia, K. R. Townsend, P. H. Jones, and J. Zambreno (corresponding author), Reconfigurable Computing Laboratory (RCL), Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010; email: {ogamal, ktown, phjones, zambreno}@iastate.edu.

DOI: http://dx.doi.org/10.1145/2807700

increase in the number of transistors per electronic chip. However, increasing the operational frequency becomes a harder challenge as the transistor density increases [Borkar and Chien 2011]. One way toward utilizing the abundant count of transistors is the use of multicore and heterogeneous cores, from which arises the need to have parallel algorithms and hardware implementation suitable for these new architectures [Alonso 2013]. Heterogeneous computing through reconfigurable computing-based systems have been shown to perform very efficiently, in terms of execution time and energy, in solving large-scale problems [Olson et al. 2012; Bakos 2010; Aluru and Jammula 2014].

One of the fundamental graph processing algorithms that serves as a classic starting step for many of the graph decomposition and analysis applications is the computation of Strongly Connected Components (SCCs). SCC is the problem of decomposing directed graphs into a set of disjoint maximal connected components [Cormen et al. 2009]. Solving the SCC problem for large-scale graphs, as with many other graph processing algorithms, is tightly coupled with the following challenges: the large memory footprint of the graph processing applications and domination of memory latency on the execution time, which is caused by the poor locality nature of large-scale graphs [Lumsdaine et al. 2007].

Trying to address these challenges, different implementations of parallel SCC detection have been proposed tackling multicore and GPGPU platforms. But to the best of our knowledge, the possible implementations of SCC detection for large-scale graphs remains uninvestigated for high performance reconfigurable computing systems. However, this is not the case for the well studied breadth-first search (BFS) graph traversal algorithm.

In this article, we present a reconfigurable architecture for parallel detection of SCC that extends our work in Attia et al. [2014]. The main contributions for this article are the following:

—A parallel BFS module based on Attia et al. [2014], which takes place as a core component in our SCC implementation.
—A hardware design for finding intersections that reduce the total execution time of the SCC problem.
—A practical hardware implementation and communication protocol for arranging our proposed processing elements.
—We describe how our architecture scales across multiple Field-Programmable Gate Arrays (FPGAs), and we show performance competitiveness for our BFS and SCC implementations.

The remainder of this article is organized as follows. In Section 2, we introduce related work. Section 3 goes through the needed background for the SCC detection algorithm and proposes our custom graph representation. In Section 4, we introduce our parallel SCC detection algorithm and the design details of the processing elements. Afterwards, we show the implementation of the architecture and resource sharing techniques in Section 5. Performance results and insights are presented and discussed in Section 6. Finally, conclusions are drawn and potential directions for future work are pointed out in Section 7.

2. RELATED WORK

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, multicore systems, Graphics-Processing Units (GPUs), and FPGAs. In this section, we review relevant work to the SCC decomposition and BFS algorithms.

There exist extensive studies on the implementation of graph algorithms in multicore platforms. Hong et al. [2013] have shown a parallel algorithm for SCC decomposition based on the FW-BW-Trim algorithm. Recently, Slota et al. [2014] gave a multistep algorithm for detecting SCCs in shared-memory multicore platforms. Shun and Blelloch [2014] shows another parallel algorithm for graph connectivity in multicore platforms. For BFS implementations, a parallel BFS algorithm for multicore architectures was described by Agarwal et al. [2010]. They use a bitmap to mark vertices 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 [Beamer et al. 2013; Checconi et al. 2012]. 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. For General-Purpose Graphics Processing Unit (GPGPU) implementations of BFS, Hong et al. [2011a] proposed a level-synchronous BFS kernel for GPUs. They showed improved performance over previous work. Merrill et al. [2012] used a prefix sum approach for cooperative allocation, leading to improved performance. In Zhong and He [2014], the authors present a GPU programming framework for improving GPU-based graph processing algorithms.

For FPGA-based implementations, Johnston and Bailey [2008] and Klaiber et al. [2013] demonstrate FPGA implementations of connected components with a specific application in analyzing streamed images. However, the problem of detecting SCCs for large-scale graphs is to date still uninvestigated. For BFS implementations, previous work by Wang et al. [2010] accommodated on-chip memory in order to store graphs. 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 high-performance reconfigurable computing platforms, such as the Convey HC-2/HC-2ex [Brewer 2010; Bakos 2010], are known for their high memory bandwidth and potential for application acceleration. For example, the Convey HC-2/HC-2ex platforms have been shown to be beneficial in speeding up various computation-intensive and memory-bound applications [Nagar and Bakos 2011; Betkaoui et al. 2012b; Townsend and Zambreno 2013; Augustin et al. 2011].

In our previous work [Attia et al. 2014], we introduced an efficient reconfigurable architecture, namely, CyGraph, for the BFS algorithm. Based on the CyGraph BFS kernel and following similar design approaches, we further extend the architecture to solve the SCC detection problem. Other close approaches to our work include Betkaoui et al. [2012a], which introduces a parallel BFS algorithm using the Convey HC-1 platform, and the recent BFS implementations by Convey using their latest HC-2ex and MX-100 systems [Wadleigh et al. 2012] illustrate the importance of this application kernel to the Graph500 supercomputer rankings [Murphy et al. 2010].

3. BACKGROUND

In this section, we introduce our custom graph representation based on the Compressed Sparse Row (CSR) format. Then, we go through the baseline SCC detection algorithm and the baseline BFS algorithm.

3.1. Graph Representation

For graph representation, we use our variant of the traditional CSR format. CSR has been widely employed for large-scale graph and matrix representations in scientific applications. Unlike an adjacency matrix representation that requires $\mathcal{O}(|V|^2)$ space

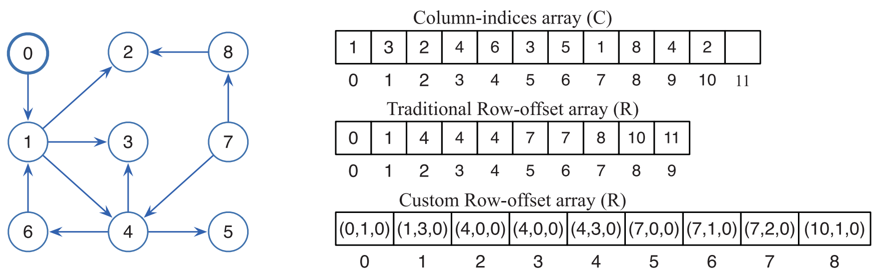


Fig. 1. Simple example for CSR graph representation format.

to represent graphs, the CSR format requires space only to store the nonzero data of the graph edges. This renders CSR a space and computation efficient data structure for representing sparse graphs.

