Optimizing Memory Performance for FPGA Implementation of PageRank

Shijie Zhou, Charalampos Chelmis, Viktor K. Prasanna
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA, USA
{shijiezh, chelmis, prasanna}@usc.edu

***Abstract*—Recently, FPGA implementation of graph algorithms arising in many areas such as social networks has been studied. However, the irregular memory access pattern of graph algorithms makes obtaining high performance challenging. In this paper, we present an FPGA implementation of the classic PageRank algorithm. Our goal is to optimize the overall system performance, especially the cost of accessing the off-chip DRAM. We optimize the data layout so that most of memory accesses to the DRAM are sequential. Post-place-and-route results show that our design on a state-of-the-art FPGA can achieve a high clock rate of over 200 MHz. Based on a realistic DRAM access model, we build a simulator to estimate the execution time including memory access overheads. The simulation results show that our design achieves at least 96% of the peak performance of the target platform. Compared with a baseline design, our optimized design dramatically reduces the number of random memory accesses and improves the execution time by at least 70%.**

***Index Terms* — FPGA, PageRank, Memory performance**

I. Introduction

Graphs have become increasingly important to represent real-world data, such as the World Wide Web and social networks [1]. However, obtaining high performance for graph processing is challenging. First, the datasets of real-world graph problems are massive and can easily overwhelm the computational and memory capabilities of the target platform [2]. Second, graph algorithms exhibit irregular memory accesses [2]. This results in poor spatial and temporal locality of memory accesses and a high data-access-to-computation ratio. Therefore, the runtime is dominated by long-latency external memory accesses.

Vertex-centric model [3] and edge-centric model [4] have been proposed to facilitate graph processing. In both models, the computation proceeds as a sequence of iterations. In the vertex-centric model, in each iteration, each vertex whose data has been updated sends the updated data to its neighbors; each vertex which has incoming messages recompute its own data based on the incoming messages. In the edge-centric model, in each iteration, all the edges are first traversed to produce updates, then the updates are iterated over and performed on the corresponding vertices.

PageRank [5] is a classic graph algorithm used to rank websites. It provides an approach to measure the importance of website pages and is widely used in search engines. According to the algorithm, a website of higher importance has a higher PageRank value. Both vertex-centric model and edge-centric model can be used to implement the PageRank algorithm.

FPGA technologies have become an attractive option for processing graph algorithms [6-10]. Recent works using FPGA to accelerate graph processing can achieve considerable speedups compared to GPU and CPU systems [7], [8]. However, due to the irregular memory access pattern of the graph algorithms, FPGA needs to perform many long-latency random memory accesses to the external memory. As a result, the processing elements on FPGA suffer many pipeline stalls, resulting in significant performance deterioration.

In this paper, we present an FPGA implementation of the PageRank algorithm. Our goal is to minimize the number of random memory accesses to the external memory (DRAM) by optimizing the data layout. Our main contributions are:

- We present an FPGA implementation of PageRank algorithm, which can achieve a high clock rate of over 200 MHz.
- We optimize the data layout to minimize the number of random memory accesses. By using our approach, most memory accesses to the DRAM are sequential.
- Based on a realistic DRAM access model, we build a simulator to estimate the system performance. The simulation results show that our design achieves up to 96% of the peak performance of the target platform.
- Compared with a baseline design, our optimized design dramatically reduces the number of pipeline stalls due to DRAM accesses. The overall execution time is improved by over 70%.

The rest of the paper is organized as follows. Section II introduces the background and related work. Section III presents the DRAM access model. Section IV discusses the algorithm and our optimization. Section V describes the architecture. Section VI reports experimental results. Section VII concludes the paper.

II. Background and Related Work

A. PageRank

The PageRank algorithm [5] is used in website search engine to measure the importance of websites. In the PageRank

This work is supported by the U.S. National Science Foundation under grants CCF-1320211 and ACI-1339756. Equipment grant from Xilinx Inc. is gratefully acknowledged.

algorithm, websites are represented as vertices and links are represented as edges. The PageRank algorithm outputs the PageRank value of each vertex which indicates the likelihood that the vertex will be reached by. A higher PageRank value corresponds to higher importance.

