2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines

High-throughput and Energy-efficient Graph Processing on FPGA

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

***Abstract*—In this paper, we propose a novel design for large-scale graph processing on FPGA. Our design uses large external memory for storing massive graph data and FPGA for acceleration, and leverages edge-centric computing principles. We propose a data layout which optimizes the external memory performance and leads to an efficient memory activation schedule to reduce on-chip memory power consumption. Further, we develop a parallel architecture on FPGA which can saturate the external memory bandwidth and concurrently process multiple input data to increase throughput. We use our design to accelerate several classic graph algorithms, including single-source shortest path, weakly connected component, and minimum spanning tree. Experimental results show that for all the considered graph algorithms, our design achieves high throughput of over 600 million traversed edges per second (MTEPS) and high energy-efficiency of over 30 MTEPS/W. Compared with a baseline design, our optimizations result in over 3.6× throughput and 5.8× energy-efficiency improvements, respectively. Our design achieves 32% throughput improvement when compared with state-of-the-art FPGA designs, and up to 7.8× speedup when compared with state-of-the-art multi-core implementation.**

I. Introduction

Graph processing has become increasingly important in many real-world applications, such as genome analysis and social networks [1]. However, obtaining high-performance for large-scale graph processing is challenging due to: (1) the datasets of graph problems are massive and can easily overwhelm the computational and memory capabilities of the target platform [2]; (2) graph problems exhibit poor spatial and temporal locality of memory accesses [3]; therefore, the runtime is dominated by external memory accesses [3]. Edge-centric graph processing [3] and vertex-centric graph processing [4] have been proposed to solve large-scale graph problems. While vertex-centric graph processing randomly accesses edges through pointers stored with vertices [4], edge-centric graph processing directly accesses edges from external memory in a streaming fashion [3]. For graphs with the property that the edge set is much larger than the vertex set, edge-centric graph processing is often advantageous compared to vertex-centric graph processing [3].

Throughput and energy-efficiency are key performance metrics, especially for large-scale problems [5], [6]. FPGA has become an attractive platform to offer acceleration and achieve high performance with low power consumption for many applications [7-9]. Recently, there has been increased focus on accelerating large-scale graph processing using FPGA [10-19]. However, the performance of external memory system remains the bottleneck due to the irregular memory access pattern of graph problems [10]. Moreover, energy-efficiency optimizations have not been explored. It is still challenging to develop high-throughput and energy-efficient FPGA design for large-scale graph processing.

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.

In this paper, we present an FPGA design for large-scale graph processing that optimizes external memory performance and at the same time is energy-efficient. We conduct comprehensive experiments to evaluate the performance with respect to throughput and energy-efficiency based on a state-of-the-art FPGA. Our main contributions are:

- We propose an optimized data layout for edge-centric graph processing, which minimizes the number of random external memory accesses and enables an efficient memory activation schedule to reduce memory power.
- We develop a parallel architecture on FPGA which saturates the external memory bandwidth and achieves high clock rate (>200 MHz) for various graph problems.
- We evaluate our design and show that it achieves high throughput of 600-1000 MTEPS and high energy-efficiency of 30-50 MTEPS/W for large-scale graph processing. This corresponds to over 3.6× higher throughput and 5.8× higher energy-efficiency compared with a baseline design, respectively.
- Our design achieves 32% throughput improvement compared with state-of-the-art FPGA designs, and up to 7.8× speedup compared with state-of-the-art multi-core implementation.

The rest of the paper is organized as follows. Section II covers background and related work. Section III presents our target system. Section IV introduces our optimization techniques. Section V discusses architecture implementation. Section VI reports experimental results. Section VII concludes the paper.

978-1-5090-2356-1/16 $31.00 © 2016 IEEE
DOI 10.1109/FCCM.2016.35

II. BACKGROUND AND RELATED WORK

A. *Edge-centric Graph Processing*

Edge-centric graph processing harnesses a scatter-gather programming model and can be applied to a variety of graph algorithms [3]. The processing is structured in a number of iterations, each consisting of a scatter phase followed by a gather phase. In the scatter phase, each edge produces a *message*, which carries the data of the source vertex of the edge and is used to update the destination vertex of the edge in the gather phase. In the gather phase, all the messages produced in the previous scatter phase are iterated over to update the corresponding destination vertices. Algorithm 1 shows a general template of edge-centric graph processing. The computation complexity of each iteration is $O(|V|+|E|)$, where $|V|$ denotes the number of vertices and $|E|$ denotes the number of edges in the graph. Compared with vertex-centric graph processing [4], the main advantage of edge-centric graph processing is that random accesses to the edges are avoided. For large-scale graphs, for which the edge set is much larger than the vertex set, edge-centric graph processing usually achieves superior performance than vertex-centric graph processing [3].

