TorusBFS: A Novel Message-passing Parallel Breadth-First Search Architecture on FPGAs

Guoqing LEI, Rongchun LI, Song GUO
National Laboratory for Parallel and Distributed Processing
National University of Defense Technology
Changsha, China

Fei XIA
Electronic Engineering College, Naval University of Engineering
Wuhan, China

***Abstract*—Graphs are a fundamental data structure used extensively in numerous domains. In graph-based applications, Breadth-First Search (BFS) is a key component which suffers from long latency of memory accesses. In this paper, we present a novel message passing parallel BFS architecture namely TorusBFS on field-programmable gate array (FPGA). By utilizing the on-chip memories to store the visitation status of vertices and to implement the current/next queue, our architecture reduces the accesses to the off-chip memories. We also present a on-chip 2-D torus message passing structure to reduce latencies of exchanging information among processing elements (PEs). Limited to the inefficient random write accesses to the off-chip memories, the experimental results show that our architecture on a single FPGA achieves relative lower performance compared with related works based on Convey HC-1/HC-2 platforms. Nevertheless, our TorusBFS is the first architecture that can be easily extended to multiple FPGAs in a distributed environment.**

Keywords-Breadth-First Search; Graph500; FPGA; Message Passing

I. Introduction

Graphs are a fundamental data structure widely used in numerous domains such as social networking analysis [1], bioinformatics [2] and artificial intelligence. Many data-intensive scientific problems can be solved through graph analysis. In the graph-based applications, Breadth First Search (BFS) is an important building block. Recently, BFS has attracted much more attentions as the kernel benchmark of the Graph500 rankings [4], which is used to measure the performance of supercomputers for the data-intensive applications.

Efficient parallel processing of large graphs is considered challenging [3]. The performance of graph-based applications is severely limited by the random nature of memory access patterns, which is a fundamental property of graph processing algorithms. Although the compute and bandwidth resources in modern computer architectures have been increasing, graph processing still suffers from the ineffective utilization of compute and bandwidth resources [7].

Recently, the field-programmable gate array (FPGA) has become a promising high performance computing platform. Combining the flexibility of software and highly customized hardware design, FPGAs can offer superior performance for many specific applications. Previous studies demonstrated the potentials of implementing the graph traversal algorithm on FPGAs. These studies either target the commercial servers namely Convey HC-1/HC-2 with four Xilinx Virtex-5 FPGAs[17][19] or in-house designed single Virtex-5 FPGA[16][18]. Limited to the on-chip memory resources for Virtex-5 FPGAs, all these previous designs can not fully exploit the benefits of on-chip memories, which is significant for improving the performance for graph processing algorithms. In this paper, we introduce a novel message-passing architecture for parallel BFS on recent high-end Xilinx Virtex-7 FPGAs (XC7VX485T), which has on-chip memories with 4× larger volume over that of Virtex-5. The main contributions for our approach are as follows:

- We introduce a novel on-chip bitmap-based distributed queue implementation method which avoids the off-chip memory accesses to decide whether the vertices are in the current level or not.
- We introduce a novel 2-D torus message-passing structure to exchange information among processing elements (PEs). Compared with related work, our structure reduces the numbers of on-chip FIFOs used to exchange information significantly.
- We have implemented the TorusBFS on Xilinx XC7VX485T FPGA, and compare the experimental results with related works based on the Convey HC-1/HC-2 platforms.
- Our TorusBFS structure is highly scalable and can be easily mapped to multiple FPGAs.

The remainder of this paper is organized as follows. We review the background and related works in section II. The proposed TorusBFS architecture is illustrated in section III. In Section IV, we extend the TorusBFS to multi-FPGA systems. The experimental results are presented in section V. Section VI concludes this paper.

