

A Message-Passing Multi-Softcore Architecture on FPGA for Breadth-First Search

Abstract—Breadth-first Search (BFS) is a fundamental graph algorithm. Due to the irregular nature of the memory accesses to graph data structures, parallelization of BFS on cache-based systems leads to poor performance. Many issues, such as memory access latency, cache coherence policy and inter-process synchronization, affect the throughput of BFS on such systems.

We propose a message-passing multi-softcore architecture on FPGA for high performance implementation of BFS. Parallelization of the algorithm is achieved by exchanging information between autonomous softcores on FPGA.

Several optimizations are performed to reduce traffic on the interconnect and enable designs with high clock rate. Implementations of such systems on a Xilinx Virtex-5 XC5VL330, with varying architectural configurations, achieve clock rates in excess of 100 MHz, even when the BRAM utilization reaches 97%. The sustained performance of our system ranges from 160 to 790 Million Edges Per Second (ME/s) on a DDR3 DRAM. This rivals the best performance from other implementations on various computing platforms, including more complex and expensive multi-core architectures, such as Cell BE. For many input data sets, the throughput of our design approaches 800 ME/s, the upperbound due to the DRAM bandwidth limit.

I. INTRODUCTION

Many data structures in real-world applications can be represented as graphs, such as in networking analysis [1], image processing [2], etc. A graph can be stored as an adjacency matrix or an adjacency list in memory. Most of graph problems are solved while traversing the graph, and BFS is one of such fundamental graph traversal algorithms. Its implementation on various computing platforms has been widely studied [3], [4], [5]. Due to the irregular nature of the fine-grained memory access to graph representations, parallelization of BFS on cache-based systems can be a difficult task [6]. While many issues, such as cache coherence policy and inter-process synchronization come into play and make the situation very complicated, a common blame can be pointed to: I/O capacity in the system.

FPGA, as a predominant reconfigurable computing device, can usually complement general purpose processors (GPPs) where they lack ideal solutions. Obtaining functions through configuration after fabrication, FPGA combines the flexibility of software and the high performance of customized hardware design. Therefore, it can outperform GPPs, especially, in applications with a regular structure and/or a great deal of parallelism [7]. Due to recent technology advancements, modern FPGAs contain close to half million of logic cells, 40 Mbits Block RAMs, and a variety of embedded hardware components. It is well documented that FPGAs excel in computation intensive tasks, such as signal processing/communication,

scientific computation, networking security, etc [8], [9], [10], [11]. This paper intends to show FPGA's potential for I/O intensive applications, such as Breadth-first Search.

We propose a multi-softcore architecture on FPGA that enables high performance BFS algorithm implementation. Due to the configurability, FPGA-based designs can supply sufficient resources to the I/O subsystem design, while catering to the demand for computing power for specific graph problems. Our proposed architecture utilizes message passing for parallel processing between cores, including both data transmission and synchronization. Each core, in charge of one partition of the graph, runs the BFS algorithm autonomously and receives necessary information from other cores and external memory. Synchronization can be conducted by each core in an individual and distributed way, which gives birth to a mechanism called “floating barrier” that can reduce the overhead of synchronization barriers. A message consistency policy is, accordingly, defined to guarantee the correctness of the algorithm execution.

Our design on FPGA adopts an classic parallelized BFS algorithm, with optimizations to reduce unnecessary traffic load on the interconnect, and to enable simple design with fast clock rate. The implementation on a Xilinx Virtex-5 can run at more than 100 MHz, when we utilize over 90% of on-chip BRAM to hold as large a graph as possible on the FPGA. The design can achieve throughputs higher than, or comparable to, the results obtained on some of the state-of-the-art chip-multiprocessors, such as Cell BE. For many sample input graphs, our design is approaching the bandwidth limit of DRAMs where the entire graph is stored. Our contributions in this work are as follows:

- To the best of our knowledge, this study is the first multi-softcore architecture on FPGA aiming to achieve competitive performance for I/O intensive applications. Our architecture also enables balancing computing capability with I/O bandwidth according to application situations.
- A variety of design optimization techniques are proposed to improve the BFS performance on this architecture. Some of them, such as the bitmap scheme to reduce traffic and the special message to allow loose barriers, can be applied to implementations on other platforms.
- The implementation on a Xilinx Virtex-5 FPGA achieves throughput close to 800 Million Edges Per Second (ME/s) for the sample graph data sets. This is favorably comparable with most advanced implementations on other chip-multiprocessor (CMP) systems.

The rest of the paper is organized as follows. In Section II, we introduce background on BFS algorithm and some related work. In Section III, we propose the architecture, and a design and its optimization techniques are presented in Section IV. The implementation experience and performance evaluation are reported in Section V. We conclude with a summary and discussion on future work in Section VI.

II. BACKGROUND AND RELATED WORK

A graph $G = (V, E)$ is composed of a set of vertices V and a set of edges E . We define the *size* of a graph as the number of vertices $|V|$. Given a vertex $v \in V$, we indicate with E_v the set of neighboring vertices of v (or neighbors, for short), such that $E_v = \{w \in V: (v, w) \in E\}$, and with a_v the vertex arity, the number of elements $|E_v|$. We denote as \bar{a} the average arity of the vertices in the graph, $\bar{a} = \sum(|E_v|/|V|)$.