The computation of the PageRank algorithm consists of several iterations. In each iteration, each vertex v updates its PageRank value based on Formula (1).

$$PageRank(v) = \frac{1-d}{|V|} + d \times \sum \frac{PageRank(v_i)}{L_i} \quad (1)$$

In Formula (1), d is a damping factor (usually set to 0.85); $|V|$ is the total number of vertices; v_i represents the neighbor of v such that v has an incoming edge from v_i; L_i is the number of outgoing edges of v_i. More details of the PageRank algorithm can be found in [5].

B. Edge-centric Graph Processing

Edge-centric model for graph algorithms maintain vertex states (eg. PageRank value) and harness a scatter-gather model to traverse the edges [4]. The computation proceeds as a number of iterations, each iteration consisting of a scatter phase and a gather phase. In the scatter phase, all the edges are traversed to produce updates. In the gather phase, all the updates produced by the previous scatter phase are iterated over and performed on the corresponding vertices. Algorithm 1 shows a general template for the edge-centric model. For the PageRank algorithm, the value of the update is equal to the PageRank value of the vertex divided by the number of outgoing edges of the vertex; in Line 11, the update is performed based on Formula (1).

Algorithm 1 Edge-centric Scatter-Gather Processing

Let e denote an edge
Let u denote an update

```
1:  while not done do
2:      Scatter:
3:      for each edge e do
4:          Read vertex e.src
5:          Produce u with u.dest = e.dest
6:      end for
7:      Gather:
8:      for each update u do
9:          Read vertex u.dest
10:         if update condition is true then
11:             Update vertex u.dest based on u
12:         end if
13:     end for
14: end while
```

The tradeoff between vertex-centric model and edge-centric model has been discussed in [4]. We adopt the edge-centric approach for our PageRank implementation. The complexity of each iteration is $O(|V| + |E|)$ for both edge-centric and vertex-centric models, where $|V|$ is the total number of vertices and $|E|$ is the total number of edges; but edge-centric model sequentially accesses edges while vertex-centric model randomly accesses edges. Thus, edge-centric model sustains higher external memory bandwidth.

C. Related Work

In [14], PageRank is solved as an eigenvector problem. The implementation is based on Virtex-5 FPGA. With 3 sparse-matrix-by-vector-multiplication (SMVM) units in the architecture, the FPGA design outperforms the implementation on a general purpose processor (GPP) platform.

To the best of our knowledge, we are not aware of any prior work that leverages edge-centric model to implement the PageRank algorithm on FPGA.

III. DRAM Access Model

Due to the limited on-chip memory resources of FPGA, it is common for FPGA designs to access data from off-chip external memory such as DRAM [6], [9]. A DRAM chip is organized into banks [11]. Each bank consists of a two-dimensional matrix of locations. At each addressable location ([Bank, Row, Column]), a fixed number of data bits are located. Each bank has its own sense amplifier array and can be operated independently. This allows accesses to different banks to be pipelined. To access a row, the row needs to be activated first; meanwhile, the previous activated row needs to be closed by a row precharge command.

DRAM is built on a highly pipelined architecture. Therefore, there is a certain amount of latency between the execution of accesses [11]. The latency depends on several timing parameters of the DRAM device [12], such as row to column command delay and row precharge delay. The latency also depends on the memory access pattern. Sequential memory access pattern can achieve much smaller latency than random memory access pattern [12].

In our design, the DRAM access pattern includes sequential memory access pattern and random memory access pattern. For the sequential memory access pattern, consecutive memory accesses access the same row of the same bank. For the random memory access pattern, consecutive memory accesses may access different rows of the same bank or different banks. We define (1) the access which reads or writes an activated row as *row hit* and (2) the access which reads or writes a closed row as *row miss*. We abstract the DRAM access model based on three scenarios:

- Row hit
- Row miss, the same bank: consecutive memory accesses access the same bank and result in row miss
- Row miss, different banks: consecutive memory accesses access different banks and result in row miss

We define the latency between the execution of memory accesses (denoted as $t_{interval}$) as a parameter for each scenario. The $t_{interval}$ can be computed based on the timing parameters of the DRAM [11], [12]. Based on $t_{interval}$ and the clock rate of the FPGA, the number of pipeline stalls that FPGA suffers for each DRAM access is $\lceil \frac{t_{interval}}{t_{FPGA\_clk}} - 1 \rceil$.