II. Background And Related Works

A. *Level-synchronous BFS algorithm*

Our implementation is based on the level-synchronous BFS algorithm, as the algorithm 1 shows. The input of algorithm is an undirected graph $G(V, E)$ composed of a set of vertices V

and a set of edges *E*. Let the number of vertices and edges be *n* and *m*. Given a source vertex $s \in V$, the BFS algorithm explores the edges of *G* to produce a breadth-first search tree rooted at *s*. The output of BFS consists of two arrays: *Level*[1:n] and *Father*[1:*n*], where *Level*[*v*] and *Father*[*v*] denote the level and father of vertex *v* in the BFS tree respectively. The vertices to be explored in the current/next level are kept into two sets namely *CQ* and *NQ*.

As the algorithm begins, *Level*[*v*] and *Father*[*v*] are set to be 0 and -1 for $v \in V$ (lines 1-3). For the source vertex *s*, *Level*[*s*] and *Father*[*s*] are set to be 1 and *s* (lines 4-5). Initially, the set *CQ* only contains the source vertex and *NQ* is empty (lines 6-7). Let the variable *level* denotes the level value of vertices in CQ. For the CQ only containing source, the level value is 1 (line 7). The level-synchronous BFS algorithm is composed of *L* iterations, where *L* is the maximal level of the BFS tree rooted at *s*. In each iteration, for each vertex $u \in CQ$, the adjacency list of *u* will be traversed (line 12). If neighbor *v* has not been visited (*Level*[*v*] = 0) (line 13), then *Level*[*s*] and *Father*[*s*] will be set to be as *level*+1 and *u* (lines 14-15). At the same time, the vertex *v* will be added to the set *NQ* (line 16). After the adjacency list of all vertices of *CQ* have been traversed, the *NQ* and *CQ* will be swaped for the exploring in the next iteration (line 17). Finally, the BFS algorithm finishes when *CQ* is empty.

```
Algorithm 1: Level-synchronous BFS algorithm
Input: $G(V, E)$, source vertex $s$;
Output: Array $Level[1:n]$, Array $Father[1:n]$
Data: $CQ$: vertices to be explored in the current level;
      $NQ$: vertices to be explored in the next level;
1  for $v \in V$ do
2      $Level[v] \leftarrow 0$;
3      $Father[v] \leftarrow -1$;
4  $Level[s] \leftarrow 1$;
5  $Father[s] \leftarrow s$;
6  $CQ \leftarrow s$;
7  $NQ \leftarrow emtpy$;
8  $level \leftarrow 1$;
9  while $CQ \neq empty$ do
10     $NQ \leftarrow emtpy$;
11     for $u \in CQ$ do
12         for $each\ neighbor\ v\ of\ u$ do
13             if $Level[v] = 0$ then
14                 $Level[v] \leftarrow level + 1$;
15                 $Father[v] \leftarrow u$;
16                 $NQ \leftarrow NQ \bigcup \{v\}$;
17     Swap$(CQ, NQ)$;
18     $level \leftarrow level + 1$;
```

B. Parallel BFS algorithm

For the system with *np* processors, a natural way is to partition set of vertices V into *np* disjoint sets V_i, where each processor own $|V_i|=|V|/np$ vertices. Each processor is responsible for exploring the vertices of its own. For the *i*th processor, the local sets namely CQ_i and NQ_i are also defined to keep the vertices in the current/next level. Each processor works similarly with the simple level-synchronous BFS algorithm. For each vertex $u \in CQ_i$ of *i*th processor, each adjacent vertex *v* of *u* will be checked. For vertex *v* that has not been visited, we set the parent of *v* to be *u*. At the same time, *v* should also be added to NQ_i if $v \in V_i$. Otherwise, *v* will be sent to the NQ_j of *j*th processor if $v \in V_j$. For parallel BFS algorithm, all processors must synchronize after the processing of CQ_i in each level has been finished.

C. Related works

In order to improve the performance of graph processing especially BFS, many researches have been conducted for various computer architectures such as CPUs, GPUs (Graphics Processing Units), MIC (Many Integrated Core), and FPGAs.

