An FPGA Framework for Edge-Centric Graph Processing

Shijie Zhou
University of Southern California
Los Angeles, CA 90089
shijiezh@usc.edu

Rajgopal Kannan
US Army Research Lab
Los Angeles, CA 90094
Rajgopal.kannan.civ@mail.mil

Hanqing Zeng
University of Southern California
Los Angeles, CA 90089
zengh@usc.edu

Viktor K. Prasanna
University of Southern California
Los Angeles, CA 90089
prasanna@usc.edu

ABSTRACT

Many emerging real-world applications require fast processing of large-scale data represented in the form of graphs. In this paper, we design a Field-Programmable Gate Array (FPGA) framework to accelerate graph algorithms based on the edge-centric paradigm. Our design is flexible for accelerating general graph algorithms with various vertex attributes and update propagation functions, such as Sparse Matrix Vector Multiplication (SpMV), PageRank (PR), Single Source Shortest Path (SSSP), and Weakly Connected Component (WCC). The target platform consists of large external memory to store the graph data and FPGA to accelerate the processing. By taking an edge-centric graph algorithm and hardware resource constraints as inputs, our framework can determine the optimal design parameters and produce an optimized Register-Transfer Level (RTL) FPGA accelerator design. To improve data locality and increase parallelism, we partition the input graph into non-overlapping partitions. This enables our framework to efficiently buffer vertex data in the on-chip memory of FPGA and exploit both inter-partition and intra-partition parallelism. Further, we propose an optimized data layout to improve external memory performance and reduce data communication between FPGA and external memory. Based on our design methodology, we accelerate two fundamental graph algorithms for performance evaluation: Sparse Matrix Vector Multiplication (SpMV) and PageRank (PR). Experimental results show that our accelerators sustain a high throughput of up to 2250 Million Traversed Edges Per Second (MTEPS) and 2487 MTEPS for SpMV and PR, respectively. Compared with several highly-optimized multicore designs, our FPGA framework achieves up to 20.5× speedup for SpMV, and 17.7× speedup for PR, respectively; compared with two state-of-the-art FPGA frameworks, our designs demonstrate up to 5.3× and 1.8× throughput improvement for SpMV and PR, respectively.

KEYWORDS

FPGA framework; Energy-efficient acceleration; High-throughput graph processing

ACM Reference Format:
Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K. Prasanna. 2018. An FPGA Framework for Edge-Centric Graph Processing. In *CF '18: CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3203217.3203233

1 INTRODUCTION

Graphs have become increasingly important for representing real-world networked data in emerging applications, such as the World Wide Web, social networks, genome analysis, and medical informatics [1]. To facilitate the processing of large graphs, many graph processing frameworks have been developed based on general purpose processors [1–6, 34–37]. These frameworks provide high-level programming models for the users to easily perform graph processing. They also focus on optimizing cache performance and exploiting thread-level parallelism to increase throughput. However, general purpose processors are not the ideal platform for graph processing [7, 8]. They induce several inefficiencies including (1) wasted external memory bandwidth due to inefficient memory access granularity (i.e., loading and storing entire cacheline data while operating on only a portion of the data) and (2) ineffective on-chip memory usage due to the poor spatial and temporal locality of graph algorithms. To address these inefficiencies, dedicated hardware accelerators for graph processing have recently gained lots of interest [7–23].

With the increased interest in energy-efficient acceleration, Field-Programmable Gate Array (FPGA) has become an attractive platform to develop accelerators [24, 25]. State-of-the-art FPGA devices, such as UltraScale+ FPGAs [26], provide dense logic elements (up to 5.5 million), abundant user-controllable on-chip memory resources (up to 500 Mb), and interfaces for various external memory technologies (e.g., hybrid memory cube [16]). Amazon Web Service has recently launched FPGA-based cloud instances to allow customers to develop FPGA accelerators for complex applications. FPGAs have also been introduced into data centers to provide customized acceleration of computation-intensive tasks [25]. Prior works that accelerate graph processing on FPGA have shown significant speedup and energy improvement over general purpose processors [9–11, 14–17]. However, most of these FPGA accelerators are algorithm-specific and require high development effort.