IV. Algorithm and Optimization

Edge-centric model accesses edges sequentially but randomly accesses the data of vertices. Therefore, it is desirable to store the data of all the vertices in the on-chip BRAM. However, for large graphs, the on-chip BRAM resources of FPGA are not sufficient to store the data of all the vertices. To address this issue, we adopt the approach in [4] to partition the graph data. In this section, we discuss the algorithm and our optimization for data layout.

A. Pre-processing and Data Layout

After the graph data is partitioned, each partition has a vertex set whose data can be fit in on-chip memory. Each partition also maintains an edge set and an update set. The edge set stores all the edges whose source vertices are in the vertex set. The update set is used to store all the updates whose destination vertices are in the vertex set of the partition. The vertex set and edge set of each partition remain fixed during the entire computation, while the update set is recomputed in every scatter phase. Hence, the update sets of different partitions are stored in distinct memory locations.

We propose an *optimized data layout* which sorts the edge set of each partition based on the destination vertices. In the following sections, we will show that our optimized data layout reduces the number of random memory accesses from $O(|E|)$ to $O(k^2)$, where k is the number of partitions.

B. Scatter Phase and Gather Phase

Both the scatter phase and the gather phase are processed partition by partition. We show the PageRank algorithm based on the partitions in Algorithm 2.

Algorithm 2 PageRank

```
1:  while not done do
2:    Scatter:
3:    for each partition do
4:      Store vertex set into on-chip BRAM
5:      for each edge e in edge set do
6:        Read vertex e.src from BRAM
7:        Let v = vertex e.src
8:        Produce u with u.value = v.PageRank / v.number_of_edges
9:        u.dest = e.dest
10:       Write u into update set of Partition ⌊u.dest / k⌋
11:     end for
12:   end for
13:   Gather:
14:   for each partition do
15:     Store vertex set into on-chip BRAM
16:     for each update u in update set do
17:       Read vertex u.dest from BRAM
18:       Update vertex u.dest based on u and Formula (1)
19:     end for
20:     Write vertex set back into DRAM
21:   end for
22: end while
```

C. DRAM Access Pattern

In the scatter phase of Algorithm 2, Line 4, 5 and 10 perform DRAM accesses. Line 4 and 5 perform sequential memory accesses since the data of vertex set and edge set are stored contiguously in DRAM. However, Line 10 performs random memory accesses to write updates into DRAM. This is due to that the updates may belong to the update sets of different partitions and need to be written into different memory locations of DRAM. In the worst case, there are $O(|E|)$ random memory accesses which result in $O(|E|)$ row misses in the scatter phase. With our data layout optimization, the updates belonging to the same partition are produced consecutively and can be sequentially written into the DRAM. This is because the edge set has been sorted based on the destination vertices. There are at most k random memory accesses[1] due to writing the updates produced by each partition. Since there are k partitions, the number of random memory accesses to the DRAM is $O(k^2)$ in the scatter phase .

In the gather phase of Algorithm 2, Line 15, 16 and 20 perform sequential DRAM accesses. There is one random memory access when the computation switches between partitions. Since there are k partitions, the number of random memory accesses to the DRAM is $O(k)$ in the gather phase.

Overall, the total number of random memory accesses to the DRAM is $O(k^2) + O(k) = O(k^2)$ for each iteration. Since each random memory access results in at most one row miss, the number of row misses is $O(k^2)$ for each iteration, which is far less than $O(|E|)$ when k is small.

V. Architecture

A. Architecture Overview

We show the top-level architecture of our design in Fig. 1. The DRAM connects to the FPGA through the memory interface. On the FPGA, the control unit is responsible for generating DRAM access addresses and keeping track of the processing progress. The processing pipeline processes the input data read from the DRAM. The on-chip memory is used to store the data of the vertex set of the partition which is being processed.

B. Control Unit

We detail the control unit in Fig. 2. Based on the progress of the current iteration, the FPGA may read the data of vertices, edges or updates from the DRAM, and write the data of updates or vertices into the DRAM. The control unit is responsible for generating the DRAM access addresses accordingly. To realize this, the control unit uses registers to record the base addresses of the vertex set, edge set and update set of each partition. It also keeps track of the number of updates that have been written into the update set of each partition. The updates produced by the processing pipeline are first stored in the output buffer. When the buffer is full, the data stored in the buffer are passed to the memory interface and written into the DRAM. The data read from the DRAM

[1] One random memory access occurs when the value of $\lfloor \frac{u.dest}{k} \rfloor$ varies.

