Accelerating Large-Scale Single-Source Shortest Path on FPGA

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—**Many real-world problems can be represented as graphs and solved by graph traversal algorithms. Single-Source Shortest Path (SSSP) is a fundamental graph algorithm. Today, large-scale graphs involve millions or even billions of vertices, making efficient parallel graph processing challenging. In this paper, we propose a single-FPGA based design to accelerate SSSP for massive graphs. We adopt the well-known Bellman-Ford algorithm. In the proposed design, graph is stored in external memory, which is more realistic for processing large-scale graphs. Using the available external memory bandwidth, our design achieves the maximum data parallelism to concurrently process multiple edges in each clock cycle, regardless of data dependencies. The performance of our design is independent of the graph structure as well. We propose a optimized data layout to enable efficient utilization of external memory bandwidth. We prototype our design using a state-of-the-art FPGA. Experimental results show that our design is capable of processing 1.6 billion edges per second (GTEPS) using a single FPGA, while simultaneously achieving high clock rate of over 200 MHz. This would place us in the 131st position of the Graph 500 benchmark list of supercomputing systems for data intensive applications. Our solution therefore provides comparable performance to state-of-the-art systems.**

I. Introduction

Graphs have become increasingly important to represent real-world networked data, including the World Wide Web, social networks, knowledge graphs, genome analysis and medical informatics. The rapid growth of such highly interconnected large-scale graph-structured data and their increased use in machine learning and data mining applications has led to the development of various graph computing systems in recent years. Current graph processing systems are able to scale to massive graphs by distributing the computation over commodity clusters [1], utilizing supercomputers [2] or multi-core systems [3], or by employing dedicated hardware such as GPUs [4], [5] and FPGAs [6], [7], [8]. Increasing the number of processing elements (PEs) to achieve high computational performance has proved practical by the dominance of computer clusters in the TOP500 supercomputer list and the Graph 500 List [2], [9]. However, high-end supercomputers often entail steep prices and high power consumption. Similarly, while distributed computational resources have become more available (e.g. through the Cloud), developing distributed graph algorithms still remains challenging [10], especially to non-experts. As graph problems grow larger in size and complexity, an interesting question arises: is fast graph computation possible on just a tiny FPGA?

This work has been funded by U.S. National Science Foundation under Grants No. 1018801 and No. 1320211. Equipment grant from Xilinx Inc. is gratefully acknowledged.

FPGA technologies have become an attractive option for solving graph problems, often achieving considerable speedups compared to GPU/CPU systems [8], [11], [12]. State-of-the-art FPGA devices provide dense logic units, large amounts of on-chip memory and high-bandwidth interfaces for various external memory technologies [13]. However, most existing FPGA-implementations accommodate on-chip memory in order to store graphs [14]; this is not suitable for large-scale graph problems. Recent FPGA-based platforms, such as the Convey hybrid-core system [15], are known for their high memory bandwidth and performance in computation-intensive and memory-bound applications [16], [12], [8]. The Convey hybrid-core system integrates Intel processors with multiple FPGA-based coprocessors [15] which are connected to memory controllers in a full crossbar fashion; the memory controllers connect to Convey-designed memory modules. However, the bandwidth between the memory controller and the memory modules becomes the bottleneck when different coprocessors attempt to read from the same memory module [15].

Processing large-scale graphs is a hard problem [10] as real-world graphs pose formidable challenges. First, real-world graph problems are characterized by massive datasets which can easily overwhelm the computational and memory capabilities of a conventional supercomputer. For example, World Wide Web, now contains more than 50 billion web pages and more than one trillion unique URLs [17]. Secondly, graphs represent relationships which are usually irregular and unstructured. The data intensive nature of large-scale graph problems makes sequential graph algorithms impractical and graph computations difficult to parallelize. Specifically, graph algorithms exhibit poor locality of memory accesses resulting in high *data-access-to-computation* ratio (i.e., the runtime is dominated by memory fetches). [8] proposed a reconfigurable architecture for parallel graph traversal on the Convey HC-2 platform that contains four programmable Virtex5-LX330 FPGAs, each accessing external memory at peak bandwidth of 20 GB/s. By spreading computation across multiple FPGAs such solutions require inter-device synchronization mechanisms, which result in increased latency. In graph-structured computation, natural graphs with highly skewed power-law degree distributions are commonly found in the real-world. It is highly likely that different coprocessors need frequently read and write data for the same vertex in designs involving multiple FPGAs. This results in inefficient utilization of the memory bandwidth.

Single-Source Shortest Path (SSSP) is a fundamental graph algorithm, which finds the shortest paths from a source vertex to all other vertices in the graph. Many applications, such as VLSI computer-aided design and urban traffic simulation [18], require high-speed SSSP computation. SSSP is also a key kernel proposed by the Graph 500 committees for the Graph 500 List [9]. There are many well-established algorithms for SSSP in the literature [19]. Among these algorithms, we consider the Bellman-Ford algorithm [20], [21], [22] because of the massive parallelism inherent in the algorithm. The Bellman-Ford algorithm relaxes the weights of edges in an iterative fashion until the shortest paths to all vertices in the graph are computed. In the worst case, the computation complexity of the algorithm is $O(ve)$, where v and e are the number of vertices and edges in the graph respectively. However, in practice, the number of iterations can be reduced by applying optimization techniques [23] such as early termination.