Solving a graph problem usually occurs while traversing a graph. Graph traversal is critical to many areas of science and engineering that demand techniques to explore large data sets represented by graphs. In these areas, search algorithms are the computational engines to discover vertices, paths and groups of vertices with desired properties. Among graph search algorithms, Breadth-first Search (BFS) is probably the most common one, and is a building block for a wide range of graph analysis applications.

BFS begins at the start node (*root*), and explores all the neighboring nodes. Then, for each of the neighbor nodes, BFS visits the unexplored neighbor nodes, and so on, until it traverses all the nodes. During the traversal, accessing a usually global data structure, where the graph is stored, is a bottleneck for a processor-based system implementation.

Speeding up BFS has major impact on many applications, and has been a topic for almost every existing computing platform. We believe FPGA-based multi-core architecture has unique ability to obtain superior throughput performance for Breadth-first Search.

A. Classic BFS Algorithm

We present the classic BFS algorithm in this section, and its bulk-synchronous parallelized (BSP) version in the next. They are the base for our study of BFS on FPGA platform.

We formalize the algorithm as following: given a graph $G(V, E)$ and a root vertex $r \in V$, the BFS algorithm explores the edges of G to discover all the vertices reachable from r , and produces a BFS tree rooted at r . Vertices are visited in levels: when a vertex is visited at level l , it is also said to be at distance l from the root. This is shown in Algorithm 1, where Q maintains a list of “frontier nodes” for immediate visiting.

The problem of searching large graphs alone poses difficult challenges, mainly due to the vast search space imposed by the sheer amount of data. However, combined with the lack of spatial and temporal locality in graph data access, the long DRAM latency becomes a main limiter for system performance. When being implemented on GPPs, the on-chip caches may be running out for maintaining the queues. Then access to external memory becomes necessary not only for the global graph data, but also for the intermediate data structures.

```

input : Graph  $G(V, E)$ , root  $r$ 
// Variables definition
1  $level \Leftrightarrow$  exploration level;
2  $Q \Leftrightarrow$  vertices to be explored in the current level;
3  $Q_{next} \Leftrightarrow$  vertices to be explored in the next level;
4  $marked \Leftrightarrow$  boolean array  $marked_i, \forall i \in [1 \dots |V|]$ ;
// Initialization
5  $\forall i \in [1 \dots |V|] : marked_i = false$ ;
6  $marked_r = true$ ;
7  $level \leftarrow 0$ ;
8  $Q \leftarrow \{r\}$ ;
9 repeat
10    $Q_{next} \leftarrow \{\}$ ;
11   for all  $v \in Q$  do
12     for all  $n \in E_v$  do
13       if  $marked_n = false$  then
14          $marked_n \leftarrow true$ ;
15          $Q_{next} \leftarrow Q_{next} \cup \{n\}$ ;
16       end
17     end
18   end
19    $Q \leftarrow Q_{next}$ ;
20    $level \leftarrow level + 1$ ;
21 until  $Q = \{\}$ ;

```

Algorithm 1: Sequential BFS

B. Parallelized BFS Algorithm

Parallelizing BFS algorithm on CMPs has been a subject of several studies [12], [6], where the reported performance ranges from sub 100 up to 800 ME/s with a variety of machine specific optimizations. One such parallel BFS algorithm implementation is presented in Figure 1 [6]. In this implementation, the input graph G is represented as an adjacency array, a widely used data structure for graphs [13]. Assuming there are N vertices in G , the adjacency array consists of a vertex array and up to N neighbor arrays. The vertex array has N elements, each storing the basic information of a vertex in G , such as the ID of the vertex, the flag showing if the vertex has been visited, the owner of the node, the link to the corresponding neighbor array and the size of the neighbor array. The neighbor array of a vertex v consists of the IDs of vertices adjacent to v .

As shown in Figure 1, the graph is statically partitioned in P disjoint subsets, where P is the number of cores. Each core i runs a thread i , and is the *owner* of the subset V_i , where $i = \{0, 1, \dots, P - 1\}$. No special algorithm is applied during the partitioning to generate locality in the subsets' data representation. When executing, thread i maintains, locally, one ready queue Q_i and P outgoing queues $Q_{i,j}$, where $j = 0, 1, \dots, P - 1$, denoting the destination cores of the outgoing queues. Thread i gathers all neighbors from vertices in Q_i and dispatches them into appropriate outgoing queues. Here, a barrier needs to ensure the dispatching is complete before all threads access the data through interconnect. After

the barrier, the core j receives vertices from $Q_{i,j}$. It then marks those unmarked and puts them into Q_j for the next level of exploration. The second barrier in the figure enforces the control dependency, i.e. the level-by-level exploration.