Agarwal et al.[5] proposes a scalable BFS implementation for advanced multicore processors such as Intel Nehalem EP and EX processors. Xia et al.[6] achieved high-quality results for BFS explorations on Intel and AMD processors. Chhugani et al.[7] reduces the access overhead of the visitation status of graph vertices by eliminating the atomic operations. Beamer et al.[8] designs a direction-optimizing BFS algorithm, which reduces the memory access by combing the top-down and bottom-up algorithms and obtains superior performance.

For the GPU implementations, Luo et al.[9] present a GPU implementation of BFS that uses a hierarchical queue management technique. Hong et al.[10] present a novel virtual warp-centric programming method to address the problem of workload imbalance. They improve the performance by several factors upon previous GPU implementations. Hong et al.[11] also present a hybrid method which combines the CPU and GPU execution. Merrill et al.[12] present a BFS parallelization focusing on fine-grained task management constructed from efficient prefix sum. They achieve higher performance over previous works using single and quad-GPU configurations. Zou et al.[13] present a direction-optimizing implementation on CPU-GPU heterogeneous platforms. They obtain improvement over the highest pervious performance for shared memory systems.

For the MIC-implementations, Saule et al. [14] evaluate scalability results of three variations of a breadth-first search algorithm using programming models such as OpenMP, Cilk Plus and Intels TBB. Gao et al. [15] discuss how to use MIC to accelerate the BFS algorithm and propose a heterogeneous hybrid BFS algorithm combining the top-down and bottom-up version.

For the FPGA-implementations, Wang et al. [16] propose a message-passing multi-softcore architecture on FPGA for Breadth-First search. The vertices of the graph are divided into equal-sized disjoint sets owned by different softcores. In their work, the next queue is implemented by transferring the vertices in the next level to the corresponding cores using FIFOs. For a system with p cores, p^2 FIFOs are needed for bi-directional communication between each pair of cores, which limits the scaling of the algorithm. Betkaoui et al. [17] introduce a parallel graph exploration algorithm using the same division method as Wang et al.[16]. They do not implement the

Identify applicable sponsor/s here. *(sponsors)*

current/next queue. The vertices in the current level are determined by repeatedly reading the distance of vertex which is compared with current level value. Ni et al.[18] present a multi-channel memory based architecture for parallel BFS algorithm. They use two DRAM modules and two SRAM chips to increase the bandwidth of the off-chip memory. Attia et al.[19] present an efficient reconfigurable architecture that adopts a custom graph representation and restructuring of the conventional BFS algorithm for parallel BFS. They store the current queue and next queue in the off-chip memory which are shared by all kernels. The current queue are split and read by interleaving method by different kernels. The writing to the next queue are solved by a token circulating through the kernels every clock cycle.

III. PROPOSED ARCHITECTURE

For efficiently processing parallel BFS algorithm on FPGAs, we must partition the tasks to multiple processing elements performing in parallel. In this paper, we divide the vertices into equal-sized disjoint sets assigned to multiple graph processing elements (PEs) to process in parallel. Each of PEs is responsible for exploring its assigned vertices. These vertices are also called local vertices for each PE. For those vertices belonging to the other PEs, a router is designed to send messages to the target PE.

The architecture of our message-passing parallel BFS algorithm is illustrated as Figure 1. The master module is responsible for initializing the PEs and routers in the PE-router Torus 2-D Array, and synchronizing the searching process of each level. It also returns the finish signal to the host when the BFS algorithm is done. Let the number of PEs is *np* and the width of the Torus 2-D array is *d* which satisfies $np=d^2$. Each of PEs is assigned a router *R* which is used to send and receive messages to and from other PEs respectively. The multi-ports memory access module is responsible for multiplexing the memory requests of PEs to the single DDR3 memory controller for improving the bandwidth utilization. Three DDR3 memory chips are available, two of which are used to store the graph data and the other one is used to store the output results.