In this paper, we introduce a single-FPGA based design to accelerate the Bellman-Ford algorithm for SSSP. Edges are streamed into FPGA from external memory. After a number of iterations, all shortest paths are computed. Our architecture processes multiple edges in parallel on FPGA, regardless of data dependencies between edges. By eliminating such data dependencies, our streaming processing approach enables memory utilization at peak bandwidth and achieves the maximum data parallelism. The main contributions of our work are:

- We propose a single-FPGA based design to accelerate SSSP on large-scale graphs. Edges are streamed into FPGA from external memory.
- With the available external memory bandwidth, our design achieves the maximum data parallelism to process input stream. In each clock cycle, multiple edges are concurrently processed, regardless of dependencies between edges and graph structure.
- We include an efficient data forwarding scheme to handle the data hazards in the pipelined architecture. The scheme ensures the correctness of the architecture without stalling the pipeline.
- We propose an optimized data layout in the external memory to enable efficient utilization of the external memory bandwidth.
- We implement our design on a state-of-the-art Xilinx Vertex-7 FPGA. The experimental results show that our design achieves a high clock rate of over 200 MHz for various values of data parallelism. This results in a high throughput of 1.6 GTEPS using a single FPGA.

The rest of the paper is organized as follows: Section II provides background and related works; Section III describes the problem we address; Section IV details the architecture of our proposed design; we discuss experimental results in Section V; we conclude the paper in Section VI.

II. Background and Related Work

A. Bellman-Ford Algorithm

In this section, we briefly introduce the Bellman-Ford Algorithm [20]. Additional details can be found in [19]. The Bellman-Ford algorithm is illustrated in Algorithm 1[1].

Algorithm 1 Bellman-Ford algorithm

```
Let $G = (V, E)$ denote the graph that consists of a set of vertices $V$ and a set of edges $E$.
Let $v$ denote the number of vertices
Let $e$ denote the number of edges
Let edge$(i, j)$ denote the edge from vertex $i$ to vertex $j$
Let $w(i, j)$ denote the weight of edge$(i, j)$
Let $w(i)$ denote the weight of vertex $i$
Bellman-Ford$(G(V, E))$
 1: for each vertex $x$ in $V$ do
 2:   if $x$ is $source$ then
 3:     $w(x) = 0$
 4:   else
 5:     $w(x) = \infty$
 6:     predecessor$(x) = null$
 7:   end if
 8: end for
 9: for $i = 1$ to $v - 1$ do
10:   for each edge$(i, j)$ in $E$ do
11:     if $w(i) + w(i, j) < w(j)$ then
12:       $w(j) = w(i) + w(i, j)$
13:       predecessor$(j) = i$
14:     end if
15:   end for
16: end for
```

In Algorithm 1, each vertex maintains the weight of the shortest path from the source vertex to itself and the vertex which precedes it in the shortest path. In each iteration, all edges are relaxed and the weight of each vertex is updated if necessary. After the ith iteration ($1 \leq i \leq v-1$), the algorithm finds all the shortest paths consisting of at most i edges. Since the theoretically longest possible path has $v - 1$ edges, $v - 1$ iterations are required to ensure the shortest path has been found for all the vertices. The computation complexity of this algorithm is $O(ve)$.

In practice, the number of vertices in any shortest path is far less than $v - 1$ [6]. If no updates are performed during an iteration, the algorithm can be immediately terminated since subsequent iterations will not incur changes as well. With the early termination technique, the algorithm requires less than $v-1$ iterations to terminate, significantly reducing the runtime [6].

B. Related Work

There have been many FPGA-based SSSP implementations in the literature [24], [25], [6], [26]. An early approach to solving graph problems on FPGAs is proposed by Babb *et al.* [24]. They develop a compilation technique to store a specific graph by building a circuit on FPGA. The circuit resembles the graph by using logic units to represent vertices and wires

[1]Our design mainly targets the graphs with non-negative edges.

to represent edges. However, this approach lacks of flexibility since any change in the graph will require recompilation and reconfiguration. More recently, [25] adds more flexibility into this method of circuit representation by storing the adjacency matrix of the graph on FPGA. However, this approach can only compute the shortest unit path, in which all the edges have unit cost.

Dandalis *et al.* [6] develop a pipelined architecture which is composed of a set of processing elements (PEs). Each PE corresponds to a vertex. Edges are stored in external memory and passed into FPGA iteratively. The FPGA does not need to be reconfigured as long as the number of vertices does not exceed the number of PEs. However, that design only processes one edge in each clock cycle, resulting in a limited speedup.