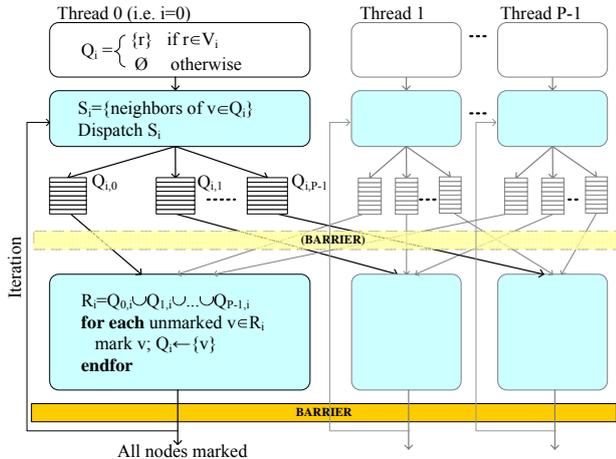


Fig. 1. A parallel BFS

When being realized on a general purpose CMP, the performance not only suffers from the problems presented in the sequential implementation, but also from the communication and synchronization overhead occurred during the level-maintaining for each cores in the system [6]. This diagram of BFS gives a glimpse of performance blockades when implemented on CMPs. While the Q_i s and $Q_{i,j}$ s can be locally maintained by a core i , the graph partition V_i may go to external memory, due to the size limitation of on-chip cache on CMPs. Moreover, the data collection stage from all outgoing queues demands high throughput from the interconnect. The barrier can also be a costly operation on general purpose processors, since it is usually implemented using global storage and is serially accessed.

C. Related Work on FPGA

There are also many studies of BFS on FPGA. An early approach to implementing graphs on FPGAs is part of the RAW project [14]. RAW's graphs are directed and stored in the FPGA by building a circuit that resembles the graph. Vertices are represented by logic units, and edges are wires connecting the vertices. HArdware Graph ARray (HAGAR) [15] at Bell Labs maps the adjacency matrix representation of a graph to reconfigurable hardware. These approaches have difficulty in updating a graph and/or lack of scalability. GraphStep [16], which gave a general high level abstract of graph computing model based on object, assumes on-chip processing of sparse graphs, and does not focus on FPGA architecture and performance optimization.

III. MULTI-SOFTCORE ARCHITECTURE FOR BFS ON FPGA

FPGA-based solutions can alleviate the hurdle imposed by the access to external memory, as well as communications

needed by data exchange and thread synchronization among the cores. We design a multi-softcore architecture, adopting the algorithm implementation shown in Figure 1.

A. Architecture Overview

The organization of our architecture is a group of P cores connected through an interconnect, as shown in Figure 2. A separate DRAM interface connects the cores to external memory, e.g. DRAM(s), where the entire graph is stored. The cores, interconnect, and DRAM interface are on chip.

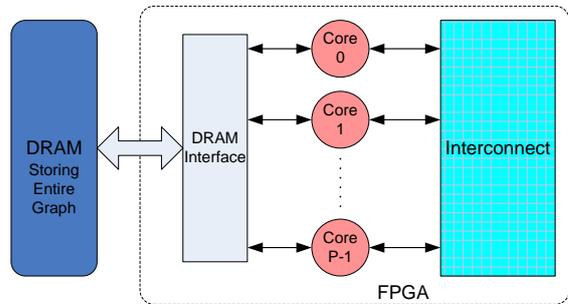


Fig. 2. Architecture overview

A core in our architecture has local memory, which stores two particular types of information. One is for the subset of the vertices in the core's partition, including flags for whether the vertices are visited; the base addresses for accessing the vertices' neighbor lists in DRAM; and the lengths of these neighbor lists. Another storage is a data buffer to temporarily hold both the vertices ready to explore and the neighbors to be sent out.

B. Message Passing Parallel Processing

Our architecture uses message passing through the interconnect to exchange information among the cores. Note that the cores share the external memory to only read the graph data; therefore, no race condition occurs. The interconnect messages include two basic types: (1) vertex information, which may consist of source core ID; destination core ID; and vertex ID, etc; (2) Synchronization information, i.e. barrier markers for the BFS algorithm. New types of messages can be added based on specific designs.

The number of messages transmitted may demand high bandwidth from the interconnect. However, the aggregate traffic is linearly related to, or limited by, the bandwidth of external memory. This is because the vertices being exchanged in interconnect are all read from DRAM(s), which are bounded by $|E|$. The number of barrier messages is $O(P \log N)$, where, usually, $N < |E|$. Hence, the system performance depends on the utilization of DRAMs, provided that the interconnect's bandwidth is adequate.

C. Structure of a Core

As shown in Figure 3, a core receives from and sends messages to the interconnect. If a received message has vertex information, it goes to vertex array, to see whether the

vertex has been visited; otherwise, the message is sent to the synchronization control. When a vertex is not visited yet, the access control uses the information for address pointer and length of neighbor list from the vertex array to access DRAMs. The returns from DRAM are the neighbors of this vertex. The neighbors are then buffered to wait for output.

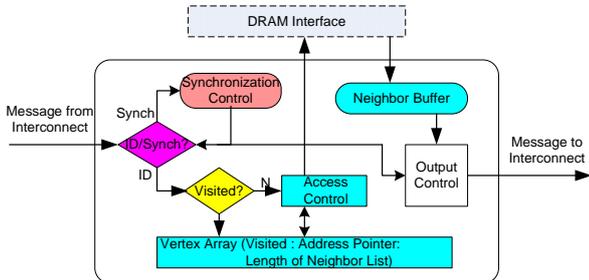


Fig. 3. Single BFS core architecture

The output control is responsible for sending vertex messages to their respective owners and sending barrier messages to all cores in the system for synchronizations. The synchronization mechanism will be elaborated in the next section when the operation of BFS is explained.