Algorithm 1 Edge-centric Graph Processing

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

B. *FPGA-based Graph Processing*

Recently, using FPGA for graph processing has sparked great research interest and achieved considerable speedup compared with multi-core CPU and GPGPU systems [10-19]. However, most of existing designs are application specific and do not address energy-efficiency [10-16]. Instead, our design serves a broader range of graph algorithms. GraphStep [17] and GraphGen [18] are FPGA frameworks based on vertex-centric graph processing and support a variety of graph algorithms. However, neither GraphStep nor GraphGen explores energy-efficiency optimizations. In [19], graph processing accelerators that address energy-efficiency are proposed on FPGA-based SoCs. However, [19] is designed for sparse graph problems, while our design solves dense graph problems as well.

III. System Overview and DRAM Access

Our design is based on a general system [20] (shown in Fig. 1) that consists of FPGA and large external memory. We target DRAM as external memory to store massive graph data. State-of-the-art DRAM (e.g. DDR4 SDRAM) provides high peak bandwidth, but the performance depends on the access pattern [22]. In many cases, the sustained bandwidth is much lower than the peak bandwidth [22], making DRAM performance the main bottleneck of the target system.

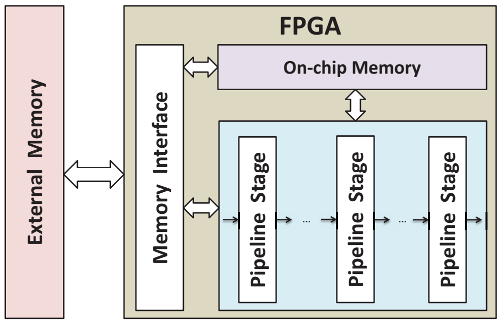

Fig. 1: System model

A DRAM chip is organized into banks, each consisting of a two-dimensional matrix of locations ([row, column]) [22]. A row of a bank needs to be first activated to enable accesses [22]. An access in an activated row is defined as *row-hit*. When there is an access to a different row (i.e. other than the activated row), the activated row must be closed and the row in which the data resides must be activated. This results in additional access latency and additional power consumption [22]. Such access is defined as *row-conflict*.

There are two common DRAM access patterns: sequential and random. For sequential access pattern, consecutive memory accesses map to the same row of DRAM. For random access pattern, consecutive memory accesses usually map to different rows of DRAM, resulting in considerable row-conflicts [22]. Note that sequential access pattern also results in row-conflicts when accesses switch to a different row. We define the row-conflict due to sequential access pattern as *compulsory row-conflict*, and the row-conflict due to random access pattern as *non-compulsory row-conflict*, respectively. Our focus is to minimize the number of non-compulsory row-conflicts.

IV. Data layout and Power Optimization

We use single-source shortest path (SSSP) algorithm [21] as an example to illustrate our optimization techniques. The ideas extend to other edge-centric graph algorithms as well. SSSP finds the shortest path from a source vertex to all the other vertices in the graph. Each vertex maintains the weight of the shortest path from the source vertex to itself. The data of each edge include source vertex index, destination vertex index and edge weight.

A. Edge-centric Graph Processing based on Partitioning

Since vertex data are randomly accessed for edge-centric graph processing (see Algorithm 1), it is desirable to store vertex data in the on-chip memory of FPGA (BRAMs). When the on-chip memory resources of FPGA are not sufficient to store all the vertex data, we use the partitioning approach in [3] to partition the graph data; then both scatter phase and gather phase are processed partition by partition. Assuming the on-chip memory can store the data of m vertices, the graph is partitioned into $\lceil \frac{|V|}{m} \rceil$ partitions. The i-th partition maintains a vertex set including m vertices whose indices are from $i \times m$ to $(i+1) \times m - 1$ $(0 \leq i < \lceil \frac{|V|}{m} \rceil)$. Each partition also has an edge list and a message list. The edge list stores all the edges whose **source** vertices are in the partition's vertex set. The message list stores all the messages whose **destination** vertices are in the partition's vertex set. The edge list of each partition remains fixed during the entire computation; the data of message list are recomputed in every scatter phase; the data of vertex set are updated in every gather phase. Fig. 2 shows an example data layout after the graph data are partitioned into three partitions. Note that the data of each vertex are uniform in size for edge-centric graph processing [3]; thus, the memory requirement for each vertex set is identical. Edge lists and message lists can be different in size; the memory requirement for each edge list depends on the number of edges whose source vertices are in the corresponding vertex set; the memory requirement for each message list depends on the number of edges whose destination vertices are in the corresponding vertex set. Algorithm 2 illustrates edge-centric SSSP after the graph is partitioned.