The architecture proposed by Jagadeesh *et al.* [26] also uses a PE for each vertex. In the preprocessing step, the incoming edges of each vertex are stored in a RAM within the corresponding PE. The design adopts Bellman-Ford algorithm. By using a broadcast scheme, each iteration takes $v+1$ cycles instead of e cycles. In the ith clock cycle of an iteration ($1 \leq i \leq v$), the ith PE broadcasts its update to all the other PEs. However, that work targets SSSP for small graph. The maximum number of vertices reported in [26] is 128.

Previous works perform SSSP on-chip for small graphs. Either the entire graph or the vertices of graph can be represented by on-chip resources. Thus, these works can not be adopted to accelerate SSSP for large-scale graphs due to lack of on-chip resources. To the best of our knowledge, our design is the first single-FPGA based work to accelerate the SSSP computation for large-scale graphs.

III. Problem Statement

Our FPGA-based design aims to accelerate SSSP computation for large-scale graphs, which can not be represented only by on-chip resources. We assume the entire graph is stored in external memory. Given a certain amount of memory bandwidth between external memory and FPGA, p memory words for p edges are streamed into FPGA per clock cycle. After a certain amount of time, the updates based on the first batch of p memory words are written into external memory. With continuous input data streams, updates are written into external memory in each clock cycle. After running the Bellman-Ford algorithm for a number of iterations, all the shortest paths are computed and stored in external memory.

IV. Architecture

In this section, we first introduce the architecture overview of our design. Then we present each building block of the architecture in details.

A. Architecture Overview

As depicted in Fig. 1, a DRAM where the entire graph is stored connects to the FPGA. In each clock cycle, p memory words each corresponding to an edge are streamed into the FPGA from DRAM. We define p as the data parallelism of the architecture. The p memory words are sorted in the sorting block to ensure that each destination vertex of the p edges will have only one valid update. The computation block determines the destination vertices that need to be updated. Finally, the valid updates are written into DRAM through the memory write block. A data forwarding circuitry is added in order to handle the data hazard.

The architecture is fully pipelined to maintain a high clock rate. Assuming the external memory bandwidth is sufficient to support the maximum data parallelism of p, the architecture can concurrently process p edges per clock cycle. The details of the design for each building block are presented in the following sections.

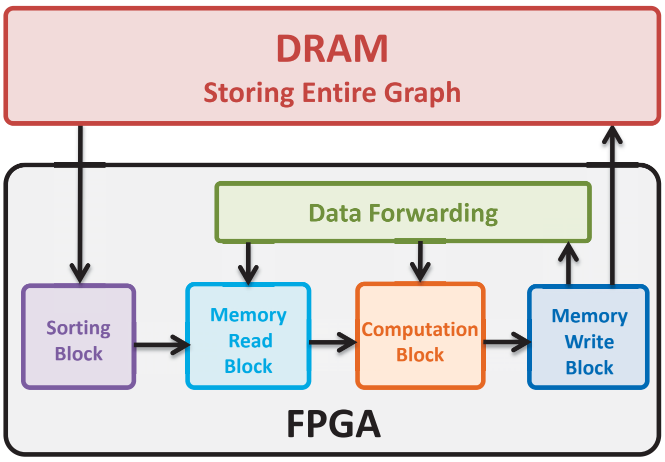

Fig. 1: Architecture overview

B. Sorting Block

The input to the sorting block is p memory words. Each memory word consists of the weight of the edge $w(i,j)$, the weight of the source vertex $w(i)$ and the associated indices i and j of the edge. There are chances that more than one edges target the same destination vertex but produce different update values. For example, memory word $< w(10,2) = 8, w(10) = 3 >$ and memory word with $< w(6,2) = 2, w(6) = 4 >$ both target vertex 2, but the possible update values for vertex 2 based on them are 11 and 6, respectively. For each destination vertex, only the minimum update value should be considered (6 in this example). The sorting block is used to identify the possible minimum update value for each destination vertex.

We adopt bitonic sorting algorithm [27] to sort the p memory words using $O((\log p)^2)$ pipeline stages. When p is a power of 2, the sorting block contains $(1+\log p)\log p/2$ pipeline stages. Each pipeline stage has $\frac{p}{2}$ comparators working in parallel. Fig. 2 depicts the sorting block for $p = 4$.

In the sorting block, each comparator is based on Algorithm 2 to determine the comparison result. We introduce a 1-bit update signal for each memory word. The update signal is used to indicate whether the memory word results in a valid update. An update signal set to 1 indicates a valid update and 0 indicates an invalid update. Initially, the update signals for the p memory words are all set to 1. Each comparator first checks the update signals of the input memory words. The memory word with update signal of 1 is smaller than the

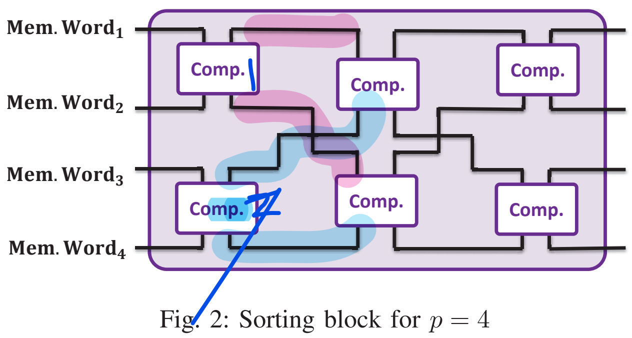

Fig. 2: Sorting block for $p = 4$

memory word with update signal of 0. If both update signals are 1, the destination vertex with a smaller index is determined to be smaller. If both update signals are 1 and the destination vertices are the same, the comparator compares the update values, $w(i,j) + w(i)$, of the memory words. In this case, the comparator also sets the update signal of the memory word with the larger update value to 0. Thus, after the sorting block, each destination vertex involved by the p memory words only has one valid update signal.