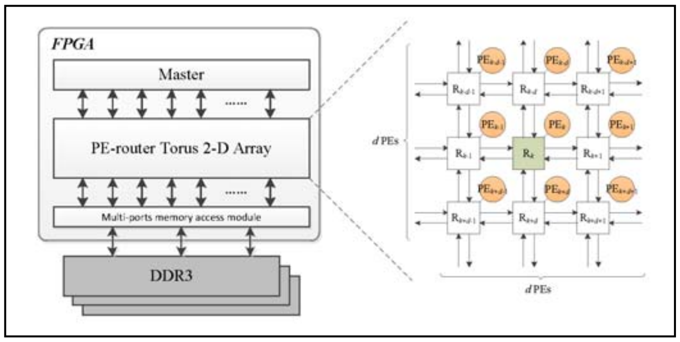

Figure 1. Block diagram of our BFS architecture.

A. PEs design

Algorithm 2 presents the BFS kernel that runs on the PE. The PE consists of two main phases namely exploring and messaging that execute in parallel. A detailed description of each phase follows:

1. **Exploring the vertices of current level**. In this phase, the values in the *Bitmap_CQ* for each local vertex i are read to judge whether i is in the current level. If *Bitmap_CQ*[i]=1, the global vertex id *u* is generated by combining the *pe_id* and local id *i* together. Then the *R* array of the global vertex *u* is read to get the start and end positions of adjacency list. In order to deal with vertex with large number of neighbors, *q* neighbors are loaded from off-chip memory and processed each time. For each neighbor *v* of the vertex *u*, if *v* belongs to this PE and *v* has not been visited, then a process called *Process_unvisited_vertex*(*v*) is performed on vertex *v* which includes four operations: 1) Add the vertex *v* to the next queue by setting *Bitmap_NQ*[*v*]=1; 2) Set the *bitmap*[*v*]=1 to indicate that the vertex *v* has been visited; 3) Set the *Level*[*v*] to be *level*+1; 4) Set the *Father*[*v*]=*u*. Otherwise, if *v* belongs to other PEs, then the message (*u*, *v*) is sent to the router and then forwarded to the owner PE of vertex *v*. After each local vertex *i* has finished processing, the value *Bitmap_CQ*[*i*] will be set to 0. The swap between *CQ* and *NQ* is implemented by alternating reading/writing between *Bitmap_CQ* and *Bitmap_NQ* depending on the level value.

Algorithm 2: BFS kernel executed on each PE

```
Input: $R[0:n-1]$: offsets of adjacency lists of all vertices;
       $C[0:m-1]$: CSR adjacency lists;
       $p$: the number of local vertices of each PE;
       $pe\_id$: the ID of PE;
       $m\_fifo$: caching messages coming from other PEs;
Output: Array $Level[0:n-1]$;
        Array $Father[0:n-1]$;
Data: $Bitmap\_CQ[0:p-1]$: local current queue;
      $Bitmap\_NQ[0:p-1]$: local next queue;
      $Bitmap\_visited[0:p-1]$: visited status of local vertices;
      $q$: number of neighbors processed each time;
      $level$: the level of vertices explored in the current level;
//1.Exploring the vertices of current level.
for $i \leftarrow 0$ to $p-1$ do
    if $Bitmap\_CQ[i] = 1$ then
        $u \leftarrow \{pe\_id, i\}$;
        for $offset \leftarrow R[u]; offset \leftarrow R[u+1]; offset += q$ do
            for $j \leftarrow 0$ to $q-1$ do
                $v \leftarrow C[offset+i]$;
                if $v \in PE$ then
                    if $Bitmap[v] = 0$ then
                        $Process\_unvisited\_vertex(v)$;
                else
                    Send message $\{u, v\}$ to the owner of $v$;
        $Bitmap\_CQ[i] \leftarrow 0$;
//2. Processing the messages from other PEs.
while $m\_FIFO$ != empty do
    $(u, v) \leftarrow Read\ m\_FIFO$;
    if $Bitmap[v] = 0$ then
        $Process\_unvisited\_vertex(v)$;
```

2. **Processing the messages from other PEs**. In this stage, the fifo *m_fifo* is read if there are messages sent to this PE. For each message (*u*, *v*), the local bitmap *Bitmap*[*v*] is read to determine whether the vertex *v* has been visited. If the vertex v

has not been visited, the process *Process_unvisited_vertex*(v) is then preformed on vertex v.