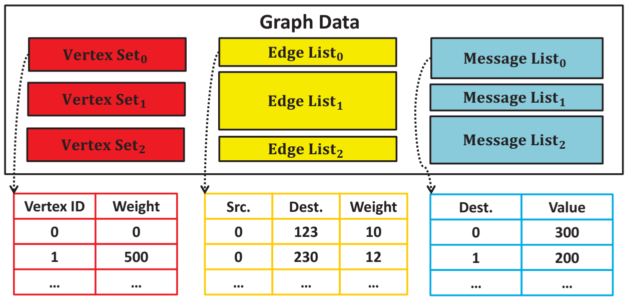

Fig. 2: Data layout after partitioning

B. Data Layout Optimization

1) Minimizing the number of non-compulsory row-conflicts: In Algorithm 2, reading vertices (Line 4, 16), edges (Line 5) and messages (Line 17) from DRAM and writing vertices (Line 24) into DRAM follow sequential access pattern; only writing messages (Line 11) into DRAM in the scatter phase follows random access pattern. This is because the produced messages are written into DRAM based on their destination vertices, which can belong to any message list in DRAM. In the worst case, writing messages into DRAM in the scatter phase results in $O(|E|)$ non-compulsory row-conflicts. Fig. 3 shows an example in which writing every message into DRAM results in a non-compulsory row-conflict.

Algorithm 2 Edge-centric SSSP

```
1: while not done do
2:   Scatter:
3:   for each partition do
4:     read vertex set from DRAM and store into BRAM
5:     for each edge $e$ in edge list (read from DRAM) do
6:       read weight of Vertex $e.src$ from BRAM
7:       let $a$ = weight of Vertex $e.src$
8:       produce a message $msg$
9:       $msg.value = e.weight + a$
10:      $msg.dest = e.dest$
11:      write $msg$ into message list of the partition whose
         vertex set contains Vertex $msg.dest$ in DRAM
12:    end for
13:  end for
14:  Gather:
15:  for each partition do
16:    read vertex set from DRAM and store into BRAM
17:    for each message $msg$ in message list (read from
       DRAM) do
18:      read weight of Vertex $msg.dest$ from BRAM
19:      let $b$ = weight of Vertex $msg.dest$
20:      if $msg.value < b$ then
21:        update weight of Vertex $msg.dest$ in BRAM
22:      end if
23:    end for
24:    write vertex set into DRAM
25:  end for
26: end while
```

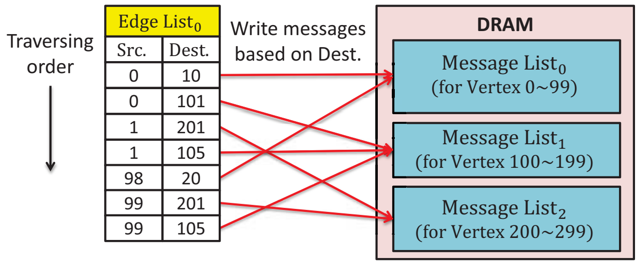
Traversing order
Edge List$_0$
Src. | Dest.
0 | 10
0 | 101
1 | 201
1 | 105
98 | 20
99 | 201
99 | 105
Write messages based on Dest.
DRAM
Message List$_0$ (for Vertex 0~99)
Message List$_1$ (for Vertex 100~199)
Message List$_2$ (for Vertex 200~299)

Fig. 3: Row-conflicts due to writing messages into DRAM

In order to minimize the number of non-compulsory row-conflicts due to writing messages into DRAM, we propose an **optimized data layout**: the edge list of each partition is sorted based on the destination vertices; if the destination veritices of two edges are identical, sorting is based on the source vertices. The computation complexity for our optimized data layout is $O(\sum_{i=0}^{k-1} |E_i| \log|E_i|)$, where $|E_i|$ denotes the number of edges in the edge list of the i-th partition.

Theorem 4.1: In the scatter phase, based on our optimized data layout, the number of non-compulsory row-conflicts due to writing messages into DRAM is $O(k^2)$, where k is the total number of partitions.

Proof: The destination vertices of messages are the same as the destination vertices of the traversed edges. Since each edge list has been sorted based on the destination vertices, the messages based on each edge list are also produced in a sorted order. Thus, the messages whose destination vertices belong to the same partition are produced consecutively and written into the same message list in DRAM. Non-compulsory row-conflict only occurs when a message belonging to a different partition (i.e. other than the partition that the previous message belongs to) is produced. Therefore, writing the messages produced by traversing one edge list results in $O(k)$ non-compulsory row-conflicts. Since scatter phase traverses k edge lists, the total number of non-compulsory row-conflicts is $O(k^2)$, which is far less than $O(|E|)$ when k is a small number. In Fig. 4, we show our optimized data layout for the example of Fig. 3.

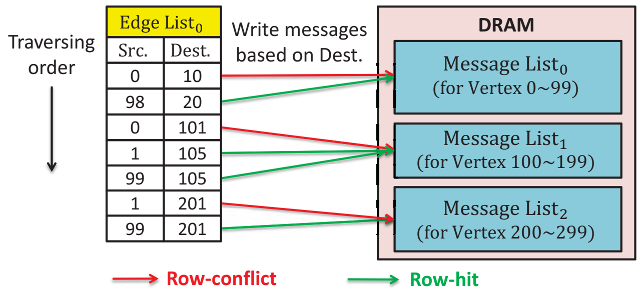
Traversing order
Edge List$_0$
Src. Dest.
0 10
98 20
0 101
1 105
99 105
1 201
99 201
Write messages based on Dest.
DRAM
Message List$_0$ (for Vertex 0~99)
Message List$_1$ (for Vertex 100~199)
Message List$_2$ (for Vertex 200~299)
Row-conflict
Row-hit

Fig. 4: Optimized data layout for Fig. 3

2) Message combination mechanism: Due to our optimized data layout, in the scatter phase, messages having the same destination vertex are produced consecutively. We propose to combine such messages before writing them into DRAM, in order to reduce the number of messages to be processed in the following gather phase. For example, for SSSP, combining multiple messages with the same destination vertex is performed by selecting the message that has the smallest value.