Algorithm 2 Comparator for the sorting block

Let j_k denote the destination vertex of $Memory\_word_p$ ($k = 0, 1$)
Let W_k denote the update value of $Memory\_word_p$ ($k = 0, 1$)
Let U_k denote the update signal of $Memory\_word_p$ ($k = 0, 1$)

```
Compare($Memory\_word_0, Memory\_word_1$)
1:  if $U_0 = U_1 = 1$ then
2:    if $j_0 = j_1$ then
3:      if $W_0 < W_1$ then
4:        $Memory\_word_0 < Memory\_word_1$
5:      else
6:        $Memory\_word_0 > Memory\_word_1$
7:      end if
8:    else
9:      if $j_0 < j_1$ then
10:       $Memory\_word_0 < Memory\_word_1$
11:     else
12:       $Memory\_word_0 > Memory\_word_1$
13:     end if
14:   end if
15: else if $U_0 \neq U_1$ then
16:   if $U_0 = 1$ then
17:     $Memory\_word_0 < Memory\_word_1$
18:   else
19:     $Memory\_word_0 > Memory\_word_1$
20:   end if
21: else
22:   Default: $Memory\_word_0 < Memory\_word_1$
23: end if
```

C. *Memory Read Block*

The memory read block fetches the current weight of the destination vertex $w(j)$ for each memory word with update signal of 1 from external memory. The $w(j)$ for the memory word with update signal of 0 is set to a default value. Assuming there are d ($1 \le d \le p$) memory words with update signal of 1, d weights need to be read from external memory.

D. *Computation Block*

After the memory read block, each input memory word to the computation block contains the update value $w(i)+w(i,j)$, indices i and j, current weight $w(j)$ of the destination vertex, and the update signal. The computation block compares the update value with the current weight of the destination vertex to determine the final update signal. If $w(i) + w(i,j)$ is less than $w(j)$, the final update signal is set to 1; 0 otherwise. There are p comparison modules in the computation block. Each comparison module is responsible for one memory word. We show the structure of the comparison module in Fig. 3.

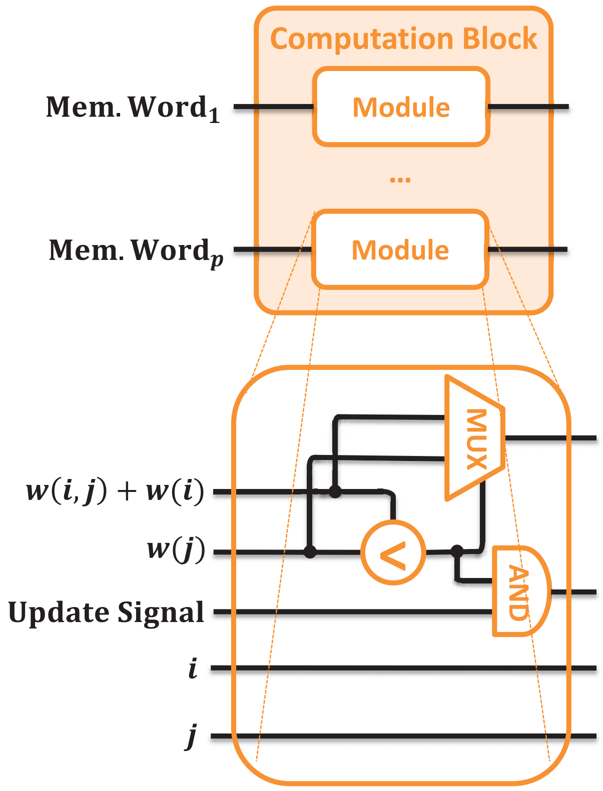

Fig. 3: Comparison Module

E. *Memory Write block*

The memory write block is responsible for writing the update values into external memory. For memory words with update signal of 1, the updated weight and the predecessor vertex of the corresponding destination vertex are written into external memory. The memory write block also checks the early termination condition. When a new iteration starts, a 1-bit signal set to 1 is used to indicate a new iteration. The memory write block keeps track of whether there is any valid update in each clock cycle. If there is no valid update for $\frac{e}{p}$ clock cycles[2] since a new iteration starts, the SSSP computation can be safely terminated.

In order to handle possible data hazard, the memory write

[2]One iteration takes $\frac{e}{p}$ clock cycles.

block forwards p memory words to the memory read block and the computation block through a data forwarding circuitry.