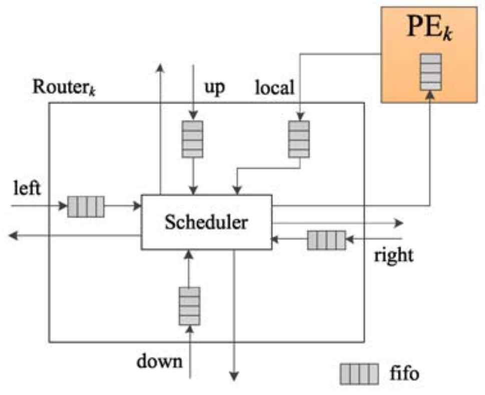

Figure 2. Block diagram of Router design.

B. *Router design*

The router module is responsible for receiving messages from five incoming directions including up, down, left, right and local PE and sending these messages to five outgoing directions that are opposite from the incoming. Figure 2 shows the structure of the router design. There are five FIFOs caching the messages from the five incoming directions. The module Scheduler repeatedly reads messages from the incoming FIFOs and decide the forwarding. The positions of both router and the PE can be denoted using their row indices and column indices in the 2-D Torus array. In this way, when the route r in the position (r_i, r_j) needs to process a message forwarding to the target PE t in the position (t_i, t_j), the forwarding direction can be generated by the rules as the table I shows. The rules work in two steps. Firstly, r_i and t_i are compared. The message will be sent to the router in the up direction if $t_i < r_i$, otherwise the down direction will be chosen. Secondly, if r_i equals t_i, r_j and t_j are then compared to choose the forwarding direction from the left three directions: left, right or local.

TABLE I. FORWARDING RULES OF ROUTERS

Forwarding direction	Rules
up	$t_i < r_i$
down	$t_i > r_i$
left	$t_i = r_i \wedge t_j < r_j$
right	$t_i = r_i \wedge t_j > r_j$
local	$t_i = r_i \wedge t_j = r_j$

A conflict occurs when more than one message are sent to the same output port at the same time. To solve this problem, five counters are used to record the number of messages for each incoming FIFO. The message from the FIFO with the highest counter will be sent to the output port when conflict occurs. After the message has been forwarded, the next message from that FIFO will be read as soon as possible. In our design, the total number of FIFOs for the routers to exchange information among *np* PEs is 5×*np*, significantly lower than $(np)^2$ of Wang et al.[16].

IV. BFS ON MULTI-FPGA SYSTEM

Our TorusBFS can be easily extended to multi-FPGA systems. The techniques presented in this section can be applied to both multiple FPGAs on one host processor and mutiple FPGAs distributed on mutiple host processors. The multi-FPGA system with mutiple FPGAs on one single host processor is considered in this paper. When mutiple host processors are employed, the communication between FPGAs can be replaced by communication between processors.

We assume that TorusBFS is performed on a 2 × 2 2D grid of FPGAs as shown in Figure 3. For each FPGA, graph data and BFS results are stored in its local DRAMs. Each FPGA can communicate with its neighbors.