Therefore, developing an FPGA framework for general graph algorithms is becoming a new trend [21–23]. However, existing FPGA frameworks are designed based on the vertex-centric paradigm, which accesses the edges of vertices through pointers or vertex indices. This can result in massive random external memory accesses as well as accelerator stalls [17].

In this paper, we propose an FPGA framework based on the edge-centric paradigm [3]. Different from vertex-centric paradigm, edge-centric paradigm traverses edges in a streaming fashion, making FPGA an ideal platform for the acceleration [27]. Our framework can accelerate general edge-centric graph algorithms and generate the optimized FPGA accelerator which is implemented as parallel pipelines to fully exploit the massive parallelism of FPGA. The main contributions of our work are:

- We propose an FPGA framework for accelerating general graph algorithms using the edge-centric paradigm. We accelerate two fundamental graph algorithms, Sparse Matrix Vector Multiplication (SpMV) and PageRank (PR), to evaluate the performance of our framework.
- We adopt a simple graph partitioning approach to partition the input graph. This enables an efficient use of the on-chip RAMs of FPGA to buffer vertex data. As a result, the processing engines on the FPGA can access the vertex data directly from the on-chip RAMs during the processing.
- Our framework exploits inter-partition and intra-partition parallelism at the same time. Distinct partitions are concurrently processed by distinct processing engines on the FPGA. Each processing engine consists of parallel pipelines to process distinct edges of a partition.
- We also develop a design automation tool, which can produce the synthesizable Verilog RTL of our design based on user's input parameters. The tool allows users to easily and quickly construct graph processing accelerators.
- Experimental results show that our designs achieve a high throughput of up to 2250 MTEPS and 2487 MTEPS for SpMV and PR, respectively. Compared with state-of-the-art FPGA designs, our framework achieves up to 5.3× and 1.8× throughput improvement for SpMV and PR, respectively.

The rest of the paper is organized as follows: Section 2 covers the background; Section 3 presents the framework overview; Section 4 discusses our optimizations; Section 5 describes the implementation detail; Section 6 reports experimental results; Section 7 introduces the related work; Section 8 concludes the paper.

2 BACKGROUND

2.1 Edge-centric Graph Processing

Edge-centric paradigm is flexible for capturing various graph algorithms with different graph structures, data types, and graph update functions [3]. Its computation follows a scatter-gather programming model. As shown in Algorithm 1, the processing is iterative, with each iteration consisting of a scatter phase followed by a gather phase. In the scatter phase, each edge is traversed to produce an algorithm-specific update based on the source vertex of the edge. In the gather phase, all the updates produced in the previous scatter phase are applied to the corresponding destination vertices. The advantage of edge-centric paradigm is that it traverses the edges in a streaming fashion. This makes FPGA an ideal acceleration platform since FPGA has been widely used to accelerate streaming applications [27].

Algorithm 1 Edge-centric Graph Processing

```
1:  while not done do
2:    Scatter phase:
3:    for each edge e do
4:      Produce an update u ← Process_edge(e, v_{e.src})
5:    end for
6:    Gather phase:
7:    for each update u do
8:      Update vertex u.dest ← Apply_update(u, v_{u.dest})
9:    end for
10: end while
```

Vertex-centric paradigm is also widely used to design graph processing frameworks [1]. However, one key issue of vertex-centric paradigm is that traversing the edges requires random memory accesses through indices or pointers [3]. The random memory accesses are highly irregular such that conventional memory controllers are not able to efficiently handle them. In this scenario, accelerator may frequently stall and the performance can significantly deteriorate [17]. Compared with vertex-centric paradigm, edge-centric paradigm completely eliminates the random memory accesses to the edges. Therefore, for large-scale graphs whose edge set is much larger than the vertex set, edge-centric paradigm can achieve superior performance than vertex-centric paradigm [3].