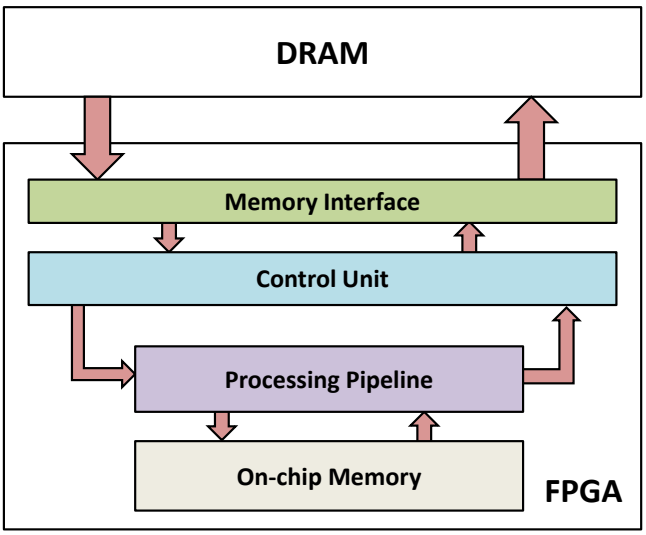

Fig. 1: Top-level Architecture

are fed into the processing pipeline and processed based on the control signal. The control unit is also responsible for checking whether the termination condition is met (eg. a certain number of iterations have been completed).

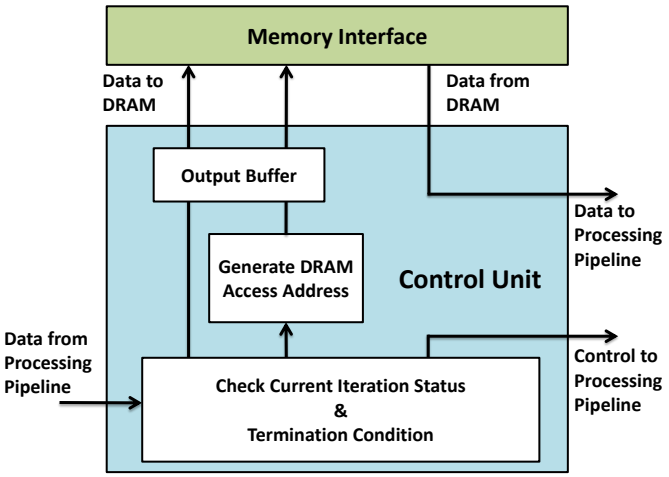

Fig. 2: Control unit

C. Processing Pipeline

Based on the input control signal, the processing pipeline perform the corresponding operation on the input data. If the input data represent edges, the processing pipeline produces updates accordingly; if the input data represent updates, the processing pipeline updates the data of the corresponding vertex.

In the gather phase, the processing pipeline needs to read and write vertices from the on-chip memory. Since there is delay between the reading and writing of the on-chip memory, read-after-write hazard may occur. To handle the potential read-after-write hazard, we include a data forwarding circuit in the architecture. The data forwarding circuit forwards the most up-to-date vertex data to all the pipeline stages. Each pipeline stage first compares the data read from the on-chip memory against the forwarded data. If the data from the on-chip memory and the forwarded data correspond to the same vertex, the computation will be based on the forwarded data.

To increase parallelism, the processing pipeline can process multiple edges or updates concurrently. We define the data parallelism of the architecture as the number of input data that can be processed concurrently. We denote the data parallelism as p. We adopt the approach in [15] to build a pRpW (p read ports and p write ports) on-chip memory using BRAMs. Thus, the data of p vertices can be read or written by the processing pipeline concurrently. In the gather phase, it is possible that multiple updates which have the same destination vertex are read in the same clock cycle. This will result in conflicts when updating the data of the vertex in on-chip memory. To address such data dependency, we add a combining network in front of the processing pipeline. In the gather phase, p updates enter the combining network in each clock cycle. The combining network sorts the p updates based on their destination vertices using bitonic sorting approach [16]. During sorting, if two updates have the same destination vertex, the updates are combined. The architecture of the processing pipeline for $p = 4$ is depicted in Fig. 3.