F. Data Forwarding Unit

Although the Bellman-Ford algorithm does not require using up-to-date weights of vertices when relaxing edges [20], [21], [22], data hazard can occur when the same vertex is updated in two consecutive clock cycles. This may result in increasing the weight of vertex. For example, as shown in Fig. 4, two memory words of two consecutive clock cycles have the same destination vertex; the memory word of the former clock cycle has $w(10, 6) = 3$ and $w(10) = 12$; the memory word of the latter clock cycle has $w(8, 6) = 6$ and $w(8) = 11$; currently, $w(6)$ is 20 in the memory. The former memory word will produce a valid update to change $w(6)$ into 15 since the corresponding update value is smaller than the current $w(6)$. However, when the latter memory word is in the memory read block, $w(6)$ is still 20 since the update resulted by the former memory word has not been written into memory. Thus, the latter memory word will also produce a valid update to change $w(6)$ into 17. Finally, the value of $w(6)$ in memory is 17 instead of 15.

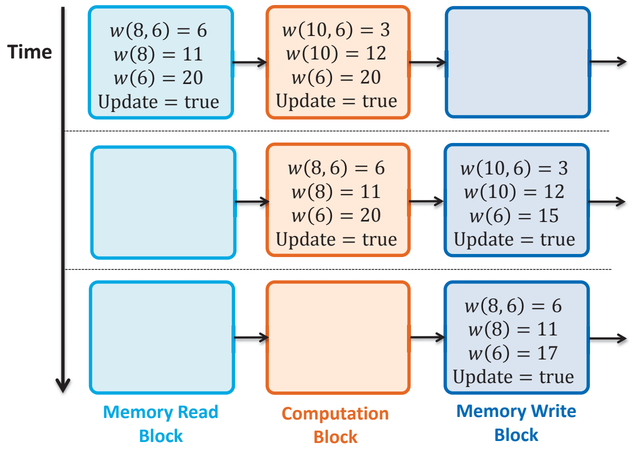

Fig. 4: Example of Data Hazard

One simple solution to handle such data hazard would be to stall the pipeline during each read from memory, to ensure that the most up-to-date value is written into memory. However, stalling the pipeline would reduce throughput. This is problematic when processing large-scale graphs. To handle these data hazards, we implement a data forwarding scheme to avoid the need for stalling the pipeline. The memory write block forwards up-to-date values to the memory read block and the computation block. In the memory read block and the computation block, each input memory word will compare its own destination vertex against the destination vertices of the p forwarded memory words. If a match is found and the corresponding forwarded word has a valid update signal, the input memory word replaces the weight of the destination vertex read from memory using the weight of the forwarded word. The correctness of such data forwarding scheme has been proved in [28]. Using the data forwarding scheme to handle the data hazard of Fig. 4 is depicted in Fig. 5. The algorithm of using the data forwarding scheme to select up-to-date values is illustrated in Algorithm 3. The computation complexity of the data forwarding scheme is $O(p^2)$.

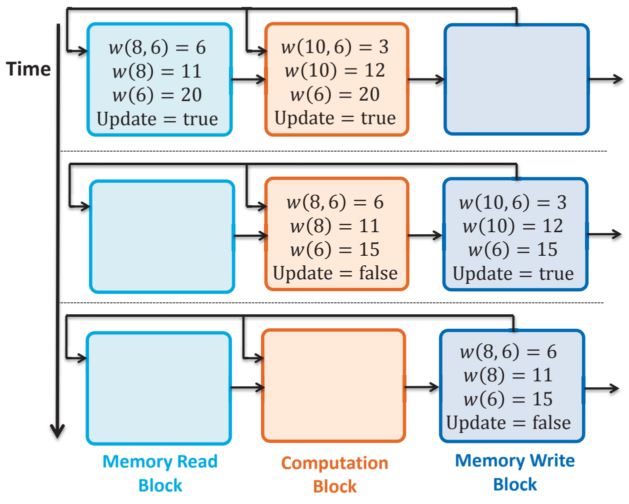

Fig. 5: Data Forwarding Scheme

Algorithm 3 Data Forwarding Scheme

```
Let $Memory\_word_k$ denote the $k$th input memory word ($k = 1, .., p$)
Let $Forward\_word_k$ denote the $k$th forwarded memory word ($k = 1, .., p$)
Let $M\_j_k$ denote the destination vertex of $Memory\_word_k$
Let $M\_W_k$ denote the weight of $M\_j_k$
Let $M\_U_k$ denote the update signal of $Memory\_word_k$
Let $F\_j_k$ denote the destination vertex of $Forward\_word_k$
Let $F\_W_k$ denote the weight of $F\_j_k$
Let $F\_U_k$ denote the update signal of $Forward\_word_k$
 **Data Forwarding**($Memory\_word_1, ..., Memory\_word_p$, $Forward\_word_1, ..., Forward\_word_p$)
1: **for** $m = 1$ to $p$ **do**
2:     **for** $n = 1$ to $p$ **do**
3:         **if** $M\_j_m = F\_j_n$ **and** $M\_U_m = F\_U_n = 1$ **then**
4:             $M\_W_m = F\_W_n$
5:         **end if**
6:     **end for**
7: **end for**
```