2.2 Data Structures

Edge-centric paradigm uses the coordinate (COO) format to store the input graph [3]. The COO format stores the graph as an edge array which has been sorted based on the source vertices of the edges[1]. Each edge in the edge array is represented as a <*src, dest, weight*> tuple, which specifies the source vertex, the destination vertex, and the weight of the edge, respectively. All the vertices are stored in a vertex array, with each vertex having an algorithm-specific attribute (e.g., PageRank value of the vertex). Each update produced in the scatter phase is represented as a <*dest, value*> pair, in which *dest* denotes the destination vertex of the update and *value* denotes the value of the update. Figure 1 shows the data structures of an example graph[2].

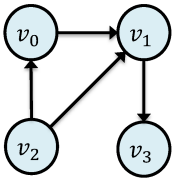
2.3 Algorithms

In this paper, we accelerate two fundamental graph algorithms which are core kernels and building blocks in many applications. This section briefly introduces these two algorithms and shows how each algorithm maps to the edge-centric paradigm.

2.3.1 Sparse Matrix-Vector Multiplication. Sparse matrix-vector multiplication (SpMV) is a widely used computational kernel in scientific applications [10]. Generalized SpMV iteratively computes

[1]For undirected graphs, each edge is represented using a pair of directed edges, one in each direction.

[2]In this example, we assume the value of each update is obtained by multiplying the edge weight and the attribute of the source vertex of the edge



Vertex array

v_{id}	attribute
v_0	0.2
v_1	0.8
v_2	1.5
v_3	4.2

Edge array

src	dest	weight
v_0	v_1	2.0
v_1	v_3	3.0
v_2	v_0	1.0
v_2	v_1	0.2

Updates

dest	value
v_1	0.4
v_3	2.4
v_0	1.5
v_1	0.3

Figure 1: Example graph and its associated data structures

$x^{t+1} = Ax^t = \bigoplus_{i=1}^{n} A_i \otimes x^t$, where A is a sparse $H \times I$ matrix with row vectors A_i, x is a dense vector of size I, $\oplus$ and $\otimes$ are algorithm-specific operators[3], and t denotes the number of iterations that have been completed. When transformed into a graph problem, each non-zero entry of A is represented as a weighted edge, and each element of x is represented as a vertex. Table 1 shows the mapping of SpMV to edge-centric paradigm, where we use $Attr(v)$ to denote the algorithm-specific attribute associated with vertex v.

Table 1: Mapping of SpMV to edge-centric paradigm

Process_edge$(e, v_{e.src})$	Apply_update$(u, v_{u.dest})$
$u.value = e.weight \otimes Attr(v_{e.src})$ $u.dest = e.dest$	$Attr(v_{u.dest}) =$ $Attr(v_{u.dest}) \oplus u.value$

2.3.2 PageRank. PageRank (PR) is used to rank the importance of vertices in a graph [28]. It computes the PageRank value of each vertex which indicates the likelihood that the vertex will be reached. The computation of PageRank is iterative. Initially, each vertex is assigned the same PageRank value. Then, in each iteration, each vertex v updates its PageRank value based on Equation (1), in which d is a constant called damping factor; $|V|$ is the total number of vertices of the graph; 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. Table 2 shows the mapping of PR to edge-centric paradigm.

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

Table 2: Mapping of PR to edge-centric paradigm

Process_edge$(e, v_{e.src})$	Apply_update$(u, v_{u.dest})$
$u.value = \frac{d \times Attr(v_{e.src})}{\#\_outgoing\_edges(v_{e.src})}$ $u.dest = e.dest$	$Attr(v_{u.dest}) =$ $Attr(v_{u.dest}) + u.value$

[3]We use standard addition and multiplication operators for SpMV in this paper.

3 FRAMEWORK OVERVIEW

Our framework targets a system architecture as depicted in Figure 2, which consists of external memory (e.g., DRAM) and FPGA accelerator. The external memory stores all the graph data including vertices, edges, and updates. On the FPGA, the on-chip RAMs are used as buffers to buffer vertex data (see Section 4.1), and the processing logic is implemented as parallel pipelines to process the edges and updates. Given an edge-centric graph algorithm, our framework maps it to the target architecture and customizes the processing logic based on the 'Process_edge()' and 'Apply_update()' functions. Users have the flexibility to decide the architecture parameters of the on-chip RAMs and processing logic based on the hardware resource constraints. Our framework also provides a design automation tool (see Section 5.3) to generate the Verilog code of the FPGA accelerator based on the architecture parameters.

