Parallel graph traversal for FPGA

Shice Ni\textsuperscript{a)}, Yong Dou, Dan Zou, Rongchun Li, and Qiang Wang

National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha, China
\textsuperscript{a)} nishice@gmail.com

Abstract: This paper presents a multi-channel memory based architecture for parallel processing of large-scale graph traversal for field-programmable gate array (FPGA). By designing a multi-channel memory subsystem with two DRAM modules and two SRAM chips and developing an optimized pipelining structure for the processing elements, we achieve superior performance to that of a state-of-the-art highly optimized BFS implementations using the same type of FPGA.

Keywords: FPGA, multi-channel memory, parallel graph traversal

Classification: Electron devices, circuits, and systems

References

[1] J. W. Babb, M. Frank and A. Agarwal: SPIE Photonics East (1996) 225.
[2] A. Dandalis, A. Mei and V. Prasanna: PDP (1999) 652.
[3] O. Mencer, Z. Huang and L. Huesbergen: FPL (2002) 915.
[4] Q. Wang, W. Jiang, Y. Xia and V. Prasanna: ICFPT (2010) 70.
[5] B. Betkaoui, Y. Wang, D. B. Thomas and W. Luk: ASAP (2012) 8.
[6] D. Chakrabarti, Y. Zhan and C. Faloutsos: ICDM (2004) 442.
[7] C. M. Richard and B. Kyle: Cray User’s Group (2010).

1 Introduction

Graph is an important representation of large data sets. Graph traversal algorithms, such as breadth-first search (BFS), have data-driven computations dictated by the property of the graph, requires fine-grained random memory accesses with poor spatial and temporal locality and leads to execution times dominated by memory subsystems.

A field-programmable gate array (FPGA) is a reconfigurable integrated circuit. Combining the flexibility of software and the high performance of customized hardware design, FPGA can offer superior performance for many specific applications. Many studies have attempted to implement the graph traversal algorithm on FPGAs. The early methods either build a circuit that resembles the graph or use low-latency on-chip memory resources to store the entire graph \cite{1,2,3}, but failed to adapt the real world graphs, which are too large to fit into on-chip random-access memory (RAM) of FPGAs. Several recent publications \cite{4,5} described strategies of graph traversal on FPGAs.
using off-chip DRAM memories to adapt the traversal of large-scale graph instances, and Betkaoui’s work [5] is the first FPGA-based BFS implementation that can compete with other high performance multi-core systems.

In this paper, we present a multi-channel memory based architecture for single FPGA chip that aims to achieve high performance for large-scale graph exploration. We use two dynamic random-access memory (DRAM) modules to increase the throughput of the memory subsystem and two static random-access memory (SRAM) chips to improve the random write/read performance on small data set. Moreover, we develop an optimized pipelining structure for the processing elements (PEs), which supports generating memory access requests at all pipeline stages, thus making it possible to saturate the memory system using fewer PEs with higher core frequency. The experimental results show that our implementation is capable of obtaining superior performance to that of a highly optimized BFS implementations using the same type of FPGA.

2 Graph representation and BFS algorithm

Sparse matrices are a general way to represent the graphs. For a graph $G(V, E)$, the sets $V$ and $E$ are respectively comprised of $n$ vertices and $m$ directed edges. A directed edge in the graph from $v_i$ to $v_j$ corresponds to a nonzero element at location $(v_i, v_j)$ in a sparse matrix. The compressed sparse row (CSR) representation is the most general storage format for storing the sparse matrix. This format is comprised of two arrays: column indexes array $col$ and row pointers array $ptr$. The $col$ array contains the corresponding column index for each nonzero element in the sparse matrix, arranged in a raster order starting with the upper-left and continuing from left to right then top to bottom. The row pointers array $ptr$ stores the locations within $col$ where each row starts and terminated with the number of nonzero elements.

Given a graph $G(V, E)$, the BFS algorithm is to traverse the vertices of $G$ in breadth-first order starting at source vertex $v_s$ to build a BFS tree. Each newly-discovered vertex $v_i$ is marked by its level in the BFS tree, which equals to its distance from $v_s$. The source vertex is at level 0, and its neighbors are at level 1, and so on. The parallel BFS algorithm is usually implemented in a level-synchronous way, in which each BFS level is processed in parallel while the sequential ordering of levels is preserved.