The traditional CSR consists of two vectors: (I) column indices array C, which contains nodes adjacency list; (II) row offset array R, which contains the indices at which the adjacency list of each node starts. Figure 1 illustrates an example of the CSR representation format. Typically, for each node i in the graph, we have to read the visited bitmap and check if the node is flagged 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. In this example, in order to find the neighbors of node 1, we read $R[1]$ to find that the adjacency list of node 1 starts at $C[1]$. Then, we read $R[2]$ to find that the adjacency list of node 2 starts at $C[4]$. Finally, we issue memory requests to read the array items $C[1]$, $C[2]$, and $C[3]$, which hold the neighbors of node 1.

Many state-of-the-art computing platforms are capable of fetching 64-bit values per memory request. Thus, in order to reduce memory requests, we have manipulated the width of the row offset array R in the CSR representation to be a 64-bit wide array. For each element, $R[i]$, we will use the least significant bit of the row index as a *visited flag*. This allows us to get all information about a specific node in one memory request. Now, each element in the new R array contains a tuple of the following data:

—Start index (32-bit): indicates the start position of the node's neighbors list in C.
—Neighbors count (31-bit): refers to the size of the node's neighbors list in C.
—Visited flag (1-bit): flag is set to 1 if a node is visited and 0 otherwise.

Referring to the example in Figure 1, we need a single memory read request at $R[1]$ in order to find the start position of node 1's adjacency list in C and the number of neighbors. Then, we make memory read requests to read the neighbors from $C[1]$ to $C[3]$.

3.2. Baseline SCC Detection

In a directed graph, $G = (V, E)$, a SCC is a maximal connected subgraph where for any pair of nodes (u, v) in the subgraph, there exists a path from u to v and a path from v to u. For any arbitrary graph, we can decompose the graph into at set of disjoint SCCs. Each node is assigned a label or color that indicates to which SCC this node belongs. Decomposing the graph into SCCs represents a fundamental primitive in many graph analysis tools and applications. For instance, it can be used to divide big graph problems into smaller and easier subsets of the problem.

Traditionally, there are two sequential SCC detection algorithms that have been widely used in the literature:

—**Tarjan's algorithm**: which is considered the optimal serial algorithm for finding SCCs [Tarjan 1972]. Tarjan's algorithm has worst-case complexity of $\mathcal{O}(|V| + |E|)$. However, this algorithm depends on the Depth-First sSearch (DFS) algorithm, which is considered hard to parallelize.

—**Kosaraju's algorithm**: first published in Sharir [1981] and composed of two complete graph traversals (either BFS or DFS). This algorithm has a runtime complexity of $\Theta(|V| + |E|)$. The capability of using BFS in Kosaraju's algorithm makes it more convenient for parallel architecture.

In this article, we target Kosaraju's algorithm. Given a directed graph $G = (V, E)$ and its transpose graph $G' = (V, E')$, the algorithm starts from any arbitrary node v and finds the set of forward reachable nodes FW by traversing graph G using BFS or DFS algorithms. Then, using the transpose graph G', we find the set of backward reachable nodes BW. The set $S = FW \cap BW$ represents a unique strongly connected component. We remove S from the graph and iterate again until no more vertices exist. Algorithm 1 shows the pseudocode for Kosaraju's algorithm.

ALGORITHM 1: Kosaraju's Algorithm

```
Input: $G, G'$: directed graph, and its transpose
Output: $SCC$: set of strongly connected components
1 $v \leftarrow randomNode()$
2 while $G.|V| \neq 0$ do
3     $FW \leftarrow BFS(G, v)$
4     $BW \leftarrow BFS(G', v)$
5     $S \leftarrow FW \cap BW$
6     $SCC \leftarrow SCC \cup \{S\}$
7     Remove $S$ from $G$
```

3.3. Baseline BFS

The Kosaraju algorithm for detecting SCCs requires a graph traversal algorithm that returns all reachable nodes starting from a given node in the graph. One commonly used traversal algorithm in real-world applications is the BFS algorithm. This is in part due to the feasibility of parallelizing the BFS algorithm.

A commonly used approach for parallel breadth-first search is level-synchronous BFS [Hong et al. 2011b], which is illustrated in Algorithm 2. The level-synchronous BFS algorithm maintains the following sets of nodes: current nodes (Q_c), next nodes (Q_n), and visited nodes (V).

The algorithm starts with the root node in the current nodes set Q_c. Then iteratively, it visits all the nodes in the set Q_c, and adds their neighbors to the next nodes set Q_n. In the next iteration, Q_c is populated with the values in Q_n and Q_n is cleared for the next iteration. The algorithm uses the set V to flag nodes as visited. The sequential version of this algorithm runs in worst-time complexity of $\mathcal{O}(|V| + |E|)$.

4. DETECTING SCCS IN PARALLEL

Our parallel SCC detection algorithm is based on the previously described Kosaraju's algorithm using BFS traversal, and targets high-performance reconfigurable computing platforms with relatively large amounts of memory bandwidth. We also explore

ALGORITHM 2: Level-Synchronous BFS Algorithm

```
Input: root node
Input: Arrays $R$, $C$ holding graph data in traditional CSR
Output: Array visited holding traversed nodes

$Q_n$.push($root$)
$level \leftarrow 0$
$root.level \leftarrow level$
while not $Q_n$.empty() do
    $swap(Q_c, Q_n)$
    for $v \in Q_c$ do                                      // in parallel
        if $visited[v] = False$ then
            $visited[v] \leftarrow True$
            $v.level = level + 1$
            $i \leftarrow R[v]$
            $j \leftarrow R[v+1]$
            for $i < j$; $i \leftarrow i + 1$ do             // in parallel
                $u \leftarrow C[i]$
                if $visited[v] = False$ then
                    $Q_n$.push($u$)
    $level \leftarrow level + 1$
```

efficiency optimizations to the BFS algorithm that reduce the number of memory requests and hence increase overall system throughput.