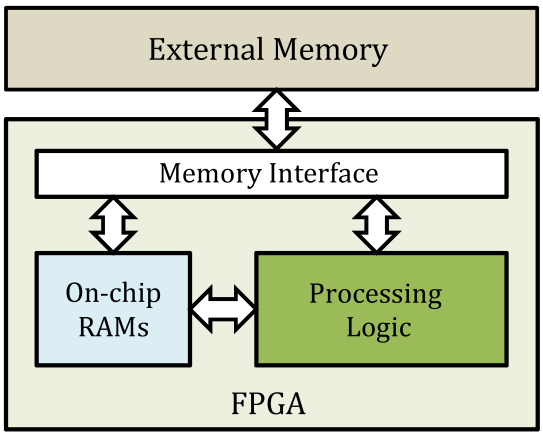


Figure 2: Target architecture of our framework

4 ALGORITHMIC OPTIMIZATIONS

In order to to improve the performance of our framework, we propose three optimizations, including (1) partitioning the input graph to enable vertex buffering (Section 4.1), (2) parallelizing the execution of edge-centric paradigm (Section 4.2), and (3) combining updates to reduce data communication (Section 4.3).

4.1 Vertex Buffering

In each iteration, vertex data (e.g., $Attr(v)$ in Tables 1 and 2) are repeatedly accessed and updated. Therefore, we propose to buffer the vertex data in the on-chip RAMs, which offer fine-grained single-cycle accesses to the processing logic. For large graphs whose vertex array cannot fit in the on-chip RAMs, we partition the graph using a simple vertex-index-based partitioning approach [29] to ensure that the vertex data of each partition fit in the on-chip RAMs.

Assuming the on-chip RAMs can store the data of m vertices, we partition the input graph into $k = \lceil \frac{|V|}{m} \rceil$ partitions, where $|V|$ denotes the total number of vertices in the graph. Firstly, the vertex array is partitioned into k vertex sub-arrays; the i-th vertex sub-array includes m vertices whose vertex indices are between $i \times m$ and $(i+1) \times m - 1$ $(0 \leq i < k)$. We define each vertex sub-array

as an **interval**. Secondly, the edge array is partitioned into k edge sub-arrays; the i-th edge sub-array includes all the edges whose source vertices belong to the i-th interval. We define each edge sub-array as a **shard**. The i-th shard and the i-th interval constitute the i-th **partition**. Each partition also maintains an array to store the updates whose destination vertices belong to the interval of the partition. We define the update array of each partition as a **bin**. Note that the data of each shard remain fixed during the entire processing[4]; the data of each bin are recomputed in every scatter phase; the data of each interval are updated in every gather phase. Figure 3 shows the data layout after the graph in Figure 1 is partitioned into two partitions and the interval of each partition has 2 vertices (i.e., $k = 2, m = 2$).