C. Memory Power Optimization

Our system is designed to handle large graph datasets. Thus, memory power is a significant component of the overall power. We optimize the memory power in order to improve the energy-efficiency of the entire system.

1) BRAM Power Optimization: FPGA power consists of static power, memory power, I/O power, clock power and logic power. For the designs which consume large amount of on-chip memory resources, on-chip memory power dominates the overall FPGA power consumption. Since it is desirable to reduce the number of partitions k by fully utilizing the on-chip memory resources [3], our design uses large amount of on-chip memory resources, which in turn leads to high on-chip memory power consumption.

The on-chip memory in our design consists of a number of BRAM modules, each storing the same amount of vertex data. In order to reduce BRAM power consumption, we implement a BRAM activation approach which selectively activates and deactivates BRAM modules through the 'enable' port of BRAM [24]. Using this approach, a BRAM module is activated when the accessed data is stored in it, otherwise it is deactivated to save the BRAM power. Our optimized data layout results in spatial locality and enables an efficient BRAM activation schedule.

Theorem 4.2: In the scatter phase, traversing the edge list of a partition activates each BRAM module at most w times, where w is the number of distinct destination verices that appear in the edge list of a partition.

Proof: In the scatter phase, reading vertex data from on-chip memory is performed based on the source vertices of the traversed edges. Recall our optimized data layout sorts the edge list based on the destination verices; when two edges have the same destination vertex, the sorting is based on the source vertices. Thus, the edges with the same destination vertex are not only stored together, but also in a sorted order based on the source vertices. This leads to an access pattern in which during a certain amount of time, the source vertices of the traversed edges are accessed from the same BRAM module. Hence, traversing the edges with the same destination vertex activates each BRAM module at most once. When there are w distinct destination vertices that appear in the edge list, each BRAM module is activated at most w times $(0 < w \leq |V|)$.

Theorem 4.3: In the gather phase, traversing the message list of a partition activates each BRAM module at most k times, where k is the total number of partitions.

Proof: In the gather phase, accessing vertex data is based on the destination vertices of messages. Let $s_{i,j}$ denote all the messages which are produced by the edge list of Partition i and written into the message list of Partition j $(0 \leq i, j < k)$. The messages of $s_{i,j}$ are in a sorted order based on the destination vertices due to our proposed data layout (see the proof of Theorem 4.1). Thus, traversing $s_{i,j}$ in the gather phase results in one BRAM module being repeatedly accessed at a time, and each BRAM module being activated at most once. For Partition q $(0 \leq q < k)$, its message list is the union of $s_{0,q}$, $s_{1,q}$, ..., $s_{k-1,q}$. Since traversing $s_{i,q}$ $(0 \leq i < k)$ activates each BRAM module at most once, traversing the union of $s_{0,q}$, $s_{1,q}$, ..., $s_{k-1,q}$ activates each BRAM module at most k times.

2) DRAM Power Optimization: DRAM power consists of access power (i.e. read power and write power), activation power, and background power [23]. The activation power is consumed when row-conflict occurs [23]. As discussed in Section IV-B-1, our optimized data layout reduces the number of non-compulsory row-conflicts from $O(|E|)$ to $O(k^2)$, thus saving the activation power.

V. ARCHITECTURE IMPLEMENTATION

A. Overall Architecture

The overall architecture of our design is depicted in Fig. 5. We detail each component in the following sections.

B. Controller

The architecture of controller is shown in Fig. 6. The progress tracker uses counters to keep track of the processing progress, including the current phase (scatter or gather), which

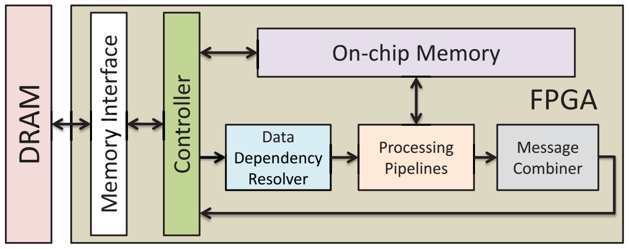

Fig. 5: Overall system architecture

partition is being processed and whether the termination condition of the algorithm is satisfied. Based on the processing progress, the progress tracker generates a control signal to inform the processing pipelines what the input data represent (edges or messages) and instruct the processing pipelines to perform the corresponding logical operations. The address generator is responsible for determining the DRAM access type (read or write) and computing the DRAM access addresses. The buffer is used to temporally store the data to be written into DRAM. When the buffer is full, FPGA stops reading from DRAM and begins writing the buffered data into DRAM. The goal of including the buffer is to avoid reading edges from DRAM and writing messages into DRAM at the same time.