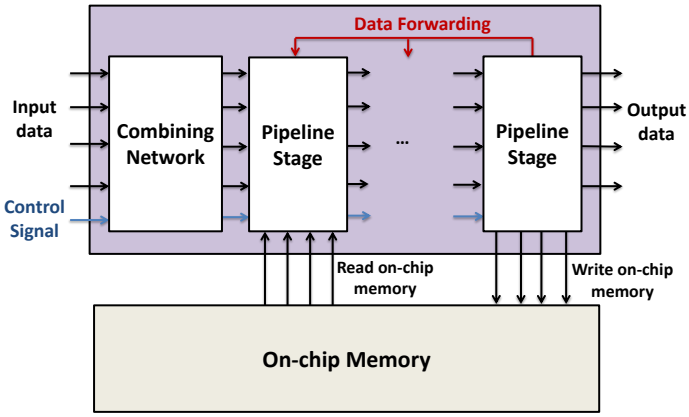

Fig. 3: Architecture of processing pipeline

VI. Performance Evaluation

A. Experimental Setup

The experiments were conducted on the Xilinx Virtex-7 xc7vx1140tflg1930 with -2L speed grade. The target platform has 712,000 slice LUTs, 1,424,000 slice registers, 850 I/O pins and upto 67 Mb of BRAM.

In the experiments, each edge consists of a 32-bit source vertex index and a 32-bit destination vertex index; each vertex maintains the PageRank value using 32 bits and the number of outgoing edges of the vertex using 32 bits; each update contains a 32-bit value and a 32-bit destination vertex index. We use the Micron 8Gb DDR3-1600 MT41K1G8 [17] in our design. The target DRAM runs at 800 MHz and has a peak

data transfer rate of 51.2 Gb/s. We evaluate our design using the following performance metrics:

- Clock rate: the clock rate sustained by the FPGA design
- Resource utilization: the utilization of FPGA resources
- Execution time: the execution time per iteration

We use Xilinx Vivado 2015.2 development tool to measure the clock rate and resource utilization of our FPGA design. The reported clock rate and resource utilization are based on post-place-and-route results.

B. Clock Rate and Resource Utilization

1) p=1: We first study the performance when the data parallelism is 1. We vary the partition size (number of vertices per partition) from 64K to 512K. Fig. 4 shows the clock rate and resource utilization of the PageRank designs. All the designs achieve a high clock rate of over 230 MHz for various partition sizes. The main bottleneck to support a larger partition is due to the BRAM resources. When the partition size reaches 512K, the BRAM resources are consumed by 55%.

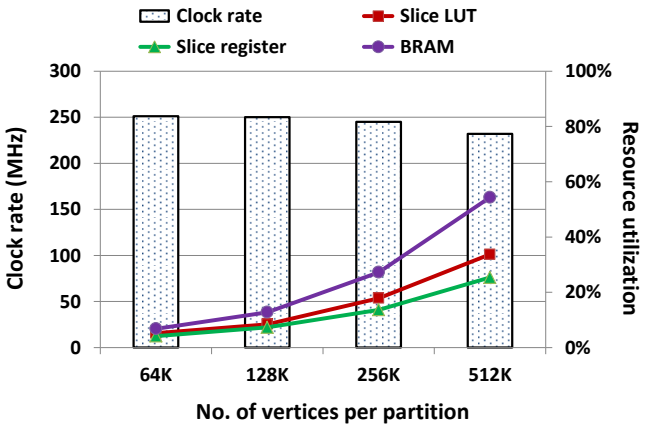

Fig. 4: Performance for $p = 1$

2) Increasing p: We further increase the data parallelism. Fig. 5 and Fig. 6 show the performance for $p = 2$ and $p = 4$, respectively. All the designs achieve a high clock rate of over 200 MHz. We observe that the clock rate slowly deteriorates as the partition size increases. This is due to a more complex routing for a larger partition size. As we increase the data parallelism with the same partition size, more resources are consumed for implementing the multi-port on-chip memory and its LVT table [15].

The main bottleneck to support a larger partition size comes from the slice LUT resources. For $p = 2$, the slice LUT resources are consumed by 53% when the partition size reaches 64K. For $p = 4$, the slice LUT resources are consumed by 96% when the partition size reaches 16K.