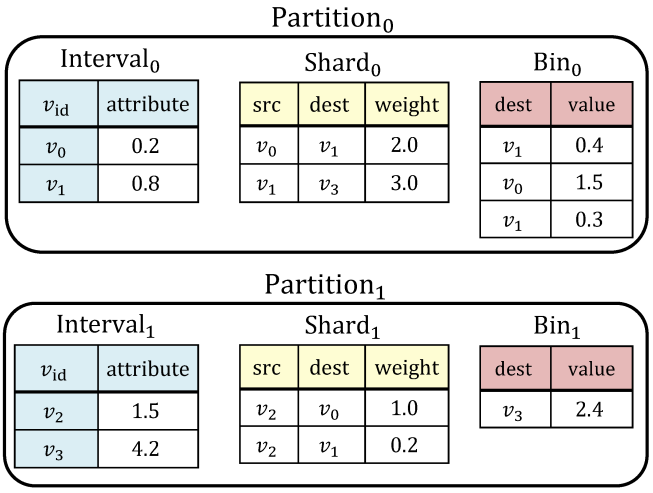


Figure 3: Data layout after graph partitioning

Algorithm 2 illustrates the computation of edge-centric paradigm after graph partitioning. All the intervals, shards, and bins are stored in the external memory. Before a partition being processed, all the data of its interval are pre-fetched and buffered in the on-chip RAMs. Then, edges (updates) are streamed from the external memory during the scatter (gather) phase. Due to the vertex buffering, the vertex data to access in Lines 6 and 13 of Algorithm 2 have already been buffered into on-chip RAMs by executing Lines 4 and 11, respectively; therefore, the processing logic can directly access them from the on-chip RAMs, other than from the external memory.

4.2 Parallelization Strategy

4.2.1 Inter-partition Parallelism. Partitioning the input graph also increases the available parallellism since that distinct partitions can be concurrently processed. We define the parallelism to concurrently process distinct partitions as **inter-partition parallelism**. Assuming the processing logic consists of p ($p \geq 1$) Processing Engines (PEs), our framework can independently process p partitions in parallel. The inter-partition parallelism of the design is denoted as p. When a PE completes the processing of a partition, it is automatically assigned another partition to process.

[4]We assume the edges of the input graph do not alter during the processing.

Algorithm 2 Edge-centric graph processing based on partitioning

```
1:  while not done do
2:    Scatter phase:
3:    for i from 0 to k − 1 do
4:      Read Interval_i into on-chip RAMs
5:      for each e ∈ Shard_i do
6:        u ← Process_edge(e, v_{e.src})
7:      end for
8:    end for
9:    Gather phase:
10:   for i from 0 to k − 1 do
11:     Read Interval_i into on-chip RAMs
12:     for each u ∈ Bin_i do
13:       Apply_update(u, v_{u.dest})
14:     end for
15:     Write Interval_i into external memory
16:   end for
17: end while
```

4.2.2 Intra-partition Parallelism. Inside each processing engine, we employ parallel pipelines to concurrently process distinct edges (updates) of each shard (bin) during the scatter (gather) phase. We define the parallelism to concurrently process distinct edges or updates of a partition as **intra-partition parallelism**. Assuming each processing engine has q ($q \geq 1$) parallel pipelines (see Section 5.2), q distinct edges (updates) of the same partition can be concurrently processed by the processing engine during the scatter (gather) phase per clock cycle. The intra-partition parallelism of the design is denoted as q.

4.3 Update Combination Mechanism

In the scatter phase, because traversing each edge produces an update, the total number of produced updates is equal to the number of edges $|E|$. Therefore, $|E|$ updates are written into the external memory in each scatter phase. In order to reduce the data communication for writing updates into the external memory in the scatter phase, we propose an **update combination mechanism** to combine the updates that have the same destination vertex before writing them into the external memory. To enable the update combination mechanism, we propose an optimized data layout by sorting the edges of each shard based on the destination vertices. Due to this data layout optimization, in the scatter phase, the updates that have the same destination vertex are produced consecutively. Thus, consecutive updates that have the same destination vertex can be easily combined as one update and then written into the external memory. For example, for PR, combining two updates is performed by summing them up. Note that this optimization also reduces the number of updates to be processed in the gather phase. Therefore, the data communication of the gather phase is reduced as well.

5 IMPLEMENTATION DETAIL

5.1 Architecture Overview

We show the top-level architecture of our FPGA design in Figure 5. The DRAM connected to the FPGA is the external memory which stores all the intervals, shards, and bins. There are p processing