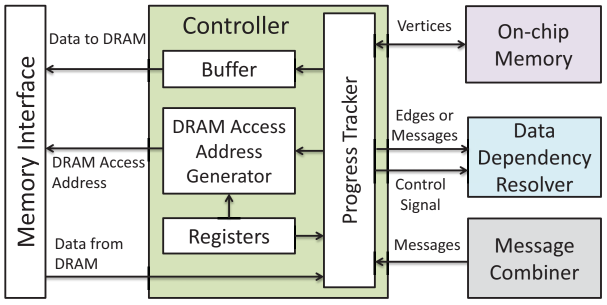

Fig. 6: Architecture of controller

C. Processing Pipelines

To increase the data parallelism of the architecture (denoted as p), we implement p ($p \geq 1$) processing pipelines working in parallel. In the scatter phase, p edges are concurrently processed per clock cycle, and in the gather phase, p messages are concurrently processed per clock cycle. The data parallelism p is constrained by the DRAM bandwidth. Specifically, $1 \leq p \leq \frac{BW}{u*r}$, where BW denotes the peak DRAM bandwidth, u denotes the data width of each input data, and r denotes the clock rate of FPGA. In Fig. 7, we show the processing pipelines for p=2. As shown, each processing pipeline contains three stages, including vertex read stage, computation stage, and vertex write stage.

In the scatter phase, at each clock cycle, each processing pipeline takes one edge as the input data, and the vertex read stage reads the source vertex of the edge (Line 6 of Algorithm 2). Then the computation stage produces the message (Line

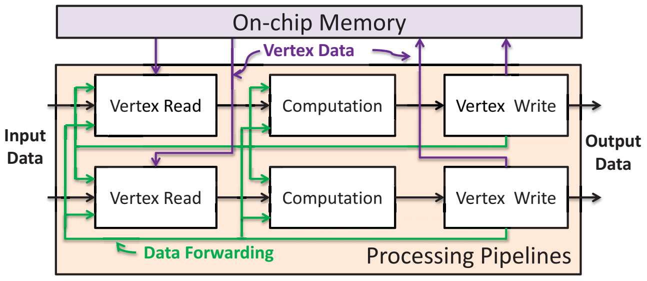

Fig. 7: Processing pipelines for $p = 2$

7-10 of Algorithm 2). Note that the vertex write stage just outputs the message since there is no need to update vertices in the scatter phase.

In the gather phase, at each clock cycle, each processing pipeline takes one message as the input data, and the vertex read stage reads the destination vertex of the message (Line 18 of Algorithm 2). Then the computation stage checks whether the message results in an update (Line 20 of Algorithm 2), in which case the vertex write stage updates the vertex data in the on-chip memory (Line 21 of Algorithm 2). Since there is delay for writing updated vertex data into the on-chip memory, read-after-write data hazard may occur at the vertex read stage and computation stage. To handle read-after-write data hazard without stalling the pipelines, we add data forwarding circuits, from each vertex write stage to all the vertex read stages and computation stages, to forward the most recent updated vertex data ($O(p^2)$ complexity).

D. On-chip Memory

The on-chip memory consists of a number of multi-ported BRAM modules. We use the approach in [26] to implement the multi-ported BRAM modules, each of which has p read ports and p write ports (denoted as pR/pW). As a result, p processing pipelines can read and write the on-chip memory concurrently. For such a pR/pW BRAM module, the memory requirement is $O(p^2)$ [26]. We also include an 'enable' port for each BRAM module, through which the BRAM module can be activated or deactivated [24]. To hide the latency of accessing vertex data from DRAM (Line 4 and 16 of Algorithm 2), we use double buffering technique: the on-chip memory is evenly divided into two chunks; while one chunk stores the vertex set of the partition which is being processed, the controller pre-fetches the vertex set of the next partition to process into the other chunk.

E. Data Dependency Resolver

In the gather phase, if multiple messages that enter the processing pipelines at the same clock cycle have the same destination vertex, conflict occurs when multiple processing pipelines concurrently update the same vertex in the on-chip memory. To address such data dependency, we implement a combining network (CN) as data dependency resolver, to combine the messages that are to be processed at the same clock cycle and have the same destination vertex. The CN sorts

p input messages based on the destination vertices in a bitonic sorting fashion [25]; during sorting, if two messages are found to have the same destination vertex, the two messages are combined as one message. Thus, the data dependency resolver ensures that the messages entering the processing stages at the same clock cycle have distinct destination vertices. When p is a power of 2, CN contains (1+log$p$)log$p$/2 pipeline stages and each pipeline stage has $\frac{p}{2}$ comparators [25]. Fig. 8 shows the architecture of CN for p=4.