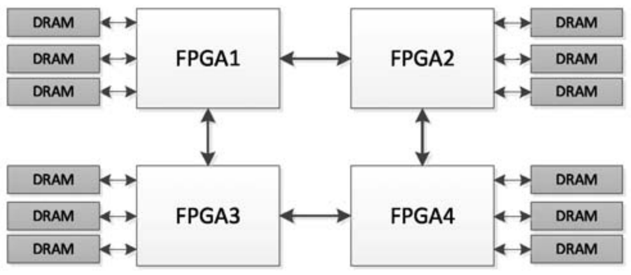

Figure 3. Multi-FPGA system for TorusBFS.

For a single FPGA, one 2D Torus array of PE-routers is employed to execute exploring of vertices in the same level at a time. Each partition of vertices is executed on the PE in parallel. For a multi-FPGA system with each FPGA containing one 2D Torus arrya of PE-routes, the vertices of graph will be divided into partitions equally distributed on mutiple FPGAs. The partition of vertices on each FPGA are processed as the single FPGA system. The fathers and levels of vertices in each partition are stored in one of local DRAMs of each FPGA. Inter-FPGA communication is required to exchange messages between FPGAs. In practice, inter-FPGA communication can be implemented by only connecting the data transfer controllers of the 2D Torus array of PE-routers in neighoring FPGAs.

One typical advantage of multi-FPGA system over the single-FPGA is the linear increasing of off-chip DRAM bandwidth. For the data-intensive application such as BFS, the performance of our TorusBFS scales linearly theoretically with the number of FPGAs in a distributed computing environment. This is because: 1) the bottleneck of our TorusBFS is the latency of off-chip memory accesses; 2) the latency of message passing between FPGAs can be ignored by directly connecting the data transfer controllers between neighboring FPGAs.

V. EXPERIMENTAL RESULTS

We test our TorusBFS architecture on the self-designed FPGA prototyping system. The system is a high performance computing embedded platform that consists a zynq processor and a coprocessor of one programmable Virtex-7 XC7VX485T FPGA. For the off-chip memories, three DDR3 chips each of which has 8GB volume are connected to the FPGA. There are also three on-chip memory controller generated with Xilinx

MIG 7 Series tools to provide access to the corresponding off-chip DDR3 chips. The memory controller works at frequency 166MHz and allows accesses to DDR3 memory at peak bandwidth of 10.6 GB/s. We have implemented 16 PEs and 16 routers in our design. The size of bitmap in each PE is 256K bits, which means that the maximal scale that our TorusBFS can process is 22. We have test the performance of TorusBFS on only one Virtex-7 FPGA recently, the performance results for four FPGAs are estimated according to the linear scalability characteristics of TorusBFS on multi-FPGA systems.

We use scale-free graphs to test the performance of our TorusBFS. The scale-free graphs are generated using the Graph500 benchmark suit based on the Recursive-Matrix (R-MAT) graph model. The parameters of the R-MAT graph are set as the default values of the Graph500 benchmark (A=0.57, B=0.19, C=0.19). The performance is measured by taking the average execution time of 64 BFS runs from 64 different source vertices which are randomly chosen. The performance is reported as the throughput in billions of traversed edges per second (GTEPs).

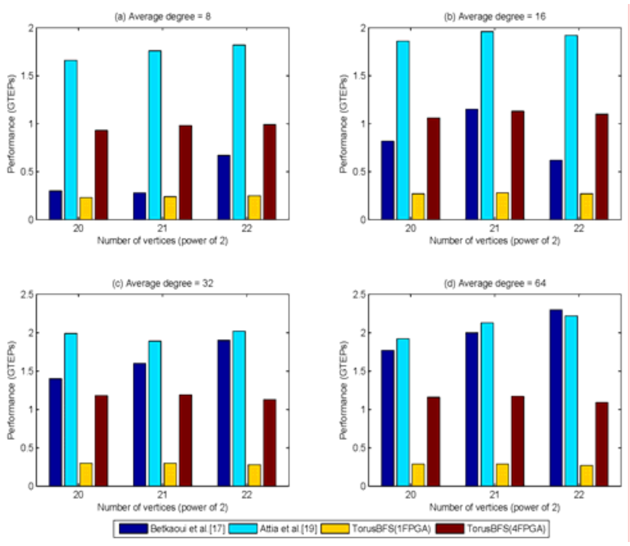

Figure 4. BFS performance against related works for RMAT graphs.