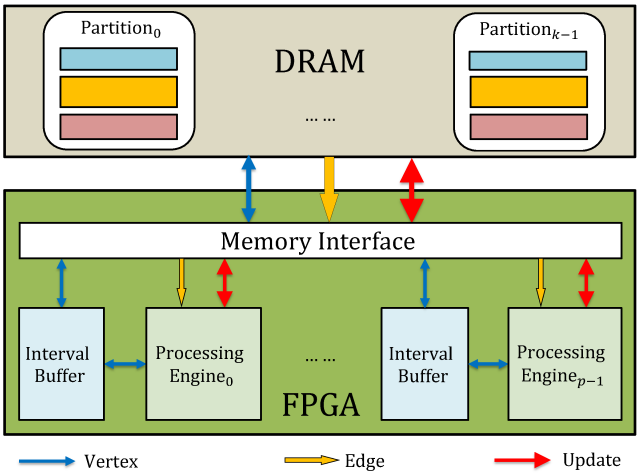


Figure 5: Top-level architecture

engines (PEs) on the FPGA, which can be customized based on the target graph algorithm. These PEs process p distinct partitions in parallel. Each PE has an individual interval buffer, which is constructed by on-chip URAMs (i.e., UltraRAMs) and used to store the interval data of the partition being processed by the PE. In the scatter phase, the PEs read edges from the DRAM and write updates into the DRAM. In the gather phase, the PEs read updates from the DRAM and write updated vertices into the DRAM.

5.2 Processing Engine

Figure 4 depicts the architecture of the processing engine (PE). Each PE employs q processing pipelines ($q \geq 1$), thus is able to concurrently process q input data in each clock cycle. Each processing pipeline has three stages, including vertex read stage, computation stage, and vertex write stage.

In the scatter phase, the input data represent edges. In each clock cycle, each processing pipeline takes one edge as input; then, the vertex read stage reads the data of the source vertex of the edge from the interval buffer; the computation stage produces the update based on the edge weight and the source vertex (i.e., the 'Process_edge()' function in Algorithm 2). Note that the vertex write stage and hazard detector do not work during the scatter phase; this is because the scatter phase only has read accesses to the vertices. All the updates produced by the processing pipelines are fed into an update combining network. The update combining network employs parallel Compare-and-Combine (CaC) units to combine the input updates based on their destination vertices in a bitonic sorting fashion [30]. Figure 6 depicts the architecture of the update combining network for $q = 4$. Each CaC unit compares the destination vertices of the two input updates. If the two updates have the same destination vertex, they are combined as one update; otherwise, the two updates are sorted based on their destination vertices and output to the next pipeline stage. When q is a power of 2, the update combining network contains $(1+\log q)\cdot \log q \cdot q/4$ CaC units.

In the gather phase, the input data represent updates. In each clock cycle, each processing pipeline takes one update as input; then, the vertex read stage reads the data of the destination vertex of the update from the interval buffer; the computation stage computes the updated data of the destination vertex (i.e., the 'Apply_update()' function in Algorithm 2); at last, the vertex write stage writes the updated data of the destination vertex into the interval buffer. Since the gather phase performs both read and write accesses to the vertex data, read-after-write data hazard may occur. In order to handle the possible data hazard, we implement a data hazard detector using a fine-grained locking mechanism. For each vertex in the partition