3 Multi-channel memory based architecture for BFS

We design a multi-channel memory based architecture that adapts the large-scale graphs and fully utilizes the parallelism on FPGA for high-performance BFS implementation. Our design is implemented on a self-designed reconfigurable algorithm accelerator which resides on a host server. The architecture of our system is shown in Fig. 1. The basic organization of the accelerator is a FPGA chip, a multi-channel memory subsystem and a PCI-E interface to the host server. The memory subsystem of the proposed system consists of four independent memory channels, where two DRAM channels connect to
two external DRAM modules (DRAM0/DRAM1) and two SRAM channels connect to two external SRAM chips (SRAM0/SRAM1). The accelerator receives the graph in CSR format and root vertex, executes the BFS algorithm and sends the data structure containing the BFS tree to the host. The BFS algorithm is executed on the PE array, which run in parallel in a multi-threaded fashion. Each PE in our architecture has local memory, which is mainly used as data buffers to temporarily hold the data to and from external memory.

For our multi-channel memory based system, a key decision is to choose the memory type for the data structures of BFS implementation. Our main selection criterions are the sizes and access patterns of the data set. There are mainly three data sets, the graph in CSR format, a BFS level array and a visitation status bitmap. The graphs from real world are usually hundreds of Megabytes (MBs) in size, so we choose DRAM memory to store the graph. The BFS level array \( L \) is used to store the structure of BFS tree, which is usually tens of MBs is size, so it is also stored in DRAM memory. The visitation status of all vertices are stored in a bitmap, which is usually several MBs in size and requires frequent random memory accesses, so we use SRAM memory to store the bitmap data. As the DRAM access takes varying amounts of time before it returns data from the off-chip memory, on-chip RAM resources are mainly used as data buffers to store data from off-chip DRAM memories, including the offset addresses for accessing the vertex neighbors and the vertex levels that are ready to be updated.

The general BFS framework for our FPGA-based BFS implementation mainly consists of three steps:

 Initialization of the data structures. The host processor initializes the BFS level array \( L \), the visitation status bitmap array \( B \) and the BFS level indicator \( \text{curLevel} \). Then, the set of vertices \( V \) of graph \( G(V, E) \) is evenly partitioned into two subsets \( V_0 = 0, 1, \ldots, \frac{|V|}{2} \) and \( V_1 = \frac{|V|}{2} + 1, \ldots, \frac{|V|}{2} - 1 \). Correspondingly, \( G, L \) and \( B \) are respectively partitioned into \( G_0/G_1, L_0/L_1 \) and \( B_0/B_1 \). After that, \( G_0/L_0 \) and \( G_1/L_1 \) are respectively sent to DRAM0.
and DRAM₁. \( B_0 \) and \( B_1 \) are respectively sent to SRAM₀ and SRAM₁.

**Concurrent computation on PE array.** After the initialization of the memory subsystem, asynchronous execution of the BFS kernel takes place on each PE for a given BFS level. Each PE explores a vertex subset of \( V \). Once a PE has explored all the vertices in its subset, it sends a termination flag signal to the PE array controller, indicating whether there had been any vertices marked for the next BFS level.

**Synchronization on PE array.** Each PE waits for all the other PEs to finish their assigned vertices. The termination is reached when there are no marked vertices for next BFS level. At that time the PE array controller issues a termination signal to the host processor to indicate that the execution is completed. After that, the host processor receives the computation result from DRAM₀ and DRAM₁.

### 4 Processing element design

The execution time of the BFS algorithm is dominated by memory access latency. Therefore, the PE is likely to be idle for most of the time while waiting for data from the memory subsystem. To take the advantage of the capability of the multi-channel memory subsystem, our PE design is based on a pipelined architecture to access the off-chip memory subsystem in parallel. Multiple concurrent memory requests can be issued from different pipeline stages of each PE, leading to superior memory access performance. For efficiently taking advantage of the high bandwidth and the parallelism of the multi-channel memory subsystem, multiple concurrent memory requests are issued from different pipeline stages of each PE and kept in flight at any given time. Fig. 2 presents an overview of the PE design for the BFS kernel. The PE design consists of four pipeline stages, which are indicated using dashed lines.

![Fig. 2. Processing element architecture](image)

The algorithm process for each PE is described as follows:

**Read vertices to be explored (S1).** For each vertex \( v_i \) of the PE’s vertex subset, this processing stage reads the BFS level \( L[v_i] \) from DRAM memory in batches into the local buffer \( Buf_{-1} \), using multiple non-blocking memory requests. This process is repeated until all vertices of the PE’s vertex subset have been processed.
Row index gathering (S2). This processing stage reads \( L[v_i] \) from Buf_1. If \( L[v_i] \) is equal to the current BFS level, it means that the vertex belongs to this level, and hence its neighbors will be explored in the current iteration; otherwise, the PE check the BFS level of the next vertex \( v_{i+1} \). For each vertex \( v_i \) to be explored, the row index pair \( ptr[v_i] \) and \( ptr[v_i + 1] \) are retrieved from DRAM memory into local buffer Buf_2. This process is repeated until Buf_1 is empty and processing stage S1 is completed.