G. Data Layout in External Memory

Our design does not require a particular data layout in external memory. However, a proper data layout can lead to a more efficient utilization of external memory bandwidth. We use DDR3 DRAM as an example to illustrate our ideas. The same analysis can be extended to other types of DRAM. A DRAM chip is organized in multiple banks, each composed of a large array of rows and columns. A fixed number of data bits (the bus width) are located at any valid [bank, row, column] address. The following major parameters define the DDR3 DRAM access performance with various activation scenarios [29], [30]:

- t_{RCD}: open/active a specific row, $\approx 15\ ns$

- t_{CCD}: minimum time between successive accesses to the same bank and row, $\approx 5\ ns$
- t_{RRD}: minimum time between successive activate commands to different banks, $\approx 8\ ns$
- t_{RC}: minimum time between issuing two successive activate commands in a single bank, $\approx 40\ ns$
- t_{RP}: precharge the long wires before switching to the next, $\approx 15\ ns$

Intuitively, t_{RC} is much larger than t_{CCD}. Every time a new row in a memory bank is accessed, the row must be precharged. Such DRAM row activation is very expensive and results in extra delay. Thus, it is desirable to minimize the number of row activations. We propose to arrange the edges based on an ascending order of the destination vertices in external memory. Edges with the same destination vertex are stored in contiguous locations in the same row. Fig. 6 shows an example of the proposed data layout.

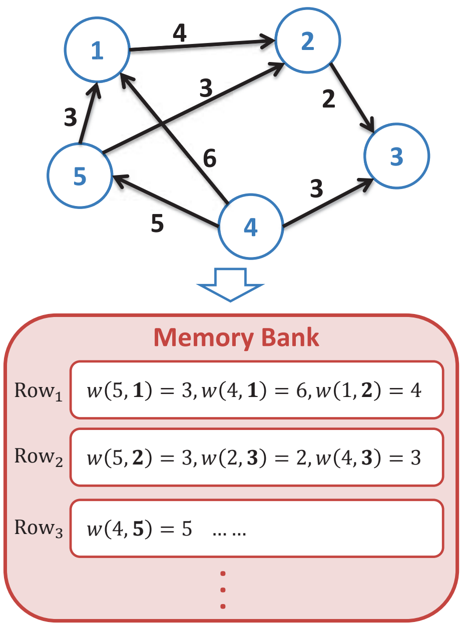
1
2
3
4
5
4
3
2
3
6
3
5
Memory Bank
Row$_1$ $w(5,\mathbf{1}) = 3, w(4,\mathbf{1}) = 6, w(1,\mathbf{2}) = 4$
Row$_2$ $w(5,\mathbf{2}) = 3, w(2,\mathbf{3}) = 2, w(4,\mathbf{3}) = 3$
Row$_3$ $w(4,\mathbf{5}) = 5$ … …

Fig. 6: Proposed Data Layout

There are two major advantages of such data layout:

- When memory words are streamed into FPGA from external memory, the number of row activations is significantly reduced, since edges are stored continuously in the same row.
- In the memory read block and the memory write block, it is highly likely that the number of vertices that need to be updated is much smaller than p. This is a direct result of many edges having the same destination vertex. Thus, the external memory bandwidth requirement between FPGA and external memory can be reduced.

V. Implementation and Performance

A. Experimental Setup

We conduct all of our experiments on a state-of-the-art Xilinx Virtex 7 XC7VX980 with -2L speed grade. The target platform has 303,600 logic slices, 850 I/O pins, 36 Mb BRAMs and up to 16 Mb distributed RAMs. The performance is evaluated using Xilinx Vivado 2014.3 development tools. We use the following performance metrics:

- Clock rate sustained by the design
- Utilization of FPGA resources

We represent the weight of each edge using a x-bit number, the weight of each vertex using a y-bit number and the index of each vertex using a z-bit number. Since the weight of vertex is the sum of weights of multiple edges, y is larger than x. In each clock cycle, FPGA needs to read $p(x+2y+2z)$ bits from external memory and write $p(y+z)$ bits into external memory. We assume that the external memory bandwidth of the target platform is sufficient to support the above requirement.

B. Varying the data parallelism

In this section, we vary the data parallelism p. We fix x, y and z at 4, 8 and 10, respectively. Fig. 7 and Fig. 8 depict the clock rate and the resource utilization for various p. We observe that:

- The clock rate decreases as the data parallelism increases. The deterioration is mainly caused by a more complex data forwarding scheme.
- The resource usage increases linearly with the data parallelism p. The increase is due to more comparison modules for the computation block and more pipeline stages for the sorting block.