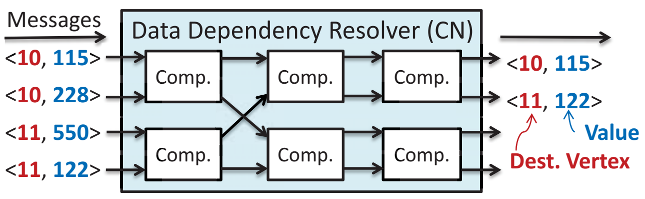

Fig. 8: Combining network for $p = 4$

F. Message Combiner

The message combiner is used to combine the messages which have the same destination vertex and are produced consecutively in the scatter phase (Section IV-B2). The message combiner first uses the CN architecture presented in Section V-E to combine the messages that are produced at the same clock cycle, then includes one more stage based on Algorithm 3 to combine the messages that are produced at different clock cycles.

Algorithm 3 Last pipeline stage of message combiner

```
1:  Initialization: $reg\_msg$ = null
2:  Input $msg_0$, ..., $msg_{q-1}$ from previous stage ($1 \leq q \leq p$)
3:  if $msg_0.dest$ = $reg\_msg.dest$ and $q = 1$ then
4:      $reg\_msg$ = Combine($msg_0$, $reg\_msg$)
5:  else if $msg_0.dest$ = $reg\_msg.dest$ and $q > 1$ then
6:      $msg_0$ = Combine($msg_0$, $reg\_msg$)
7:      Output $msg_0$, ..., $msg_{q-2}$ to controller
8:      $reg\_msg$ = $msg_{q-1}$
9:  else
10:     Output $reg\_msg$, $msg_0$, ..., $msg_{q-2}$ to controller
11:     $reg\_msg$ = $msg_{q-1}$
12: end if
```

VI. Experimental Results

A. Experimental Setup

The experiments were conducted based on the Xilinx Virtex UltraScale xcvu160flgb2104 with -2L speed grade. The target device has 926,400 slice LUTs, 1,852,800 slice registers, and up to 12.8 MB BRAMs. The clock rate, resource utilization, and power of FPGA are evaluated using Xilinx Vivado 2015.2. The designs were verified by post-place-and-route simulations and the reported results are post-place-and-route results. We use Micron 2GB DDR4 SDRAM (MTA8ATF51264HZ-2G3) as the external memory. The target DRAM operates at 1.2 GHz and has a peak bandwidth of 19.2 GB/s. We evaluate DRAM performance including sustained bandwidth and power using DRAMSim2 [28], a widely used tool to evaluate DRAM performance for the target system [29], [30].

We use our design to study three classic graph algorithms, including single-source shortest path (SSSP), weakly connected component (WCC), and minimum spanning tree (MST). The graph datasets for the experiments (Table I) are real-life graphs obtained from [27]. We assume that graph does not change during runtime, and graph data have been pre-processed and stored in DRAM based on our proposed data layout (Section IV-B). The pre-processing can be achieved by traversing the entire edge set for partitioning in $O(|E|)$ time, and then sorting the edge list of each partition using streaming sorting network [31], [32] in $O(|E|)$ time. We use throughput (millions of edges traversed per second (MTEPS)) and energy-efficiency (throughput per Watt (MTEPS/W)) as our main performance metrics.

TABLE I: Graph datasets

Name	$\|V\|$	$\|E\|$	Description
com-Youtube	1.1M	2.8M	Social network
wiki-Talk	2.3M	4.8M	Web graph
cit-Patents	3.6M	15.7M	Citation network
soc-LiveJournal	4.7M	65.8M	Social network

B. Resource Utilization and Clock Rate

In Table II, we show the resource utilization and clock rate of the FPGA designs for p=8, which maximize throughput given the peak DRAM bandwidth. As p increases from 1 to 8, we observe slight clock rate degradation, which is due to more complex implementation for data forwarding circuits ($O(p^2)$ complexity) and multi-ported BRAM modules ($O(p^2)$ complexity) for larger p. The on-chip memory of our design has the capacity to store the data of 32K vertices. Since the on-chip memory is used for double buffering (Section V-D), the vertex set of each partition contains 16K vertices. The main bottleneck to support a larger vertex set for each partition is due to the BRAM resource limitation.

TABLE II: Resource utilization and clock rate

Algorithm	Slice LUT	Register	BRAM	Clock rate
SSSP	40.6%	21.5%	93.7%	230 MHz
WCC	29.2%	16.7%	64.5%	255 MHz
MST	31.8%	17.4%	64.5%	249 MHz

C. Throughput and Energy-efficiency

We show the performance with respect to throughput and energy-efficiency in Table III. For all the graph algorithms and datasets in our experiments, our design achieves high throughput of 600-1000 MTEPS and high energy-efficiency of 30-50 MTEPS/W. The achieved energy-efficiency is among the top 30 in the Green Graph 500 benchmark list [6], which maintains the most energy-efficient systems for graph processing.