D. Operation of BFS and Realization of Barriers

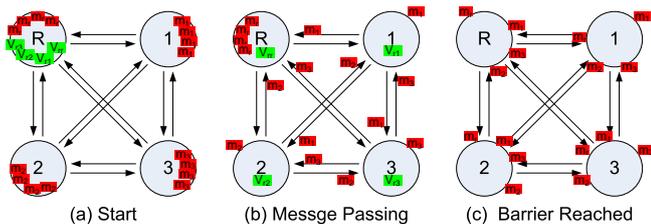


Fig. 4. Operation of BFS on multi-softcore architecture

We illustrate the BFS operations on our architecture, and explain how the synchronization works, using a 4-core system in Figure 4. Note that we do not show cycle-by-cycle operations of the algorithm, nor the memory access latency here. Suppose that a vertex is designated as the root for the BFS, and the core that owns this root is the root core, denoted R in the figure. In this process, m_i and m_r represent barrier markers from core i or R respectively, and $v_{r,j}$ is a vertex message sent from root core R to core i .

The root core starts by reading the neighbor list of the root, and then sends messages carrying the neighbor vertices' information to their respective owners via communication links. To determine a vertex's owner core, the root core checks the vertex ID according to the partition method. For example, if we evenly distribute the ID number space to the cores, we can check which range the vertex ID falls into to find its owner core. The lines with an arrow end shown in Figure 4 are not real connections, but for the purpose of demonstration only, indicating feasible message transmission between the

cores supported by the underlying interconnect. The beginning stages of BFS are shown in Figure 4 (a). During this time, the cores, other than the root, send barrier markers out. Note that in a core, the same barrier marker is sent to all cores in the same cycle by the output control module.

As stated before, BFS performs on a level-by-level basis. So, the root vertex is at *Level 0*, and all its neighbors are at *Level 1*. The root core starts in *Level 1*, while non-root cores advance to *Level 1* after sending out the first batch of markers. When messages arrive at their destination, the vertex messages are processed first. As shown in Figure 4 (b), the $v_{r,j}$ s enter the cores, while the barrier markers hold in buffers, i.e. the communication links. The on-hold marker messages prevent the following vertex messages in the same channels, which are not shown in this figure, from entering their destination cores. Note that at this moment, the barrier markers from the root core have not been sent.

Later in Figure 4 (c), the barrier markers from the root arrive at their destinations. A core counts received barrier marker messages to decide if a barrier is reached, which occurs when the markers from all cores in the system arrive at this core. The core then informs the synchronization control to advance to *Level 2*, and it then prepares to send a new round of marker messages after the neighbors buffered for *Level 1* are sent out. This process continues in a loop until the BFS is complete. For simplicity, we omit the operations where cores 1, 2 and 3 send the buffered neighbors out.

a) *Distributed Synchronization*: From the above description, we learn that a core reaches its barrier when it gets barrier markers from all other cores. Each core decides its own synchronization based on this condition, and a core can arrive at the barrier at a different time than other cores, a phenomenon we call "floating barrier."

b) *Message Consistency Policy*: The correct operation of barriers is guaranteed if a vertex message is always processed before its ensuing barrier marker message from the same sending core. This constraint is called message consistency policy in our architecture. To ensure this occurrence, a core should send a barrier marker to another core only after it sends out all neighbor vertices in the current level that belonging to that core. The core on the receiving end should process vertices in its current level only, and then process the corresponding marker message.

IV. DESIGN AND OPTIMIZATION

We present a design of our architecture on FPGA and introduce several optimization techniques to improve system performance. Our design has all the cores connected with each other through communication channels to form a complete graph. The communication channels are built using FIFOs on FPGA, so that a message can be buffered there before entering a core. Especially, a barrier marker can hold the following messages from that channel in the FIFO. When we have P cores in the system, there will be P^2 channels for bi-directional communication between each pair of cores. A core, therefore, has P input ports and P output ports, each

serving one neighboring core only. The core sends messages to itself through a channel, including both vertex and barrier marker messages. This scheme enables easier compliance with the message consistency policy, since the vertex messages can wait in FIFOs for a preceding barrier marker to be processed.

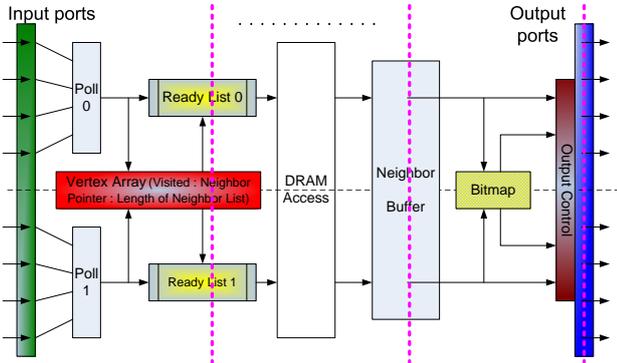


Fig. 5. Single core design

The core design illustrated in Figure 5 adopts an approach similar to pipelining. Although no actual pipeline stage registers are employed, the storage devices, such as Ready Lists, Neighbor Buffer and channel FIFOs, have registers on their output, therefore, serving as pipeline registers. The pipeline stages are indicated using dashed lines in the figure. Note that the DRAM Access takes varying amounts of time, before returning data from the external memory to the neighbor buffers. The working regime of a core is as follows:

- Input ports are scanned for an incoming message. When a vertex message is found, it is checked in Vertex Array to see whether it has been visited. A visited vertex is discarded, but for an unvisited vertex, its entry in Vertex Array is updated, and the output from Vertex Array is pushed into Ready List.
- If a barrier marker is found, it is counted until the total number of barrier markers is equal to the number of cores in the system. At this time, a barrier is reached, and a marker message is created and then enters Ready Lists.
- Entries in Ready Lists, carrying such information as address pointer and length of neighbor list, are used to access DRAMs. The returned neighbors from DRAMs are written into Neighbor Buffer, waiting to be output. A barrier marker enters Neighbor Buffer automatically.
- The vertices from Neighbor Buffer are checked against Bitmap (see Section IV-A) for whether they have been sent from the core before. If not, they are sent to an appropriate output port by the output controller. However, a barrier marker will be sent to every output port.

A. Bitmap to Reduce Interconnect Traffic

A vertex may appear in a core’s neighbor lists multiple times, due to the fact that it may be neighbor to multiple vertices in the core’s graph partition. During the BFS, these vertices could be sent output more times than is necessary. This redundancy can be seen in Figure 6. We divide a graph of one

million vertices with an arity of 16 and topology of *Neighbor*, *Bipart* and *Rand* (see Section V-B1 for the definitions), onto multi-softcore systems with 4, 8 and 16 cores. Vertex redundancy is measured by the surplus appearances of a vertex in cores’ neighbor lists. We can see that the more cores, the less the redundancy occurs.

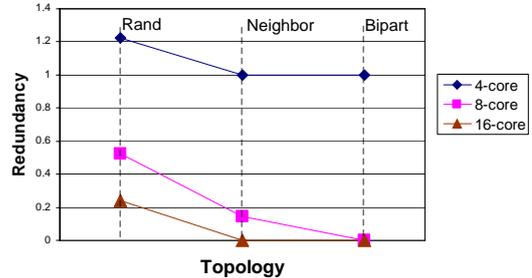


Fig. 6. Redundancy in a core’s graph partition

Based on the above observation, we design a bitmap scheme to reduce the redundancy of traffic in communication links. Shown in Figure 5, a bitmap is used to record whether a vertex has been sent out by the core before. If not, the vertex is sent to an output port, and the bitmap is updated.

B. Dual Processing Unit in One Core

It is shown in Figure 5 that the polling modules and Ready Lists are duplicated in a core. Utilizing dual-ported BRAMs on FPGA, we design a dual processing scheme for the BFS to improve throughput performance. The polling units, each scanning one half of input ports, write results into its respective Ready List. The other local memories, such as Vertex Array, Neighbor Buffer, and Bitmap, need only one copy, and they should be capable of supporting two accesses at the same time.

C. Port Polling Schedule

Having a “perfect” schedule for input port polling may reduce the likely backlog in communication channels, and it may also enable better utilization of the duplex processing units by balancing their workloads and keeping them equally busy most of the time. However, the performance of scheduling algorithm strongly depends on graph topologies as well as the location of root vertices; thus, it is futile to look for that scheduling discipline.

In our design, we choose to use a polling scheme similar to round robin. Starting from one input port, a processing unit of a core checks its half for the first port with an input of a **vertex** message. If this vertex has not been visited, it enters the corresponding Ready List. Otherwise, it is discarded, and the processing unit moves on to the next cycle of polling, starting from where it left off. This process repeats itself until the BFS completes.

V. IMPLEMENTATION AND PERFORMANCE

A. Implementation on FPGA Platforms

We evaluated our architecture on a Xilinx Virtex-5 XC5VL330. It is an FPGA for general logic applications with

a large amount of logic resources at 207360 LUT-FF pairs and mid-sized BRAMs at 10368 Kbits. Based on Figure 5, the whole Bitmap and part of the Vertex Array, i.e. visited, require “write” operations in dual-port mode, so it is necessary to store them in BRAM. Other demands for storage can be met by either BRAMs or distributed RAMs. We evaluated our design with 4- and 8-core configurations, all supporting a graph of 256K vertices with the arity of 16. The resource consumption and timing performance are reported in Table I, excluding an implementation of a DRAM controller that can be on or off the chip of FPGA.

TABLE I
RESOURCE CONSUMPTION AND TIMING PERFORMANCE OF DESIGNS ON
FPGA

# of Cores	LUT-Flip Flop Pairs	Block RAMs	Clock Rate
4	5286	233	211 MHz
8	11599	280	129 MHz

Compared with the consumption of logic fabric, the utilization of BRAM is very high, which shows that our architecture relies on the availability and configuration of memory resources to achieve high performance. If we constrain our design on the FPGA, the size of the design, e.g. the number of cores, the size of supported graphs, is to be decided by the amount of BRAMs on the FPGA. However, we can employ off-chip SRAMs to support more cores and larger graphs while retaining performance for the designs. For the purpose of evaluating the resource consumption and timing performance of a design, we only implemented designs that can fit on the chip here. Our results show that a large design utilizing almost all on-chip BRAMs can work at clock rates over 100 MHz.