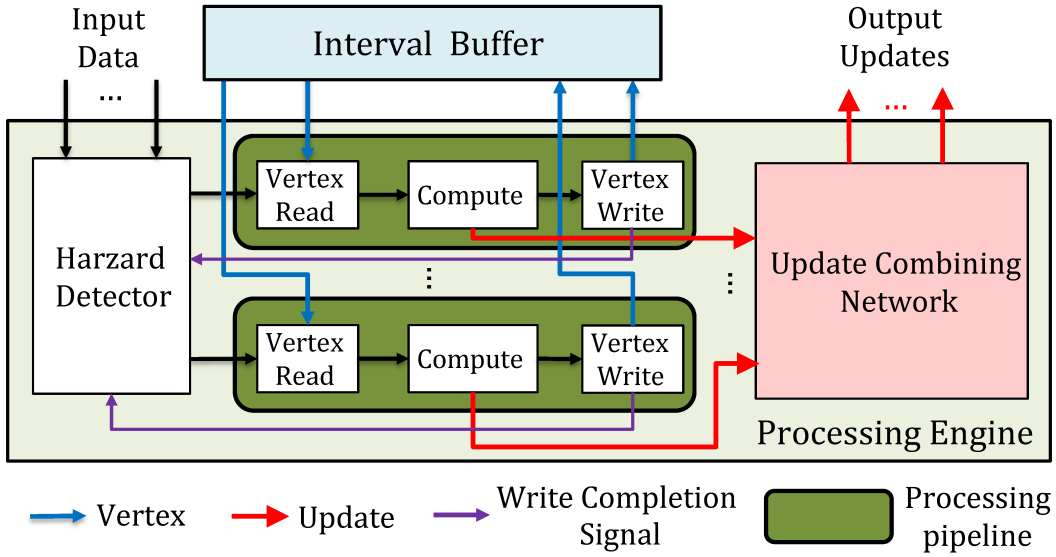


Figure 4: Architecture of processing engine

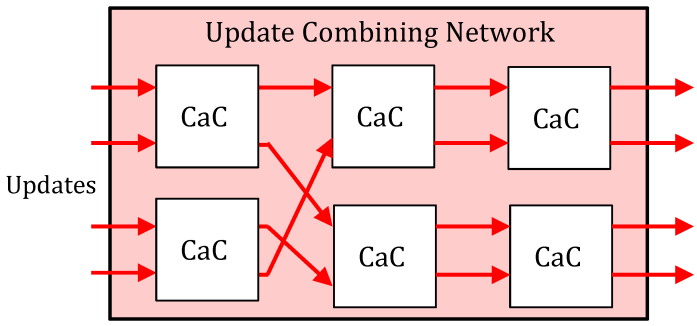


Figure 6: Update combining network for $q = 4$

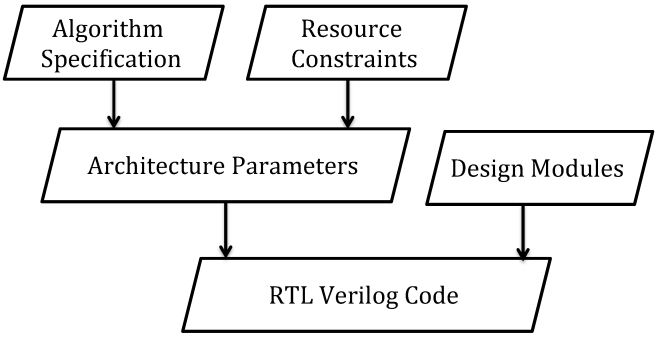


Figure 7: Development flow of design automation tool

being processed, the hazard detector stores a 1-bit flag in Block RAMs (BRAMs). A flag with value 1 means the attribute of the corresponding vertex is being computed by one of the processing pipelines, and thus cannot be read at this time. For each input update, the hazard detector checks the flag of the destination vertex: if the flag is 0, the update is fed into the processing pipeline and the flag is set to 1; otherwise, the pipeline stalls until the flag becomes 0. Note that when the processing pipeline writes any updated vertex data into the interval buffer, it also sends a vertex-write-completion signal to the hazard detector to set the flag of the corresponding vertex back to 0. Therefore, deadlock will not occur.

5.3 Design Automation Tool

We have built a design automation tool to allow users to rapidly generate the FPGA accelerators based on our framework. Figure 7 illustrates the development flow of the tool. First, users need define the algorithm parameters of the target edge-centric graph algorithms (e.g., data width of each vertex attribute) and provide resource constraints of the FPGA design; for example, users can specify how much block RAM resource can be used. Then, based on the input information, the framework determines the architecture parameters (e.g, the number of processing engines) and generates the corresponding design modules, such as hazard detector, processing pipelines, and update combining network, etc,. Finally, the tool connects these design modules to produce the RTL design of the FPGA accelerator, which automatically includes our proposed optimizations such as vertex buffering, parallel pipelined processing, and update combining.

6 PERFORMANCE EVALUATION

6.1 Experimental Setup

The experiments are conducted using the Xilinx Virtex UltraScale+ xcvu3pffvc1517 FPGA with -2L speed grade [26]. The target FPGA device has 394,080 slice LUTs, 788,160 slice registers, 25 Mb of BRAMs, and 90 Mb of UltraRAMs. Four DDR3 SDRAM chips are used as the external memory, with each chip having a peak access bandwidth of 15 GB/s. To evaluate our designs, we perform post-place-and-route simulations using Xilinx Vivado Design Suite 2017.2. Table 3 summaries the key characteristics of the graphs used in the experiments. These are real-life graphs and have been widely used in the related work [3–6, 22, 23].