A. *Performance analysis*

Figure 4 demonstrate how our TorusBFS performs against the BFS implementations from Betkaoui et al.[17] and Attia et al.[19]. Both of these two implementations target the same platform, the Convey HC-1/HC-2, which consists of a coprocessor board of four programmable Virtex5-LX330 FPGAs. The comparison shows throughput with number of vertices that spans from 220 to 222 and average vertex degree that spans from 8 to 64.

For the average degree of 8, the TorusBFS on one FPGA performs as well as that of Betkaoui et al.[17] for scale 20 and 21. For the average degrees of 8 and 16, our TorusBFS on four FPGAs performs better than that of Betkaoui et al.[17] for all scale ranges. While for the larger degrees of 32 and 64, our TorusBFS on both one and four FPGAs performs worse than that of both Betkaoui et al.[17] and Attia et al.[19]. It can be inferred that the workload imbalance and latency of off-chip memory access limit the higher processing rate of TorusBFS.

Figure 5 shows as the average vertex degree grows from 8 to 64, how the performance of our TorusBFS scales on single FPGA. The performance for average vertex degrees of 16, 32 and 64 is similar and larger than that of 8. It can be inferred that larger vertex degrees introduce more memory accesses and improve the bandwidth utilization ratio.

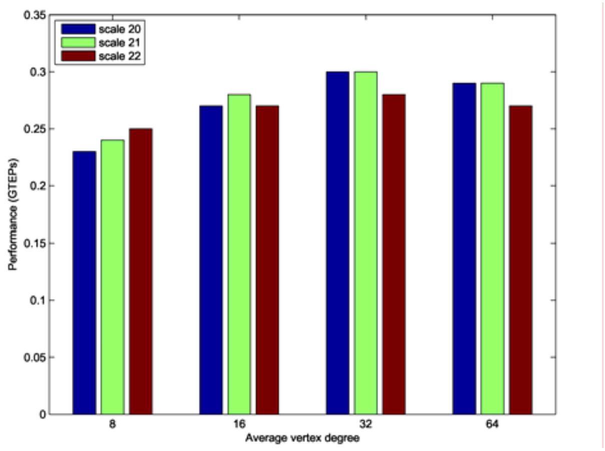

Figure 5. Performance scaling with respect to average vertex degree for RMAT graphs of scale 20, 21 and 22.

B. *Resource utilization*

Table II shows the device resource utilization compared with related implementations. Our TorusBFS architecutre consumes about 91% of block ram memories, most of which are used for storing the visitation status of vertices and implementing current/next queue. However, it should be noted that about 10% of the FPGAs' logical and 15% of the block rams are occupied for the required interfaces (e.g. Memory controller interface, and memory requests multiplexing).

TABLE II. RESOURCE UTILIZATION

	Slice LUTs	**BRAMs**	**Slice LUT-FF**
Betkaui ea al.[17]	80%	64%	n/a
Attia et al.[19]	55%	55%	74%
Our TorusBFS	66%	91%	21%

VI. CONCLUSION

In this paper, we have proposed a novel FPGA-based message-passing parallel BFS architecture, namely TorusBFS. Compared with other implementations based on Convey HC-1/HC-2 platforms, our implementation shows relative lower performance, which resulted from the lower off-chip memory bandwidth utilization. One limitation of our TorusBFS is higher consumption of on-chip block rams. The graph scale over 22 can not be processed in a single FPGA chip. In order to perform BFS for larger graphs scale (e.g. 23), more advanced FPGA chips may be needed. Nevertheless, our TorusBFS can easily be mapped to mutiple FPGA chips to process larger graphs, the performance scales linearly with the number of FPGAs theoretically. We will test the performance of TorusBFS on multi-FPGA system in future.