4.1. SCC Algorithm

Algorithm 3 shows our parallel SCC algorithm based on Kosaraju's algorithm with parallel BFS. Given a graph and its preprocessed transpose graph represented in CSR formats, the algorithm starts with the *Trim()* procedure, which is a simple and important step to improve the performance by detecting any SCC of a single node. For any node in the graph that has an in-degree or out-degree of zero, the *Trim()* procedure will color it as a size-1 SCC. Then, the algorithm behaves in the same manner shown in the sequential Kosaraju. We use a parallel BFS algorithm based on the level-synchronous BFS to compute the set of reachable nodes in forward and backward starting from the node v. Then, the *parallelIntersection()* algorithm computes in parallel the intersection between the forward and backward reachability sets.

ALGORITHM 3: parallelSCC(R, C, R', C')

```
Input: Arrays $R$, $C$ representing graph in CSR format
Input: Arrays $R'$, $C'$ representing graph in CSR format
Output: Array $S$ holding colored vertices

$color \leftarrow 1$
$Trim(R, C, R', C')$
$v \leftarrow randomNode()$

while $v \neq \phi$ do
    $parallelBFS(R, C, FW, v)$
    $parallelBFS(R', C', BW, v)$
    $v \leftarrow parallelIntersection(S, R, R', FW, BW, color)$
    $color \leftarrow color + 1$
```

On each iteration, the *parallelIntersection()* returns the first uncolored node it encounters. We terminate the algorithm if there are no more uncolored nodes. The algorithm returns with a colored vector of the size of the input graph nodes. Each color represents to which SCC the specific node at this index belongs.

4.2. Parallel BFS

For our BFS implementation, we consider an optimized BFS implementation based on the traditional level-synchronous BFS algorithm. First, the use of the custom graph representation allows one to find out if a node is visited or not and the indices of its adjacency in just one memory request. Thus, we are cutting down the memory requests to half. Then, in order to save memory requests, we push the row index of node $R[i]$ instead of node ID to next queue. Hence, we make use of the data we fetched in a previous iteration. Algorithms 4 and 5 show the BFS after optimizations are applied.

ALGORITHM 4: parallelBFS(R, C, T, v)

```
Input: Arrays $R, C$ representing graph in custom CSR format
Input: Source node $v$ where BFS traversal starts
Output: Queue $T$ of traversed nodes (reachable nodes)
1  $Q_n.push(R[v])$
2  $R[v].visited \leftarrow 1$
3  $T.push(v)$
4  while not $Q_n.empty()$ do
5      $swap(Q_c, Q_n)$
6      $T \leftarrow T + count$                          // Shift T pointer
7      $count \leftarrow bfsLevel(R, C, Q_c, Q_n, T)$      // in parallel
```

ALGORITHM 5: bfsLevel(R, C, Q_c, Q_n, T)

```
Input: Arrays $R, C$ holding graph data, and root node
Input: Shared queues $Q_c, Q_n$
Output: Queue $T$ holding traversed nodes
1  while not $Q_c.empty()$ do
2      $v\_csr \leftarrow Q_c.pop()$
3      $start \leftarrow v\_csr[63..32]$
4      $size \leftarrow v\_csr[31..1]$
5      for $i \in [start, size - 1]$ do
6          $u \leftarrow C[i]$
7          $u\_csr \leftarrow R[u]$
8          if $u\_csr[0] = 0$ then
9              $R[i].visited \leftarrow 1$
10             $Q_n.push(u\_csr)$
11             $T.push(u)$
```

4.3. Finding the Intersection

Due to the iterative nature of the SCC detection algorithm and the nature of the platform we target, a naive algorithm to find the intersection is not feasible. Usually, the multi-FPGA target platform will work as a coprocessor that is triggered on every iteration by the host processor. Thus, it is more convenient to run and find the intersection on the coprocessor in order to save time and utilize the coprocessor high bandwidth access to the memory resources.

The intersection algorithm described in Algorithm 6 works in parallel to find all the common nodes between the backward and forward reachability sets. If a node was marked as visited in both forward and backward traversal, we color it to be included in the current SCC. However, if the node was not visited in both traversals, we mark the node unvisited in both the original and transposed graph and choose it as a possible next start for the next iteration. For the algorithm to behave correctly, it is important to keep the colored nodes flagged as visited (i.e., visited flag is binary 1) and to recover the flag of the uncolored nodes to binary 0. Finally, if we did not encounter any possible start nodes for the next iteration, we scan the SCC vector searching for a node that is not colored yet.

ALGORITHM 6: parallelIntersection(SCC, R, R', FW, BW, $color$)

```
Input: $R, R'$: graph, and reversed graph CSR data
Input: $FW, BW$: lists of forward and backward reachable nodes
Input: $color$: current SCC color
Output: $v$: next unvisited vertex
Result: $SCC$: colored vertices

1  $v \leftarrow \phi$
2  for $u \in \{FW \cup BW\}$ do                          // in parallel
3      if $SCC[u] = 0$ then
4          if $R[u].visited = 1$ and $R'[u].visited = 1$ then
5              $SCC[u] \leftarrow color$
6          else                                          // Unvisit
7              $R[u].visited \leftarrow 0$
8              $R'[u].visited \leftarrow 0$
9              $v \leftarrow u$                          // valid next start node
10 if $v = \phi$ then
11     for $i \in \{0, |R|\}$ do
12         if $R[i].visited = 0$ then
13             $v \leftarrow i$
14 return $v$
```

5. HARDWARE IMPLEMENTATION

In this section, we present the methodology that we used to prototype our SCC detection algorithm on multiple FPGAs. Then, we show how different design elements manage to share and reuse resources efficiently.