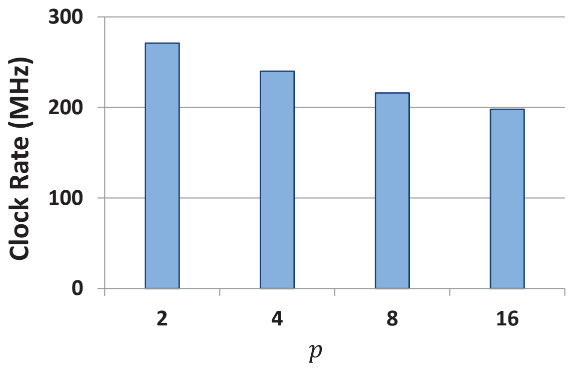

Fig. 7: Clock rate for various p

C. Varying the graph size

In this section, we vary the graph size by varying parameter z. In these experiments, we fix x, y and p at 4, 10 and 8, respectively. Fig. 9 and Fig. 10 show the clock rate and the resource utilization for various values of z. The key observations are:

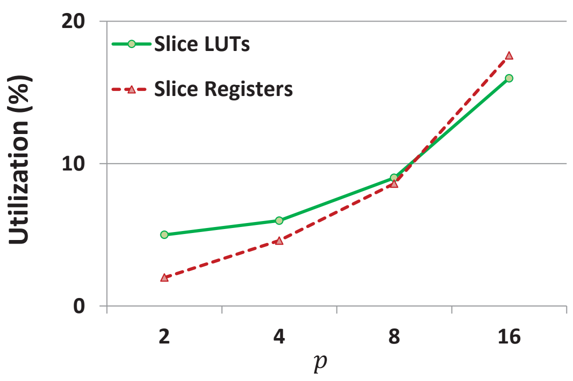

Fig. 8: Resource utilization for various p

- The clock rate drops slowly as the graph size increases. This is due to higher routing complexity as a result of the larger data width.
- The slice LUT utilization and slice register utilization do not increase substantially.

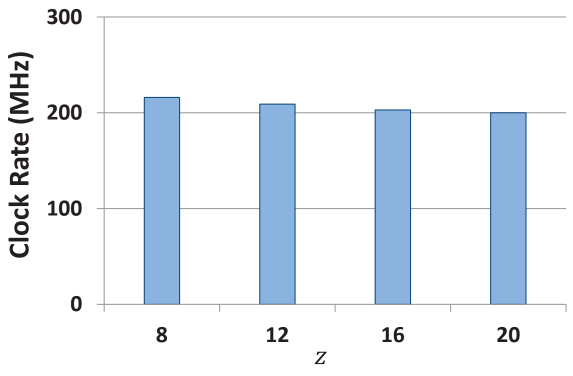

Fig. 9: Clock rate for various z

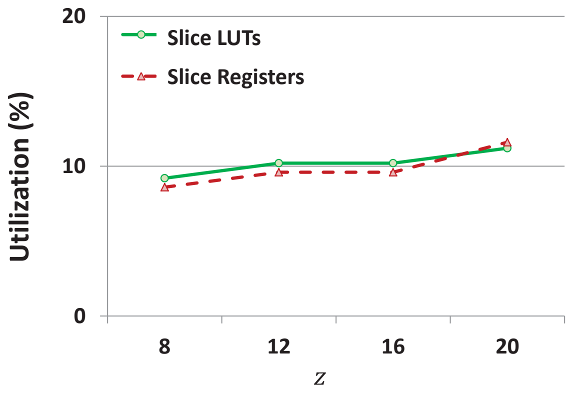

Fig. 10: Resource utilization for various z

D. Comparison with Previous Works

We compare our design with [6] and [26]. The comparison results are summarized in Table I.

TABLE I: Summary of comparison

Approach	Clock cycles per iteration	Dependent on graph size
[6]	e	Yes
[26]	$v+1$	Yes
Our design	e/p	No

Both [6] and [26] target small graphs, all the vertices of which can be represented by on-chip resources. [6] only processes one edge per clock cycle, resulting in a limited speedup. By using a broadcast scheme, each iteration in [26] takes $O(v)$ cycles, but [26] requires each vertex to store a sorted list of incoming edges in on-chip memory. Sorting the incoming edges for each vertex introduces significant overhead. Compared with [6] and [26], our design has the following advantages:

- Our design is independent of the size of graph.
- Our design does not introduce extra preprocessing overhead.
- Our design can process multiple edges per cycle regardless of dependencies between edges.

We also compare the performance of our design with [8] and [12], which use the Convey HC platform [15]. [8] and [12] study the breadth-first search (BFS) algorithm, which is also a fundamental 'search' graph operation [2]. The comparison is based on the throughput in billions of edges traversed per second (GTEPS). For fair comparison, assuming the clock rate sustained by each design is 200 MHz and each FPGA can access external memory at a peak bandwidth of 20 GB/s. The comparison results are summarized in Table II. It can be observed that our design achieves more than 2x performance per FPGA. Note that the memory word needed for each edge is larger for SSSP than BFS.