C. Execution Time and Memory Performance

We use real-world web graphs to study the system performance with respect to execution time. The datasets are

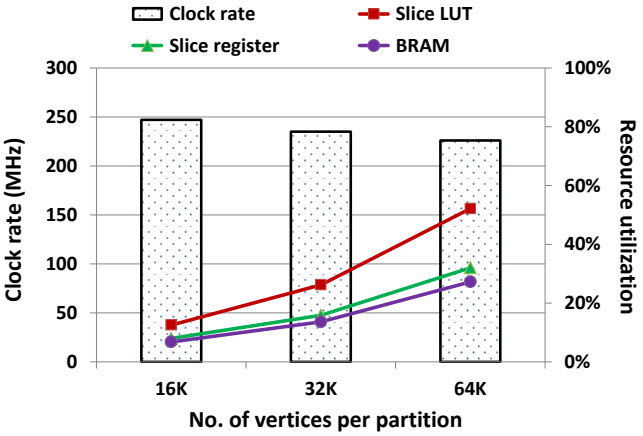

Fig. 5: Performance for $p = 2$

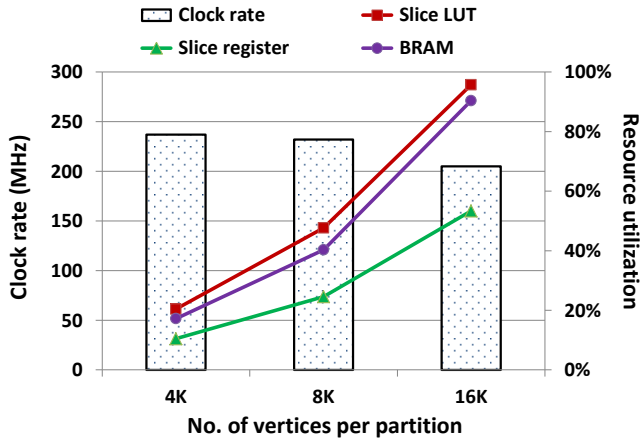

Fig. 6: Performance for $p = 4$

obtained from [18]. Table I summaries the datasets used in the experiments.

TABLE I: Datasets

Name	No. of vertices	No. of edges
web-NotreDame	325,729	1,497,134
web-Google	875,713	5,105,039
web-Berkstan	685,230	7,600,595
wiki-Talk	2,394,385	5,021,410

In the experiments, we use the FPGA design with data parallelism of 4 and partition size of 16K. The FPGA design runs at 200 MHz to saturate the peak bandwidth of the target DRAM. To evaluate the overall system performance, we build our own simulator. We did not use the publicly available DRAM simulators such as DRAMSim2 [19] due to (1) these DRAM simulators need to integrate an FPGA simulator as well to simulate the full system and (2) these DRAM simulators are designed for complex multi-thread

applications while the memory access pattern involved in our problem is much simpler.

Given a graph, our simulator first generates a sequence of memory accesses based on Algorithm 2. Then the memory accesses are reordered and executed based on the First Ready-First Come First Serve (FR-FCFS) scheduling policy [13], which is widely used in memory controllers to optimize memory performance for single-thread applications. Based on our DRAM access model introduced in Section III, we compute the number of pipeline stalls that FPGA needs to suffer for each DRAM access. Finally, the total execution time is obtained as the number of clock cycles that FPGA executes, including the pipeline stall overheads.

We compare the performance of our optimized design against a baseline design and the peak performance of the target platform. The baseline design uses the data layout in [4] which does not include our data layout optimization. To compute the peak performance, we assume that every DRAM access results in row hit and the pipeline on FPGA does not suffer any stalls due to DRAM accesses. In Fig. 7, we show the performance comparison with respect to the execution time. In Table II, we show the number of pipeline stalls due to the DRAM accesses for the optimized design and baseline design, respectively.