ACKNOWLEDGMENT

This work is partially supported by National High Technology Research and Development Program of China under No. 2012AA012706, and by National Science Foundation China under Grants 61125201, 61202127 and 61303061.

REFERENCES

[1] Ediger, D., Jiang, K., Riedy, J., Bader, D.A., Corley, C., Farber, R., Reynolds, W.N.: Massive social network analysis: Mining twitter for social good. In: Parallel Processing (ICPP), 2010 39th International Conference on. pp. 583–593. IEEE (2010)

[2] Alvarez Vega, M.: Graph kernels and applications in bioinformatics (2011)

[3] Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J.: Challenges in parallel graph processing. Parallel Processing Letters 17(01), 5–20 (2007)

[4] Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray Users Group (CUG) (2010)

[5] Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable graph exploration on multicore processors. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp.1–11. IEEE Computer Society (2010)

[6] Xia, Y., Prasanna, V.K.: Topologically adaptive parallel breadth-first search on multicore processors. In: Proceedings of the 21st IASTED International Conference. vol. 668, p. 91 (2009)

[7] Chhugani, J., Satish, N., Kim, C., Sewall, J., Dubey, P.: Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. pp. 378–389. IEEE (2012)

[8] Beamer, S., Asanovic, K., Patterson, D.: Direction-optimizing breadth-first search. Scientific Programming 21(3-4), 137−148 (2013)

[9] Luo, L., Wong, M., Hwu, W.m.: An effective gpu implementation of breadth-first search. In: Proceedings of the 47th design automation conference. pp. 52–55. ACM (2010)

[10] Hong, S., Kim, S.K., Oguntebi, T., Olukotun, K.: Accelerating cuda graph algorithms at maximum warp. In: ACM SIGPLAN Notices. vol. 46, pp. 267–276. ACM (2011)

[11] Hong, S., Oguntebi, T., Olukotun, K.: Efficient parallel graph exploration on multicore cpu and gpu. In: Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. pp. 78–88. IEEE (2011)

[12] Merrill, D., Garland, M., Grimshaw, A.: Scalable gpu graph traversal. In: ACM SIGPLAN Notices. vol. 47, pp. 117–128. ACM (2012)

[13] Zou, D., Dou, Y., Wang, Q., Xu, J., Li, B.: Direction-optimizing breadth-first search on cpu-gpu heterogeneous platforms. In: High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC EUC), 2013 IEEE 10th International Conference on. pp. 1064–1069. IEEE (2013)

[14] Saule, E., Catalyurek, U.V.: An early evaluation of the scalability of graph algorithms on the intel mic architecture. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. pp. 1629–1639. IEEE (2012)

[15] Gao, T., Lu, Y., Zhang, B., Suo, G.: Using the intel many integrated core to accelerate graph traversal. International Journal of High Performance Computing Applications 28(3), 255–266 (2014)

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

[17] Betkaoui, B., Wang, Y., Thomas, D.B., Luk, W.: A reconfigurable computing approach for efficient and scalable parallel graph exploration. In: Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on. pp. 8–15. IEEE (2012)

[18] Ni, S., Dou, Y., Zou, D., Li, R., Wang, Q.: Parallel graph traversal for fpga. IEICE Electronics Express 11(7), 20130987–20130987 (2014)

[19] Attia, O.G., Johnson, T., Townsend, K., Jones, P., Zambreno, J.: Cygraph: A reconfigurable architecture for parallel breadth-first search. In: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International. pp. 228–235. IEEE (2014)

AUTHORS PROFILE

Guoqing LEI is a PhD candidate in National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha, 410073, China.

Rongchun LI is an assistant researcher in National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha, 410073, China.

Song GUO is a PhD candidate in National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha, 410073, China.

Fei XIA is an associative professor Electronic Engineering College, Naval University of Engineering Wuhan, China