5.1. SCC Architecture

As shown in the previous section, the parallel SCC algorithm employs two main components, which are the *parallelBFS* and the *parallelIntersection*. In order to gain speedup over the baseline design we implement these parallel functions as hardware modules on the same FPGA. Figure 2 illustrates the system architecture of our implementation for multi-FPGA platforms with an assumption of a ring network between the FPGAs.

Each FPGA in our design includes a *BFS module* and *intersection module*, and each of them consists of multiple parallel kernels. The number of kernels is bounded by the number of available memory controller ports and available FPGA resources. Since the BFS and intersection module calls are called sequentially, a good implementation would be sharing FPGA resources and memory controller ports between the two modules. In

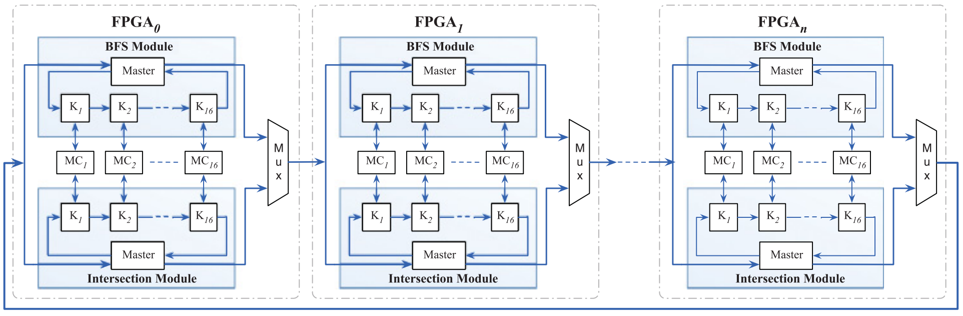


Fig. 2. Our SCC architecture for multiple FPGA platforms. Each FPGA consists of two modules: BFS module and intersection module. Each module employs 16 kernels (kernel denoted as K_i and connects to a single memory controller MC_i).

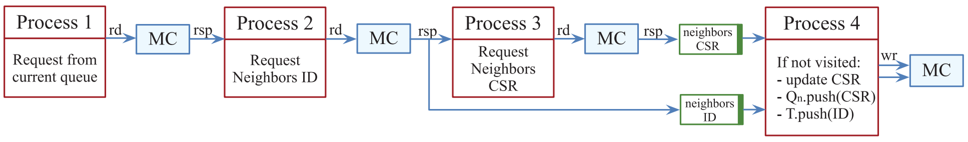


Fig. 3. CyGraph BFS kernel pipeline.

order to meet timing constraints, the kernels inside each module are maintained in a pipelined fashion only to pass control signals and parameters at initialization. The master processing element in each module is responsible for triggering and managing the flow of processing inside each FPGA.

5.2. CyGraph BFS Kernel

The BFS kernel serves as the building block for the *parallelBFS* module in our implementation. As shown in Figure 3, it is convenient to convert the BFS kernel to a pipeline of four stages based on Algorithm 5. 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 processing an issuing memory request that will be handled by the next process. 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. As shown in Figure 4, we propose a more efficient kernel design that consists of five processes, requests multiplexer, and a response decoder.

In our new kernel, 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). 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 a First In First Out (FIFO) queue. Process 3 gets the neighbors' ID and requests their CSR information, which will be available to process 4. Finally, process 4 checks the CSR information of each neighbor. If a neighbor is not visited, process 4 issues three memory requests:

—Enqueue neighbor's information (CSR) to the next queue.
—Enqueue neighbor's ID to the reachability queue.
—Flag this neighbor node as visited by updating (CSR).

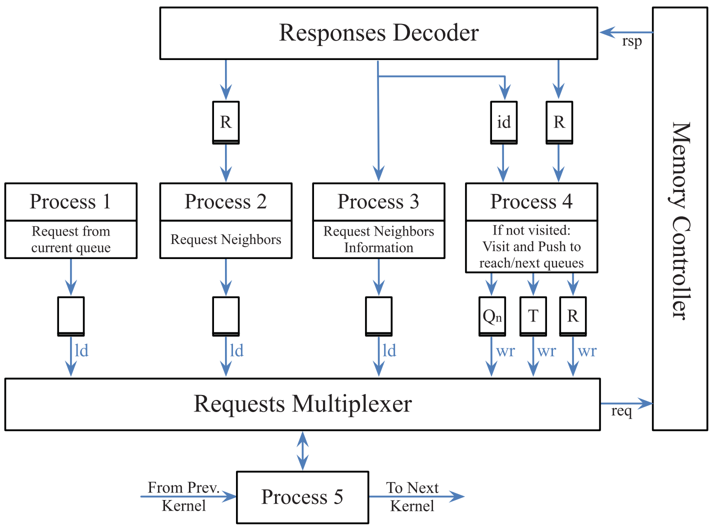


Fig. 4. BFS kernel architecture.

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 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 the next level.

5.3. Intersection Kernel

Intersection module is the second component in our implementation and it is responsible for two tasks; the first is to find the intersection between the two reachability sets (forward and backward reachability sets); the second is to unvisit the node if it is not detected for the current strongly connected component and return it as a possible start for the next iteration. The second task is important for the correctness of our algorithm. We employ the same design methodology shown in the previous subsection in order to design the intersection kernel. Figure 5 shows how to implement an intersection kernel using a pipeline of four processes. In order to utilize memory requests and reduce latency, we multiplex the memory requests of all the processes and decode responses back to the corresponding process. As illustrated in Figure 6, the new kernel consists of five processes, a requests multiplexer, and a response decoder.

For each intersection kernel, we start with process 1 interleave reading the forward reachability queue FW. Then, process 2 expects the memory responses for process 1

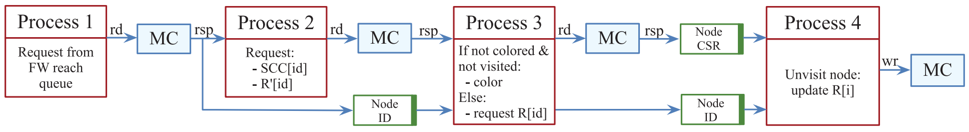

Process 1
Request from FW reach queue
rd
MC
rsp
Process 2
Request: - SCC[id] - R'[id]
rd
MC
rsp
Node ID
Process 3
If not colored & not visited: - color Else: - request R[id]
rd
MC
rsp
Node CSR
Node ID
Process 4
Unvisit node: update R[i]
wr
MC

Fig. 5. Intersection kernel pipeline.