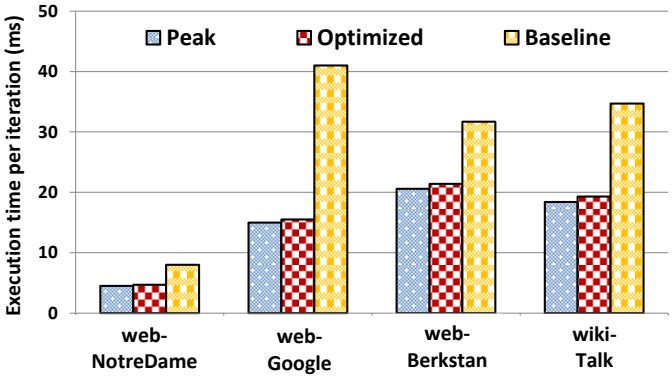

Fig. 7: Performance comparison of execution time

TABLE II: Performance comparison

Dataset	No. of pipeline stalls	
	Optimized	Baseline
web-NotreDame	30,973	682,014
web-Google	120,674	8,238,665
web-Berkstan	154,473	2,218,232
wiki-Talk	172,041	3,242,586

It can be observed that our optimized design dramatically reduces the number of pipeline stalls. As shown in Fig. 7, the performance of our optimized design is close to the peak performance. In all the experiments, our design can achieve at least 96% of the peak performance. Compared with the baseline design, the execution time per iteration is improved by up to 70%.

VII. Conclusion

In this paper, we presented an FPGA implementation of the classic PageRank algorithm. We optimized the data layout to minimize the number of random memory accesses to the DRAM. We conducted comprehensive experiments on a state-of-the-art FPGA. Post-place-and-route results show that our design achieves a high clock rate of over 200 MHz. We built a simulator to evaluate the system performance. The simulation results show that our optimized design achieves at least 96% of the peak performance of the target platform. Compared with the baseline design which does not include our optimization techniques, our optimized design improves the execution time by over 70%.

References

[1] "Graph 500," http://www.graph500.org/

[2] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in Parallel Graph Processing," in Parallel Processing Letters, vol. 17, no. 01, pp. 5-20, 2007.

[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in Proc. of SIGMOD, pp 135-146, 2010.

[4] A. Roy, I. Mihailovie and W. Zwaenepoel, "X-Stream: Edge-centric Graph Processing using Streaming Partitions," in Proc. of SOSP, pp. 472-488, 2013.

[5] L. Page, S. Brin, M. Rajeev and W. Terry, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report, 1998.

[6] Q. Wang, W. Jiang, Y. Xia and V. K. Prasanna, "A Message-passing Multi-softcore Architecture on FPGA for Breadth-first Search," in Proc. of FPT, pp. 70-77, 2010.

[7] B. Betkaoui, D. B. Thomas, W. Luk and N. Przulj, "A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study," in Proc. of FPL, pp. 1-8, 2011.

[8] O. G. Attia, T. Johnson, K. Townsend, P. Jones and J. Zambreno, "CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search," in Proc. of IPDPSW, pp. 228-235, 2014.

[9] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martnez and C. Guestrin, "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation," in Proc. of FCCM, pp. 25-28, 2014.

[10] S. Zhou, C. Chelmis and V. K. Prasanna, "Accelerating Large-Scale Single-Source Shortest Path on FPGA," in Proc. of IPDPSW, pp. 129-136, 2015.

[11] B. Jacob, S. W. Ng, and D. T. Wang, "Memory Systems: Cache, DRAM, Disk," Morgan Kaufman, 2007.

[12] Z. Dai, "Appliction-Driven Memory System Design on FPGAs," PhD thesis, 2013.

[13] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens, "Memory Access Scheduling," in Proc. of ISCA, pp. 128-138, 2000.

[14] S. McGettrick, D. Geraghty and C. McElroy, "An FPGA Architecture for the PageRank Eigenvector Problem," in Proc. of FPL, pp. 523-526, 2008.

[15] C. E. LaForest and J. G. Steffan, "Efficient Multi-Ported Memories for FPGAs," in Proc. of FPGA, pp. 41-50, 2010.

[16] K. E. Batcher, "Sorting Networks and Their Applications", in Proc. of Spring Joint Computer Conference, vol. 32, pp. 307-314, 1968.

[17] "DDR3 SDRAM Part," http://www.micron.com/products/dram/ddr3-sdram/8Gb#/width%5B%5D=x8&datarate%5B%5D=DDR3-1600

[18] "Stanford Large Network Dataset Collection," http://snap.stanford.edu/data/index.html#web

[19] P. Rosenfeld, E. Cooper-Balis and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," Computer Architecture Letters, vol. 10, 2011.