B. Simulation of the Multi-Softcore Architecture

1) *Features of Sample Input Graphs*: To systematically study the performance of BFS on our multi-softcore architecture, 3 types of graphs with various topologies were used as inputs. Illustrated in Figure 7, they are (1) General graphs, denoted as *Neighbor*. The vertices were arranged as shown in (a). Each vertex, marked by the numbers in the squares, had edges connecting to its left and right neighbors. (2) Bipartite graphs, denoted as *Bipart*. The vertices were arranged in two sets as shown in (b), where each vertex had edges connecting to a given number of vertices with adjacent IDs in the other set. (3) Random graphs, denoted as *Rand*. Each vertex has edges connecting to a random subset of the vertices. Figure 7 (c) illustrates one such graph. *Neighbor* and *Bipart* have relatively good locality, as the neighbors of the vertices are close to each other.

2) *SystemC Modeling and Verification*: We utilized SystemC, one of the Electronic System Level languages (ESLs), to model and verify the functionality and evaluate design options of our multi-softcore architecture for BFS. SystemC is a class library built upon standard C++, and it can model large-scale digital systems with cycle accuracy [17]. Its event-driven simulation kernel can support thread-like execution of many functional definitions inside a design component. The

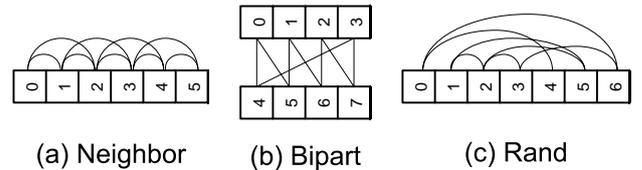


Fig. 7. Topologies of sample graphs

executables run on top of OS, and can be 10-100 times faster than other simulation techniques. C/C++ programming also adds the ability to handle large amounts of data.

We simulated our architecture with the design described in Section IV for systems of 4, 8, and 16 cores respectively. A DRAM interface, seen in Figure 2, connects every core through an individual hardware channel to DRAM(s). The DRAM interface polls the cores in round-robin fashion, similar to the port polling used in a core, for their DRAM requests, and queues the requests in order to access the external memory. We assume one DRAM (and its controller) is employed only to obtain the lower bound of the performance for our design, and facilitate the comparison with other works on different platforms.

We utilized the DRAM simulator used in [18]. To evaluate our BFS architecture, we set the DRAM to be a DDR3, Micron MT41J128M8. This DRAM has 8 banks with a burst length of 8. The DRAM runs at 400 MHz, supporting 800 Mbps data transmission per wire. Our core is set up to run at 100 MHz, which can be easily achieved on FPGA as shown in Section V-A. One neighbor information is delivered by the DRAM during one data transfer.

C. Experimental Results and Discussion

The aforementioned sample graphs are input to our SystemC model for architectural verification and design space exploration. We vary the size of the sample graphs from 1K to 8K to 64k vertices, and their arities also are chosen from 4, 8, and 16. Therefore, a test case in our experiment has 4 variables, i.e. the number of cores, the topology, size and arity of the input graphs. We measure the throughput performance through the number of edges visited in a unit of time, as Million Edges Per Second (ME/s), as well as the resource consumption of each important architectural components, such as depth of channel FIFO, size of ready list and neighbor buffer, etc. We also look into the hardware resource consumption of DRAM interface to provide a comprehensive understanding of resource demand on FPGA from our BFS architecture.

We first show the throughput performance of our design on different graph topologies. This measurement is limited by available DRAM bandwidth as we explained before. Based on our simulation setup, this upperbound is 800 ME/s. We present the results for both *Neighbor* and *Bipart* in Figure 8. The X-axis is the arity of the graphs, which is either 4, 8, or 16, and the Y-axis is the computed throughput. One can see that the throughput ranges from 160 to almost 800 ME/s, depending on the topology and arity of the graphs more than

the number of cores and the size of the graphs. When the arity is larger than 8, the throughput is between 500 and 795 ME/s , compared favorably with other attempts on BFS [6], [4]. The low throughput for low-degreed graphs is expected because the burst length of the DRAM is 8; thus, much of the DRAM bandwidth is wasted, and it, in turn, limits the utilization of cores. The reason that the graph size did not matter for these results is due to the regular distribution of neighbors for each vertex in these two synthetic graphs. When a graph is large enough, the access pattern to DRAM is likely to repeat, which renders the size not to be a factor in system performance.