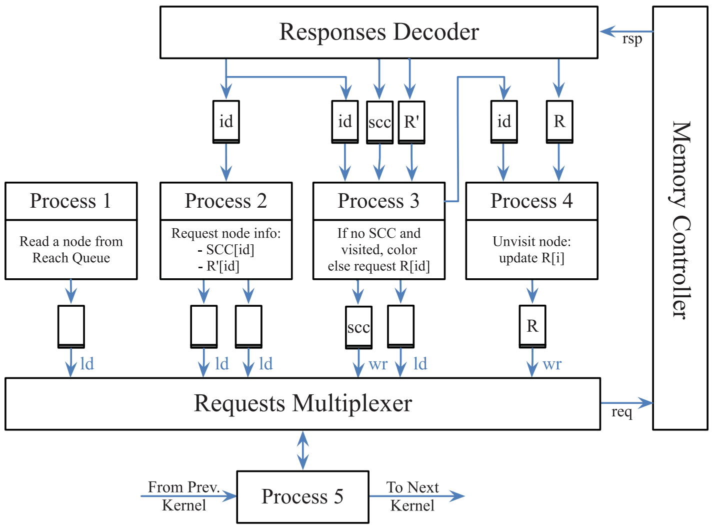


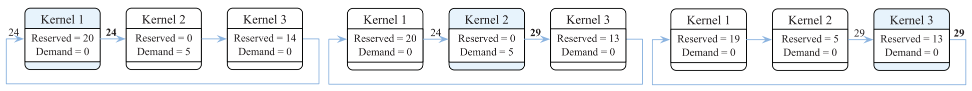
Fig. 6. Intersection kernel architecture.

and issue memory requests to request the node's current color and the node's CSR value in the transpose graph. Given that information along with node ID, process 3 knows that the node was visited in the forward traversal and if it is visited or not through the backward traversal. If the node is visited in backward traversal and not colored yet, process 3 colors this node. Otherwise, if the node is not colored and not visited in backward traversal, process 3 requests its CSR in order to mark the node unvisited. Finally, process 4 expects the CSR of the nodes that should be marked unvisited along with their ID and issue memory requests to update the CSR information. After finishing processing the forward reach queue, the processes follow the same approach to process the backward reachability queue BW. Process 5 keeps track of memory requests; if a node is found to be not colored yet and does not belong to the current SCC, it returns this node later as a possible next start for the next iteration and shares this information with other kernels.

5.4. FPGA Communication

In this architecture, we deploy a kernel-to-kernel communication, which is a 32-bit-wide ring network that is used to share information between kernels and extended between multiple FPGAs through FPGA I/O ports. In the SCC module, the kernel-to-kernel communication usage is limited to share a possible start node for the next iteration when it is found. Thus, other kernels should stop searching and iteration terminates.

In BFS module, we use the kernel-to-kernel communication to allow multiple reads and writes to the same memory-based queue. Different kernels have to read and write in parallel to shared queues. In the case of reading from the current queue, this problem



(a) Timestep i (b) Timestep $i+1$ (c) Timestep $i+2$

Fig. 7. A simple example showing a scenario of three kernels utilizing kernel-to-kernel interface to write to the next queue.

is trivial and is solved by interleaving or splitting the current queue. However, in the case of writing to the next queue and reach queue, kernels do not know exactly how many items will be written by each kernel.

In order to solve this problem, we designed our platform as a ring network as shown in Figure 2. 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 and reach queue. The kernel-to-kernel interface works as follows:

—The master kernel initiates 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 and reach queue from the last token.
—The token is reset at the beginning of every level.

In Figure 7 we show a simple example for how kernel-to-kernel interface functions in a scenario of three kernels. In this example, we take a snapshot of a system at an arbitrary timestep i in which the latest reserved address in the next queue is 23. At timestep i, kernel 1 receives a token with value 24 (i.e., next free address in next queue is 24). Since the demand counter equals zero, kernel 1 passes the token without changing its value. In the next timestep $(i+1)$, kernel 2 receives the token with value 24. Since its demand counter equals 5, kernel 2 sends the token to kernel 3 with value 29 (i.e., kernel 2 will start writing to next queue addresses from 24 to 28 and the next free address in next queue is 29). Meanwhile, kernel 3 wrote an item to its reserved space and decreased the reserved counter by one. In timestep $i+2$: Kernel 3 receives the token with value 29. Since the demand counter of kernel 3 equals zero, kernel 3 will forward the token back to kernel 1 without changing its value. Kernels might be consuming their reserved spaces or incrementing the demand independently from the kernel-to-kernel communication.

6. IMPLEMENTATION RESULTS

We ported our architecture to the Convey HC-2 and HC-2ex platforms [Bakos 2010]. The Convey HC-2 and HC-2ex are heterogeneous high-performance reconfigurable computing platforms that consist of a coprocessor board of four programmable FPGAs (Virtex5-LX330 on the HC-2 and Virtex6-LX760 on the HC-2ex). The Convey

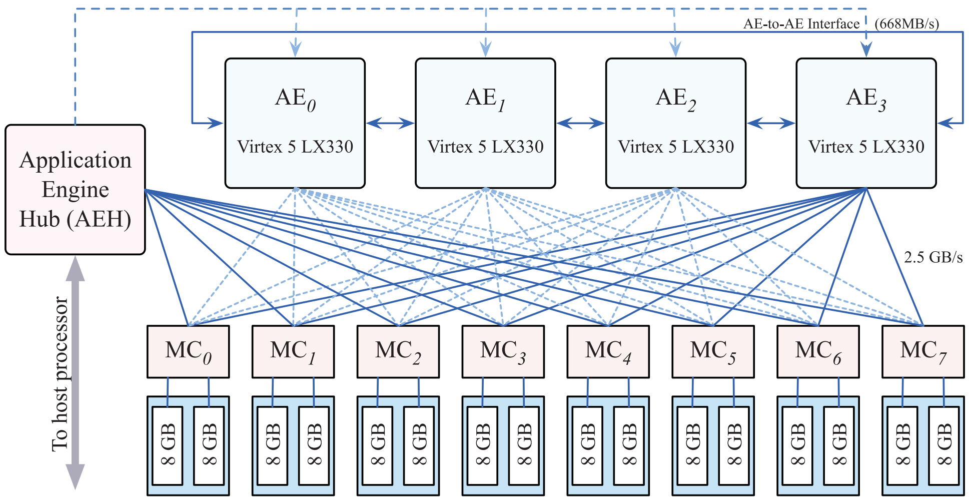


Fig. 8. The Convey HC-2 coprocessor board. The board consists of four programmable FPGAs connected to eight memory controllers through a full crossbar connection.