TABLE III: Throughput and energy-efficiency

Alg.	Dataset	Throughput (MTEPS)	Power (W) DRAM	Power (W) FPGA	Energy-eff. (MTEPS/W)
SSSP	com	708	0.49	23.73	29.2
	wiki	657			27.1
	cit	687			28.3
	soc	872			36.0
	Average	731			30.2
WCC	com	854	0.49	17.42	47.7
	wiki	779			43.5
	cit	747			41.7
	soc	1068			59.5
	Average	862			48.1
MST	com	840	0.49	18.57	44.1
	wiki	766			40.1
	cit	732			38.4
	soc	1043			54.7
	Average	845			44.3

D. Comparison with Baseline Design

To show the effectiveness of our optimizations, we compare our optimized design with a baseline design, which uses the data layout proposed in [3] (Section IV-A), and does not include our data layout optimization (Section IV-B) or power optimization (Section IV-C). Table IV summarizes the comparison results based on the average performance for our graph datasets. We observe that, our data layout optimization results in at least 18.2× reduction of non-compulsory row-conflicts, which in turn leads to a higher sustained DRAM bandwidth and less DRAM power consumption. Our optimized design achieves at least 3.6× higher throughput than the baseline design.

We show the power consumption comparison in Fig. 9. We observe that our optimized design reduces the BRAM power by over 10×. As a result, the total power consumption is reduced by over 2×, and the energy-efficiency of the entire system is improved by up to 8.9×.

In Fig. 10, we show the runtime comparison based on the dataset 'soc'. Our optimized design reduces runtime by 3.8× and 1.5×, for the scatter phase and gather phase, respectively. The runtime reduction of scatter phase is due to our data layout optimization (Section IV-B1) that reduces the number of non-compulsory row-conflicts. The runtime reduction of gather phase is due to our message combining mechanism (Section IV-B2) which reduces the number of messages to be processed. We found our observations to hold for all the datasets in our experiments, hence we consider them to be robust.

TABLE V: Comparison with FPGA-based Design

Approach	Platform	Peak BW (GB/s)	Throughput per FPGA (MTEPS)
[11]	4 FPGAs+DRAM	80.0	∼550
Ours	FPGA+DRAM	19.2	∼730

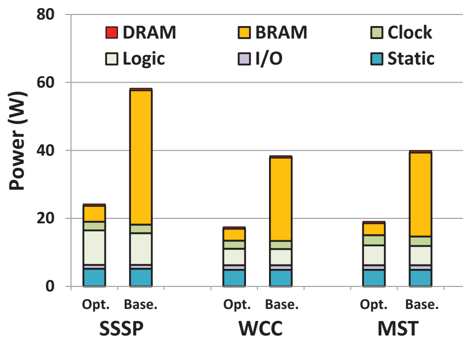

Fig. 9: Power consumption comparison

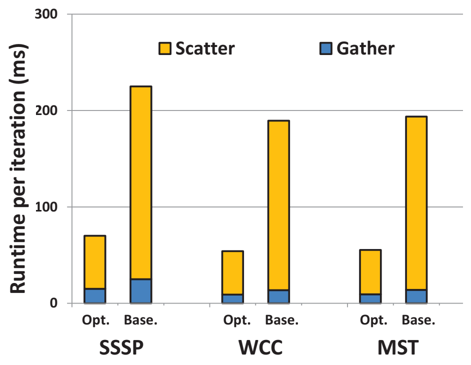

Fig. 10: Runtime comparison

E. Comparison with State-of-the-art FPGA Design

We compare our design with one state-of-the-art FPGA design for graph processing [11]. Table V summarizes the comparison results based on SSSP performance. Our design achieves 1.32× higher throughput.

F. Comparison with State-of-the-art Multi-core Design

We compare our FPGA design with a highly optimized multi-core implementation for edge-centric graph processing [3]. The target platform of [3] is dual-socket AMD Opteron 6272 with 32 physical cores (2.1 GHz); the main memory has a peak bandwidth of 25 GB/s. We summarize the results in Table VI. Our design achieves 1.7×-7.8× speedup.

Although FPGA has a lower clock rate than multi-core system, FPGA is more advantageous for edge-centric graph processing due to: (1) DRAM accesses for multi-core implementation need go through cache hierarchies, while FPGA directly accesses data from DRAM; (2) cache pollution may occur for multi-core implementation, resulting in useful vertex data being evicted from on-chip memory.

TABLE VI: Comparison with multi-core implementation

Algorithm	Throughput (MTEPS) FPGA	Throughput (MTEPS) Multi-core	Speedup
SSSP	731	434	1.7×
WCC	862	110	7.8×
MST	845	139	6.0×

TABLE IV: Performance comparison with baseline design

Alg.	Approach	Sustained BW (GB/s)	Non-compulsory row-conflict rate	Throughput (MTEPS)	MTEPS Imprv.	Power (W) DRAM	Power (W) FPGA	Energy-eff. (MTEPS/W)	MTEP/W Imprv.
SSSP	Optimized	15.3	0.8%	731	3.6×	0.49	23.73	30.2	8.9×
	Baseline	5.3	14.8%	202		0.51	58.52	3.4	
WCC	Optimized	17.3	0.8%	862	3.7×	0.49	17.42	48.1	5.8×
	Baseline	6.0	15.7%	235		0.51	38.13	8.2	
MST	Optimized	16.9	0.8%	845	3.7×	0.49	18.57	44.3	5.8×
	Baseline	5.9	15.8%	230		0.51	39.76	7.6	