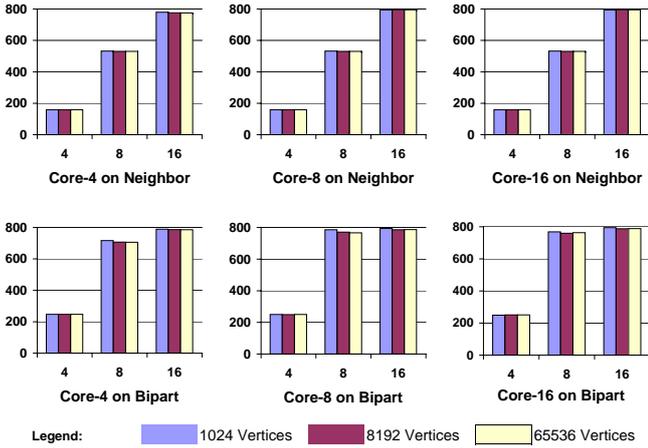


Fig. 8. Throughput measurement for graphs with *Neighbor* and *Bipart* topology

The throughput results for *Rand* graphs, shown in Figure 9, share some similarity with the first two topologies, such as the correlation of throughput with the arity of the graphs, etc. However, as the size of the graphs grows, the measured throughputs are reducing. This is due to the random nature of this topology resulting in more waste of DRAM bandwidth.

Note that for some test cases, the throughputs are approaching the 800 ME/s limit set by DRAM. This is reasonable because we use DRAM solely for the purpose of reading, and the 8-bank configuration of DRAM also allows us to open rows concurrently to avoid long latency, thus, greatly improving the bandwidth utilization.

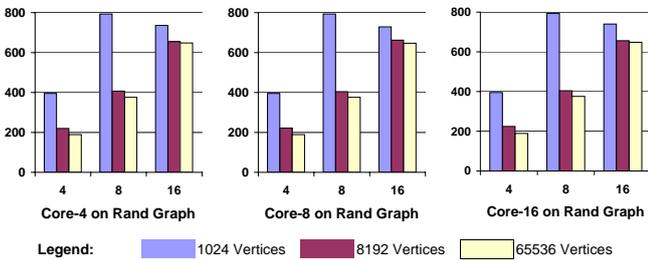


Fig. 9. Throughput measurement for graphs with *Rand* topology

A major component in our design is the channels connecting each pair of the cores. For both *Neighbor* and *Bipart* graphs,

the regular distribution of vertices' neighbors brings great load balancing to the cores, and the resulting BFS tree also has large depth, which translates into fewer vertices in each level. As a result, the channel's buffer consumption is low and evenly distributed among all cores. Our simulation shows that the buffer usage of each channel is between 2-4 for all test cases using *Neighbor* and *Bipart* graphs. However, in the case of *Rand* graphs, the imbalance of load on the cores during the BFS process and shorter BFS tree, meaning more vertices per level, leads to the buffer "explosion" as shown in Figure 10. Here, the X-axis indicates the number of cores in the systems and the size of input graphs, and the size of buffers varies from about 10 up to 250. It also shows that the buffer usage grows along with the graph size. When the number of cores increases, the buffer size of individual channels decreases accordingly. However, since the number of channels is quadratic of the number of cores, the total buffer consumption still rises. Generally, the buffer size also increases as the arity of a *Rand* graph becomes higher.

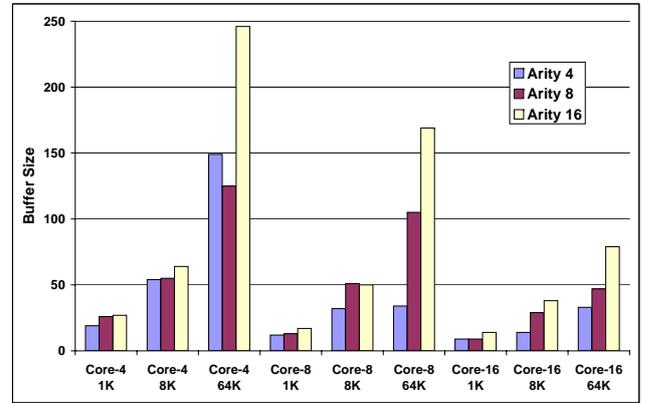


Fig. 10. Channel buffer usage for *Rand* graphs

Within a core, another infrastructure, the ready list, shares similar property with the channels, in that the buffer consumption of a core's ready list for *Neighbor* and *Bipart* is modest, but for the *Rand* graphs, it becomes very large. Comparatively, the neighbor buffer in a core does not consume much storage. The simulation shows that this feature is insensitive to all parameters of the experiment, except that it grows roughly linearly with the arity of the graphs.

Since the DRAM interface module is placed on the same FPGA chip with the cores and channels, its resource consumption needs investigation. Inside the interface, a state machine polls the input ports in round-robin fashion to find valid requests from the cores, and then queues the requests in a central queue to access the DRAM controller later. The resource consumption of the central queues for *Neighbor* and *Bipart* graphs are shown in Figure 11 (a). We present only the results for the system with 8 cores running on a graph of 8K vertices for both topologies, as the results for these two topologies depend mostly on the arity of a graph, and its indiscrimination of all other system and input graph

parameters. We can see the queue size is positively correlated to the arity of a graph. However, the amount of memory consumption is modest, even for very large graphs.