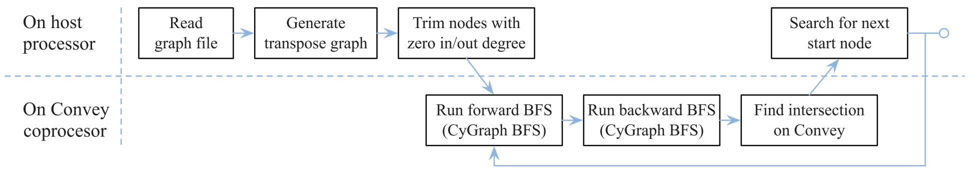

On host processor
On Convey coprocesor
Read graph file
Generate transpose graph
Trim nodes with zero in/out degree
Search for next start node
Run forward BFS (CyGraph BFS)
Run backward BFS (CyGraph BFS)
Find intersection on Convey

Fig. 9. The workflow of the SCC algorithm in Convey HC-2/HC2-ex platform.

programmable FPGAs are called *Application Engines* (AEs) and the custom FPGA configurations are called *personalities*. Each of the four AEs is operating at 150MHz and connected through a full crossbar to eight 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 80GB/s (i.e., each AE can access memory at a peak bandwidth of 20GB/s). Furthermore, the AEs are connected together through a full-duplex 668MBps 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 8, demonstrates the coprocessor board of the Convey HC-2 platform.

We deploy our implementation following the system architecture illustrated in Figure 2 for multiple FPGAs. Our custom implementation uses 16 CyGraph BFS kernels and eight intersection kernels per AE (i.e., 64 BFS kernels and 32 intersection 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 Figure 2. The processing starts at the host processor, which will process the graph data and generate transpose graph data. Then, we trim step the graph by coloring nodes that have in-degree or out-degree of zero because there is no gain from computing those in parallel. Then, we copy the graph data to the Convey coprocessor memory and invoke our coprocessor to start processing. Figure 9 shows

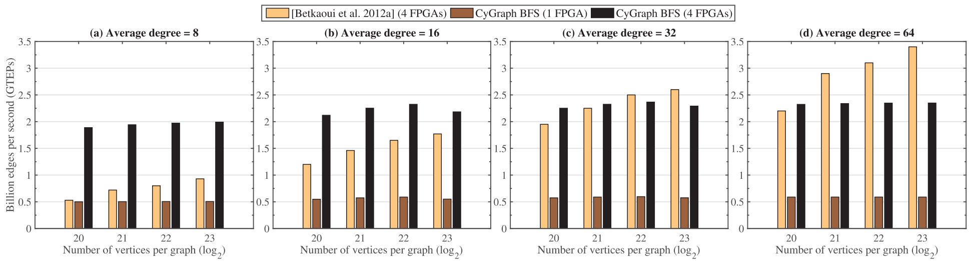


Fig. 10. BFS execution time for CyGraph against Betkaoui et al. [2012a] using random graphs.

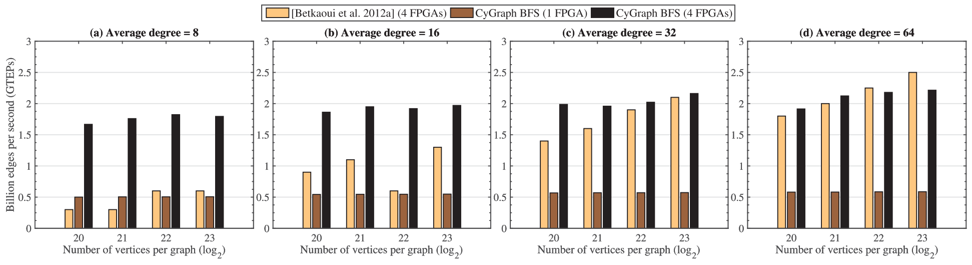


Fig. 11. BFS execution time for CyGraph against Betkaoui et al. [2012a] using RMAT graphs.

the workflow of the processing in the target platform. Our implementations for the CyGraph BFS and SCC modules are fully written in VHDL.

In the following two subsections, we will compare our CyGraph BFS implementation against a state-of-the-art implementation. Then, we show experiments of how our parallel SCC implementation, using CyGraph BFS and intersection modules, perform against a software sequential algorithm.

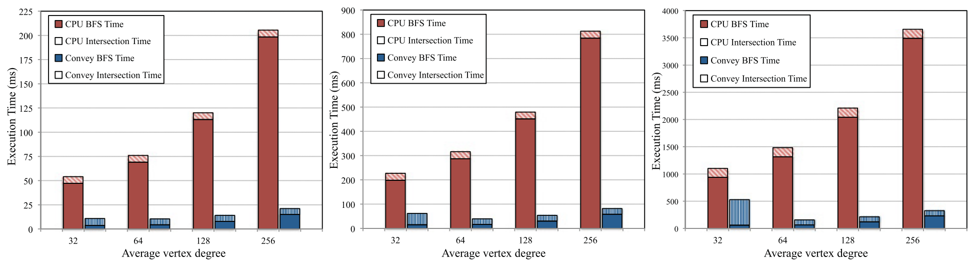
6.1. CyGraph BFS Experimental Results

For testing the BFS module, we have generated random and R-MAT graph data using GT-graph, a suite of synthetic graph generators [Bader and Madduri 2006]. R-MAT, Recursive Matrix, is a common approach toward generating realistic graphs [Chakrabarti et al. 2004]. Figures 10 and 11 show how our CyGraph BFS module performed against the BFS implementation from Betkaoui et al. [2012a]. Both of the implementations target the same platform, the Convey HC-2. We compare our implementation using one and four AEs against their implementation using four AEs. The comparison shows throughput in GTEPS (10^9 Traversed Edges per Second) and uses graphs with number of nodes that spans from 2^{20} to 2^{23}, and average vertex degrees that span from 8 to 64.