TABLE II: Performance comparison

Approach	Graph size v	No. of FPGAs	GTEPS per FPGA
[8]	2^{20}	4	0.5
[12]	2^{20}	4	0.6
Our design	2^{20}	1	1.6

VI. Conclusion

In this paper we presented a high-performance pipelined architecture on FPGA to accelerate SSSP for large-scale graphs. We assumed that the entire graph was stored in external memory. We proposed an optimized data layout for storing edges in external memory based on an accurate model of DRAM access. By effectively using the available external memory bandwidth, our design achieved the maximum data parallelism to continuously process the input edge stream, regardless of graph structure or dependencies between edges. We evaluated our design using a state-of-the-art Virtex 7 FPGA. Experimental results showed that the proposed architecture achieved a high clock rate of over 200 MHz for various data parallelism. This corresponded to a high throughput of traversing 1.6 billion edges per second (GTEPS), which could rank 131 in the Graph 500 List.

In the future, we will explore to accelerate other key graph algorithms such as BFS using FPGA. We will also compare the performance of our design with other accelerators such as GPU and multi-core systems.

References

[1] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski "Pregel: a system for large-scale graph processing." in Proc of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010.

[2] "The Graph 500 List," http://www.graph500.org/.

[3] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC10. IEEE Computer Society, pp. 1-11, 2010.

[4] Pawan Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," In Proc. of the 14th International Conference on High Performance Computing, pp 197-208, 2007.

[5] G. J. Katz and J. T. Kider Jr, "All-pairs shortest-paths for large graphs on the GPU" In Proc. of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hard- ware, pp. 47-55, 2008.

[6] A. Dandalis, A. Mei, V. K. Prasanna, "Domain specific mapping for solving graph problems on reconfigurable devices," Parallel and Distributed Processing, vol. 1586, pp 652-660, 1999.

[7] S. Hong, Tayo O. and Kunle O., "Efficient parallel graph exploration on multi-core CPU and GPU," in Proc of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 78-88, 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 the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, pp. 228-235, 2014.

[9] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," Cray User's Group, May 2010.

[10] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," in Parallel Processing Letters, vol. 17, no. 01, 2007, pp. 520.

[11] B. Betkaoui, D. B. Thomas, W. Luk and N. Przulj, "A Framework for FPGA Acceleration of Large Graph Problems: Graphlet Counting Case Study," International Conference on Field-Programmable Technology, pp. 1-8, 2011.

[12] B. Betkaoui, Y. Wang, D. B. Thomas and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," International Conference on Application-Specific Systems, Architectures and Processors, pp. 8-15, 2012.

[13] "Virtex-7 FPGA Family," http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/.

[14] Q. Wang, W. Jiang, Y. Xia, and V. Prasanna, "A message-passing multisoftcore architecture on FPGA for Breadth-first Search," in Proc of the International Conference on Field-Programmable Technology (FPT), Dec. 2010, pp. 70-77.

[15] J. D. Bakos, "High-Performance Heterogeneous Computing with the Convey HC-1," Computing in Science and Engineering, vol. 12, iss. 6, pp. 80-87, 2010.

[16] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "Parallel FPGA-based all pairs shortest paths for sparse networks: A human brain connectome case study," in Proc of the International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 99-104.

[17] "The size of the World Wide Web," http://www.worldwidewebsize.com/.

[18] D. Z. Chen, "Developing algorithms and software for geometric path planning problems," ACM Computing Surveys (CSUR), vol. 28 , no. 18, 1996.

[19] B. V. Cherkassky, A. V. Goldberg and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," Mathematical Programming 31, vol. 73, iss. 2, pp. 129-174, 1996.

[20] R. E. Bellman, "On a Routing Problem," Quart. Applied Math, vol. 16, pp. 87-90, 1958.

[21] L. R. Ford, Jr. and D. R. Fulkeison, "Flows in Networks," Princeton Univ. Press, Princeton, NJ, 1962.

[22] E. F. Moore, "The Shortest Path Through a Maze," In Proc. of the Int. Symp. on the Theory of Switching, pp. 285-292. Harward University Press, 1959.

[23] J. Y. Yen, "An Algorithm for Finding Shortest Routes from All Source Nodes to a Given Destination in General Networks," Quart. Applied Math, vol. 27, pp. 526-530, 1970.

[24] 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.

[25] L. Huelsbergen, "A Representation for Dynamic Graphs in Reconfigurable Hardware and Its Application to Fundamental Graph Algorithms," in Proc. of Field Programmable Gate Arrays, pp. 105-115, 2000.

[26] G.R. Jagadeesh, T. Srikanthan and C.M. Lim, "Field programmable gate array-based acceleration of shortest-path computation," IET Computers and Digital Techniques, vol. 5, iss. 4, pp. 231-237, 2011.

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

[28] D. Tong and V. K. Prasanna, "Dynamically Configurable Online Statistical Flow Feature Extractor on FPGA," in Proc. of High Performance Extreme Computing Conference (HPEC), pp. 1-6, 2013.

[29] "DDR3 SDRAM MT41J128M16 - 16 Meg $\times$ 16 $\times$ 8 Banks," http://www.micron.com/products/dram/ddr3-sdram, Micron Technology, 2006.

[30] G. Yuan and T. Aamodt, "A Hybrid Analytical DRAM Performance Model," in Proc of the 36th annual international symposium on Computer architecture, 2009.