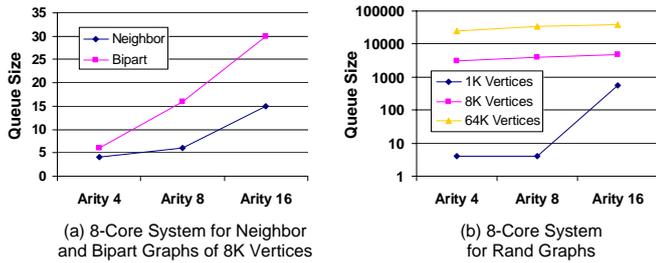


Fig. 11. Resource consumption of DRAM central queue

To the contrary, the results for *Rand* graphs present significant differences. We plot the results in Figure 11 (b) for the system with 8 cores only, as the results for 4-core and 16-core systems are similar to the 8-core one for this study. Note that the figures (a) and (b) have different legends, and the Y-axis in (b) is logarithmic. When the size of a graph becomes larger and larger, the size of the queue also grows. The demand for buffer storage is independent of the number of cores, and it is only slightly dependent on the arity of input graphs.

The results show that our design, targeting substantially large graphs, consumes small amount of logic resource. Even with 97% of BRAM utilization, the implementation still can run at a clock rate of 100 *MHz* or more, and achieve superb throughput performance, ranging from 160 to 790 *ME/s*. This is favorably comparable to the work on CMPs and other parallel computing platforms, including some of the state-of-the-art processors, such as Cell BE, as shown in Table II.

TABLE II
PERFORMANCE COMPARISON WITH STATE-OF-THE-ART BFS
IMPLEMENTATIONS

Solutions	DRAM Bandwidth	Maximum Throughput
Our design	800 <i>Mpbs</i>	790 <i>ME/s</i>
AMD Opteron 2350 [6]	1.33 <i>Gbps</i>	311 <i>ME/s</i>
Cell BE [4]	3.2 <i>Gbps</i>	787 <i>ME/s</i>

VI. CONCLUSIONS

This paper presented a message-passing multi-softcore architecture on FPGA for breadth-first search on a graph. By availing sufficient communication bandwidth, our architecture can overcome synchronization hurdle while achieving optimized I/O performance. Using message passing to implement barriers enables the cores in our architecture to make synchronization decisions in an individual and distributed way. This scheme reduces the overhead incurred on cache-based systems. Utilizing proposed optimization techniques, our architecture can be realized with high operating clock rate for designs on FPGA. The achieved throughput performance is comparable to the solutions on other state-of-the-art multicore systems, while consuming only a small fraction of logic resources on FPGA. Our future work will study the effects of deploying

more DRAM modules on the system performance. We can also explore the design space when a hierarchical interconnection network is adopted to alleviate the problem when many more cores are added into the system, and saving resources used for interconnectivity become imperative.

REFERENCES

- [1] J. A. Barnes and F. Harary, "Graph theory in network analysis," *Social Networks*, vol. 5, no. 2, pp. 235–244, 1983.
- [2] Z. Wu and R. Leahy, "An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, pp. 1101–1113, 1993.
- [3] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 523–530.
- [4] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, pp. 1381–1395, 2007.
- [5] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on blugene/1," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 25.
- [6] Y. Xia and V. Prasanna, "Topologically Adaptive Parallel Breadth-First Search on Multicore Processors," in *Proc. of Parallel and Distributed Computing and Systems (PDCS 2009)*, Cambridge, Massachusetts, USA, November 2009.
- [7] N. Tredennick and B. Shimamoto, "Reconfigurable systems emerge," *Proceedings of the 2004 International Conference on Field Programmable Logic and Its Applications*, vol. 3203, pp. 2–11, 2004.
- [8] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, pp. 1432–1442, 2003.
- [9] M. Gokhale, D. Dubois, A. Dubois, M. Boorman, S. Poole, and V. Hogsett, "Granidt: Towards Gigabit Rate Network Intrusion Detection Technology," in *Proc. of 12th International Conference on Field-Programmable Logic and Applications*, Montpellier, France, September 2002, pp. 404–413.
- [10] K. Masselos, A. Pelkonen, M. Cupak, and S. Blionas, "Realization of wireless multimedia communication systems on reconfigurable platforms," *J. Syst. Archit.*, vol. 49, no. 4-6, pp. 155–175, 2003.
- [11] L. Zhuo and V. Prasanna, "High Performance Linear Algebra Operations on Reconfigurable Systems," in *Proc. of SuperComputing 2005*, Washington, USA, November 2005.
- [12] O. Villa, D. P. Scarpazza, F. Petrini, and J. F. Peinador, "Challenges in mapping graph exploration algorithms on advanced multi-core processors," in *IPDPS*, 2007, pp. 1–10.
- [13] D. K. Blandford, G. E. Blelloch, and I. A. Kash, "An experimental analysis of a compact graph representation," in *In Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments*, 2004, pp. 49–61.
- [14] J. W. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures," *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, vol. 2914, no. 1, pp. 225–236, 1996.
- [15] O. Mencer, Z. Huang, and L. Huelsbergen, "Hagar: Efficient multi-context graph processors," in *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 2002, pp. 915–924.
- [16] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon, "Graphstep: A system architecture for sparse-graph algorithms," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 143–151, 2006.
- [17] Open SystemC Initiative (OSCI), "http://www.systemc.org/home/."
- [18] Removed for double blind review.