Figures 11(a) and 11(b) show that CyGraph maintains a speedup over the Betkaoui et al. implementation [Betkaoui et al. 2012a] 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 Betkaoui et al. [2012a] 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 a bitmap with a large number of vertices sequentially, updating a larger number of values per iteration and thus achieving higher eventual throughput. On the contrary, our implementation gains higher throughput in sparse graphs by using a memory queue data structure to mark nodes to updated in the next level. However, with the increase in average degree, this memory queue becomes quite close in size to the size of their bitmap.

Table I. Average Performance of CyGraph BFS Against Betkaoui et al. [2012a] in Billion Traversed Edges Per Second (GTEPS)

	Random graphs			R-MAT graphs		
Average degree	CyGraph One FPGA	CyGraph Four FPGAs	Betkaoui et al. Four FPGAs	CyGraph One FPGA	CyGraph Four FPGAs	Betkaoui et al. Four FPGAs
8	0.504	1.949	0.745	0.504	1.761	0.450
16	0.567	2.220	1.520	0.546	1.925	0.975
32	0.584	2.308	2.325	0.570	2.032	1.750
64	0.589	2.339	2.900	0.584	2.108	2.138



(a) Number of vertices = 2^{16} (b) Number of vertices = 2^{18} (c) Number of vertices = 2^{20}

Fig. 12. Our parallel SCC execution time using four FPGAs against the sequential software implementation.

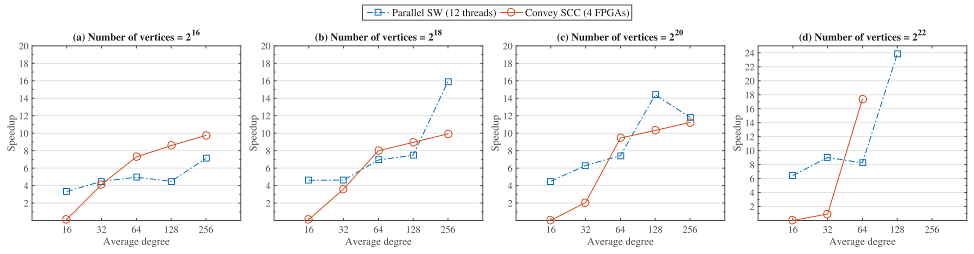


Fig. 13. A comparison of our SCC implementaiton using 4 FPGAs against a parallel software implementation using 12 threads running on the host processor. The results show speedup over a sequential software implementation.

Using the four Virtex-5 FPGAs, our design is capable of sustaining throughput that spans from 1.8GTEPS to 2.3GTEPS for large-scale random graphs. The summarized results in Table I show that on average our implementation obtains higher speedups over Betkaoui et al. [2012a] and comes quite close to its peak performance. Our performance performed 2.6× and 3.9× faster than Betkaoui et al. [2012a] for random and R-MAT graphs with average degree of 8.

6.2. SCC Experimental Results

Unfortunately, there are no previous FPGA implementations for SCC detection to compare against our results. Thus, we have generated random graph data using GT-graph and compared results against a sequential algorithm of SCC detection. We evaluate graphs with number of nodes that spans from 2^{16} to 2^{22} and average vertex degrees that span from 16 to 256. Our implementation utilizes the whole 4 FPGAs of the Convey machine. For benchmarking the sequential algorithm we used an Intel Xeon X5670 CPU with peak frequency of 3GHz. Figures 12 and 13 show the performance evaluation of our parallel SCC implementation against a sequential software implementation.

Table II. Resources Utilization

	Machine	Slice LUTs	Slice Registers	BRAM (Kb)
CyGraph BFS (one FPGA)	HC-2	111,538 (53%)	124,181 (59%)	6,678 (67%)
CyGraph BFS (four FPGAs)	HC-2	114,530 (55%)	125,638 (60%)	6,678 (67%)
SCC (four FPGAs)	HC-2	132,139 (63%)	150,585 (72%)	8,982 (86%)
SCC (four FPGAs)	HC-2ex	147,936 (31%)	153,248 (16%)	11,052 (21%)

Figure 12 shows the breakdown of execution time in milliseconds on the y axis against different average vertex degrees. The execution of the BFS algorithm obviously dominates the execution time of the SCC algorithm for our implementation and the sequential implementation. A first look over all instances shows that ours outperform the baseline sequential algorithm. The CyGraph BFS component contributes the major part of this performance by performing 15× faster than the sequential BFS implementation on average for the instances under investigation. The execution time does not include the overhead for copying the graph from the host processor to the Convey coprocessor. The overhead of copying data is minimal relative to the time required to set up the graph (on average 0.36% of the setup time).

In Figure 13, we summarize the speedups of our implementation against a baseline parallel implemenation described in Hong et al. [2013] using 12 threads.[1] The speedup is calculated over Kosaraju's sequential algorithm. Hong et al. [2013] proposed other improved algorithms for SCC; however, we do not compare against them since we do not employ these improvements in our implementation. The experimental results in Figure 13 show that our implementation outperforms the sequential implementation by speedup factor up to 17× in large-scale graphs of 4 million nodes and average degree of 64 (about 268 million edges). Our implementation is also competitive with the parallel implementation and outperforms it for average vertex degrees 64 and 128. With the increase of average vertex degree, the graph becomes more connected, which provides a chance for our implementation to cover memory latency and run for a longer time at its maximum performance. This explains the higher speedups achieved by the increase of average vertex degree. Thus, our implementation might be beneficial for applications with dense large-scale graphs.