VII. Conclusion

In this paper, we presented an FPGA design for large-scale edge-centric graph processing. We proposed a data layout which optimized the DRAM performance and resulted in an efficient memory activation schedule to reduce on-chip memory power consumption. We developed a parallel architecture which sustained high DRAM bandwidth and was energy-efficient. Our design achieves high throughput of 600-1000 MTEPS and high energy-efficiency of 30-50 MTEPS/W for three classic graph algorithms. Compared with a baseline design, our proposed optimizations result in at least 3.6× throughput improvement and 5.8× energy-efficiency improvement, respectively. Compared with state-of-the-art multi-core implementation, our design achieves up to 7.8× speedup. In the future, we plan to include solid state drive in the system to handle even larger graph datasets.

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. 520-535, 2007.

[3] A. Roy, I. Mihailovie and W. Zwaenepoel, "X-Stream: Edge-centric Graph Processing using Streaming Partitions," in Symposium on Operating Systems Principles (SOSP), pp. 472-488, 2013.

[4] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in ACM International Conference on Management of data (SIGMOD), pp. 135-146, 2010.

[5] R. Nambiar and M. Poess, "Performance Evaluation and Benchmarking," Springer-Verlag Berlin Heidelberg, ISBN 978-3-642-18205-1.

[6] "Sixth Green Graph 500 List," http://green.graph500.org/lists.php

[7] J. Matai, J Kim, and R. Kastner, "Energy Efficient Canonical Huffman Encoding," in IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 202-209, 2014.

[8] A. Putnam, A. M. Caulfield, E. S. Chung, et. al, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in International Symposium on Computer Architecture (ISCA), pp. 13-24, 2014.

[9] S. R. Kuppannagari, R. Chen, A. Sanny, S. G. Singapura, G. C. Tran, S. Zhou, Y. Hu, S. P. Crago, and V. K. Prasanna, "Energy Performance of FPGAs on PERFECT Suite Kernels," in IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-6, 2014.

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

[11] O. G. Attia, T. Johnson, K. Townsend, P. Jones and J. Zambreno, "CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search," in International Parallel and Distributed Processing Symposium Workshop (IPDPSW), pp. 228-235, 2014.

[12] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating Large-Scale Single-Source Shortest Path on FPGA," in International Parallel and Distributed Processing Symposium Workshop (IPDPSW), pp. 129-136, 2015.

[13] 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 International Conference on Field Programmable Logic and Applications (FPL), pp. 99-104, 2012.

[14] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing Memory Performance for FPGA Implementation of PageRank," in International Conference on Reconfigurable Computing and FPGAs (ReconFig), pp. 1-6, 2015.

[15] G. Lei, Y. Dou, R. Li, and F. Xia, "An FPGA Implementation for Solving Large Single Source Shortest Path Problem," in IEEE Transactions on Circuits and Systems II: Express Briefs, 2015.

[16] G. Dai, Y. Chi, Y. Wang, and H. Yang, "FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search," in International Symposium on Field-Programmable Gate Arrays, pp. 105-110, 2016.

[17] M. Delorimier, N. Kapre, N. Mehta, and A. Dehon, "Spatial Hardware Implementation for Sparse Graph Algorithms in GraphStep," in ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 6, iss. 3 , pp.17:1-17:20, 2011.

[18] 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 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 25-28, 2014.

[19] N. Kapre and P. Moorthy, "A Case for Embedded FPGA-based SoCs in Energy-Efficient Acceleration of Graph Problems," in Supercomputing Frontiers and Innovations, pp. 76-86, 2015.

[20] S. Neuendorffer and K. Vissers, "Streaming Systems in FPGAs," Embedded Computer Systems: Architectures, Modeling, and Simulation, pp.147-156, 2008.

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

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

[23] "Calculating Memory System Power for DDR3," http://www.micron.com/~/media/Documents/Products/Technical%20Note/DRAM/TN41_01DDR3_Power.Pdf

[24] "Reducing Power Consumption in Xilinx FPGAs," http://vhdlguru.blogspot.com/2011/07/

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

[26] C. E. LaForest and J. G. Steffan, "Efficient Multi-Ported Memories for FPGAs," in International Symposium on Field Programmable Gate Arrays (FPGA), pp. 41-50, 2010.

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

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

[29] B. Akn, F. Franchetti, and J. C. Hoe, "Understanding the Design Space of DRAM-Optimized Hardware FFT Accelerators," in IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 248-255, 2014.

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

[31] M. Zuluaga, P. A. Milder, and M. Puschel, "Computer Generation of Streaming Sorting Networks," in Design Automation Conference (DAC), pp. 1245-1253, 2012.

[32] R. Chen, S. Siriyal, and V. K. Prasanna, "Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA," in International Symposium on Field Programmable Gate Arrays (FPGA), pp. 240-249, 2015.