Neighbor vertex gathering (S3). This processing stage reads each row index pair \( ptr[v_i] \) and \( ptr[v_i + 1] \) from Buf_2. The neighbors of vertex \( v_i \), which are \( col[ptr[v_i] : ptr[v_i + 1]] \), are retrieved from DRAM memory in batches into local buffer Buf_3. This process is repeated until Buf_2 is empty and processing stage S2 is completed.

Status look-up and update (S4). This processing stage reads each gathered neighbors \( v_i \) from Buf_3 and checked its visitation status by reading status bitmap from SRAM memory. Unvisited vertices will have their distance value updated to the current BFS level plus one. Status bitmap is also updated accordingly. This process is repeated until Buf_3 is empty and processing stage S3 is completed.

5 Implementation and performance evaluation

We evaluated the proposed architecture on a self-designed FPGA-based reconfigurable algorithm accelerator, which is composed of one Xilinx Virtex5 XC5VLX330 FPGA, two Kingston KVR 667D2S5/2G 2 GB DRAM modules, two Samsung K7N323601M-QC20 4 MB SRAM chips (there are three SRAM chips on our board, among which one chip is used to store the configuration bits of the FPGA) and a PCI-E interface (implemented by a small-scale Virtex5 XC5VLX50T FPGA) to the host computer.

Table I presents the configuration details of the platforms used in our experiments and in the previous work.

| Table I. Specification of platforms used in our performance comparison |
|-----------------------------|-------------|-----------|-------------|--------------|
| Platform        | Frequency | Core num. | Memory size | DRAM bandwidth |
| ASAP12-1F [5]   | 75 MHz    | 512       | 16 GB       | 80 GB/s       |
| Ours            | 160 MHz   | 32        | 4 GB        | 12.8 GB/s     |

The graph we used in this paper is the scale-free graphs, representing the common features observed in many large-scale real-world graphs. The scale-free graphs are generated using the Kronecker generator [6] of Graph500 benchmark [7]. As in previous related work, we measure the BFS performance as the number traversed edges per second (TEPS), which is computed by dividing the actual edge number of the BFS tree over the execution time.

We implemented our design with 8-PE, 16-PE and 32-PE configurations. Our BFS hardware design is expressed in RTL using Verilog HDL, and was
compiled using Xilinx ISE v13.1. The resource consumption and timing performance are reported in Table II, which includes the two DRAM controllers. As shown in Table II, all the implementations can reach a clock frequency of over 160 MHz and it does not drop visibly with the array size growth from 8-PE to 32-PE, also showing the good scalability of our design.

| PE number | Slice LUTs | Block RAMs | Maximum frequency (MHz) |
|-----------|------------|------------|-------------------------|
| 8         | 24%        | 47%        | 212                     |
| 16        | 31%        | 59%        | 196                     |
| 32        | 47%        | 86%        | 172                     |

Set the vertex number to 16 million with an average arity from 8 to 32, we compare the BFS performance on our platform to that of ASAP12-1F, as shown in Table. III. ASAP12-1F, the single FPGA-based implementation of [5], is a state-of-the-art highly optimized BFS implementation on FPGA for large scale graphs, which is capable of obtaining superior performance to that of multi-core CPU platforms. Using a single Virtex5 LX330 FPGA with 32 PEs, our design outperforms ASAP12-1F using the same type of FPGA with 128 PEs by a factor of 1.25x to 1.32x.

| Arity | ASAP12-1F (GTEPS) | Ours (GTEPS) | Speedup |
|-------|--------------------|--------------|---------|
| 8     | 0.39               | 0.49         | 1.26    |
| 16    | 0.53               | 0.66         | 1.25    |
| 32    | 0.72               | 0.95         | 1.32    |

6 Conclusion

In this article, a multi-channel memory based architecture for efficient parallel processing of large-scale graph traversal on FPGA is proposed. By using two DRAM modules and two SRAM chips to improve the performance of the memory subsystem and developing an optimized pipelining structure for the PEs, the implementation is able to provide high bandwidth utilization for graph traversal algorithms. The experimental results show that our implementation is capable of obtaining superior performance to that of a state-of-the-art highly optimized BFS implementations using the same type of FPGA.

Acknowledgments

This research is supported by National Science Foundation of China under Grants No.61125201 and the Doctoral Fund of Ministry of Education of China under Grants No.60911062.