6.3. Resource Utilization

Table II shows the device resource utilization for both the SCC and CyGraph BFS implementations. We note that our resource utilization does not increase significantly from using one FPGA to four FPGAs since it replicates the same design for each FPGA, plus only a few more gates to control the AE-to-AE interface.

As shown in the table, our CyGraph BFS module leaves relatively enough resources available, which we employed to add our intersection modules. Since the SCC implementation uses CyGraph BFS as a core component, we conclude that the intersection module does not occupy a small part of the FPGA resources. However, the remaining resources might not be plentiful in case more computation power is required. In Convey HC-2 platform, about 10% of the FPGAs' logic and 25% of the block rams are occupied for the required interfaces (e.g., dispatch interface and MC interface). The resources utilization reported in Table II includes this overhead. Thus, for large designs that require more resources, using the Convey HC-2ex machine is a more convenient option.

[1]The data in Figure 12(c) are missing data points due to bugs related to the large size of the graph.

7. CONCLUSION AND FUTURE WORK

In this article, we have investigated a novel parallel FPGA-based implementation for the detection of strongly connected components. First, we developed a parallel BFS and intersection kernels following a design approach that minimizes the algorithm's memory footprint and covers memory latencies. Then, we proposed a hardware architecture and communication protocol that utilizes our kernels in a scalable multi-FPGA design. Then, we demonstrated how our implementation for detecting SCCs outperforms sequential software implementation as much as 17× in large-scale graphs. Also, our BFS implementation achieved competitive performance against state-of-the-art FPGA implementation. Finally, we showed how our implementation utilizes resources on different FPGAs. We have publicly released the source code for this project on our research laboratory's Github repository.[2]

The suggested future work includes:

—Reduce the execution time spent on searching for new start points by implementing the search in hardware.
—Integrating our implementation in real-world graph analysis applications.
—Investigating the hardware implementation of alternative strongly connected component algorithms.

REFERENCES

Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. 2010. Scalable graph exploration on multicore processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Gustavo Alonso. 2013. Hardware killed the software star. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1–4.

Srinivas Aluru and Nagakishore Jammula. 2014. A review of hardware acceleration for computational genomics. *IEEE Design & Test* 31, 1 (Feb. 2014), 19–30.

Osama G. Attia, Tyler Johnson, Kevin Townsend, Phillip Jones, and Joseph Zambreno. 2014. CyGraph: A reconfigurable architecture for parallel breadth-first search. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing Workshops (IPDPSW)*. 228–235.

Werner Augustin, Jan-Philipp Weiss, and Vincent Heuveline. 2011. Convey HC-1 hybrid core computer the potential of FPGAs in numerical simulation. In *Proceedings of the International Workshop on New Frontiers in High-Performance and Hardware-Aware Computing (HipHaC)*.

David A. Bader and Kamesh Madduri. 2006. GTgraph: A Suite of Synthetic Graph Generators. Retrieved from www.cse.psu.edu/~madduri/software/GTgraph.

Jason D. Bakos. 2010. High-performance heterogeneous computing with the convey HC-1. *Computing in Science & Engineering* 12, 6 (Nov. 2010), 80–87.

Scott Beamer, Aydin Buluç, Krste Asanovic, and David Patterson. 2013. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. 1618–1627.

Brahim Betkaoui, Yu Wang, David B. Thomas, and Wayne Luk. 2012a. 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)*. 8–15.

Brahim Betkaoui, Yu Wang, David B. Thomas, and Wayne Luk. 2012b. 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)*. 99–104.

Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. *Communications of the ACM* 54, 5 (May 2011), 67–77.

Tony M. Brewer. 2010. Instruction set innovations for the Convey HC-1 computer. *IEEE Micro* 30, 2 (2010), 70–79.

[2]A public release of our project's source code is available at www.github.com/zambreno/RCL.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining (SDM)*. Chapter 43, 442–446.

Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. 2012. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd. ed.). MIT Press.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). 1223–1231.

Minas Gjoka, Maciej Kurant, Carter T. Butts, and Athina Markopoulou. 2010. Walking in facebook: A case study of unbiased sampling of OSNs. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*.

Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011a. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 267–276.

Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011b. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 78–88.

Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

Christopher T. Johnston and Donald G. Bailey. 2008. FPGA implementation of a single pass connected components algorithm. In *Proceedings of the IEEE International Symposium on Electronic Design, Test and Applications (DELTA 2008)*. 228–231.

Michael J. Klaiber, Donald G. Bailey, Silvia Ahmed, Yousef Baroud, and Sven Simon. 2013. A high-throughput FPGA architecture for parallel connected components analysis based on label reuse. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. 302–305.

Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. In *Parallel Processing Letters*, Vol. 17. 5–20.

Oliver Mason and Mark Verwoerd. 2007. Graph theory and networks in biology. In *IET Systems Biology*, Vol. 1. 89–119.

Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 117–128.

Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the graph 500. In *Cray Users Group (CUG)*.

Krishna K. Nagar and Jason D. Bakos. 2011. A sparse matrix personality for the convey HC-1. In *Proceedings of the IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

Corey B. Olson, Maria Kim, Cooper Clauson, Boris Kogon, Carl Ebeling, Scott Hauck, and Walter L. Ruzzo. 2012. Hardware acceleration of short read mapping. In *Proceedings of hte IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 161–168.

Micha Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72.

Julian Shun and Guy E. Blelloch. 2014. A simple parallel Cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing* 1, 1, Article 8 (Oct . 2014), 8:1–8:20 pages.

George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 550–559.

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1, 2 (1972), 146–160.

Kevin Townsend and Joseph Zambreno. 2013. 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)*. 185–191.

Kevin Wadleigh, John Amelio, Kirby Collins, and Glen Edwards. 2012. Hybrid breadth first search implementation for hybrid-core computers. In *SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*. 1354–1354.

Qingbo Wang, Weirong Jiang, Yinglong Xia, and Viktor Prasanna. 2010. A message-passing multi-softcore architecture on FPGA for breadth-first search. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*. 70–77.

Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 25, 6 (June 2014), 1543–1552.

Received November 2014; revised April 2015; accepted July 2015