Table 3: Real-life graph datasets used for evaluation

Dataset	# Vertices $\|V\|$	# Edges $\|E\|$	Description
WK [31]	2.4 M	5.0 M	Wikipedia network
LJ [32]	4.8 M	69.0 M	Social network
TW [33]	41.6 M	1468.4 M	Twitter network

For SpMV, each edge is represented using 96 bits (32-bit *src*, 32-bit *dest*, and 32-bit *weight*); each vertex attribute is a 32-bit floating-point number (IEEE 754 single precision); each update is represented using 64 bits (32-bit *value* and 32-bit *dest*). For PR, each edge is represented using 64 bits (32-bit *src* and 32-bit *dest*); each vertex is represented using 64 bits (32-bit *attribute* and 32-bit to record the number of outgoing edges of the vertex); each update is represented using 64 bits (32-bit *value* and 32-bit *dest*).

6.2 Performance Metrics

We use the following performance metrics for the evaluation.

- Resource utilization: the resource utilization of the target FPGA device, including logic slices, registers, on-chip RAMs, and DSPs
- Power consumption: the total power consumed by the FPGA accelerator
- Execution time: the average execution time per iteration
- Throughput: the number of Traversed Edges Per Second (TEPS) [22, 23]

6.3 Resource Utilization and Power Consumption

We empirically set the number of PEs to 4 (p=4), the number of pipelines in each PE to 8 (q=8), and the interval size to 128K (m=128K) to maximize the processing throughput and saturate the external memory bandwidth. For both SpMV and PR, the FPGA accelerators sustain a high clock rate of 200 MHz. Table 4 reports the resource utilization and power consumption of the FPGA accelerators.

Table 4: Resource utilization and power consumption

Algorithm	LUT (%)	Register (%)	DSP (%)	BRAM (%)	URAM (%)	Power (Watt)
SpMV	30.4	23.8	2.8	8.3	40.0	7.4
PR	30.0	23.5	2.8	8.3	40.0	6.0

6.4 Execution Time and Throughput

We report the execution time performance and throughput performance in Table 5. We observe that our FPGA designs achieve a high throughput of up to 2250 MTEPS and 2487 MTEPS for SpMV and PR, respectively.

Table 5: Execution time and throughput

Algorithm	Dataset	Average T_{exec} per iteration (ms)	Throughput (MTEPS)
SpMV	WK	5.0	1004
	LJ	36.2	1906
	TW	652.5	2250
PR	WK	4.5	1116
	LJ	32.7	2110
	TW	590.4	2487

6.5 Impact of Update Combination

To show the effectiveness of our proposed update combination mechanism and optimized data layout, we compare with a baseline FPGA design. The baseline design uses the standard data layout of the COO format without our data layout optimization. Figure 8 shows that our proposed optimization reduces the volume of data communication by 1.3× to 1.8×.

6.6 Comparison with State-of-the-art

6.6.1 Comparison with State-of-the-art Multi-core Design. We first compare the performance of our design with several highly-optimized multi-core designs [3–6]. Table 6 shows the results of the comparison based on the same datasets. It can be observed that our FPGA designs achieve up to 20.5× and 17.7× higher throughput for SpMV and PR, respectively. In addition, the power consumption of our FPGA designs (< 10 Watt) are much lower than multi-core platforms (typically > 80 Watt). Hence, from energy-efficiency perspective, our framework achieves even larger improvement.

Compared with multi-core platforms, FPGA has the following advantages: (1) the external memory accesses for multi-core implementations need go through cache hierarchies, while FPGA accelerators can directly stream data from the external memory; (2) cache pollution may occur for multi-core implementation, resulting in useful vertex data being evicted from cache; while the on-chip RAMs of FPGA is fully user-controllable; (3) when using multi-threading technique, multi-core implementations may require expensive atomic operations to prevent race conditions (e.g., memory locks), which can significantly result in additional overhead.

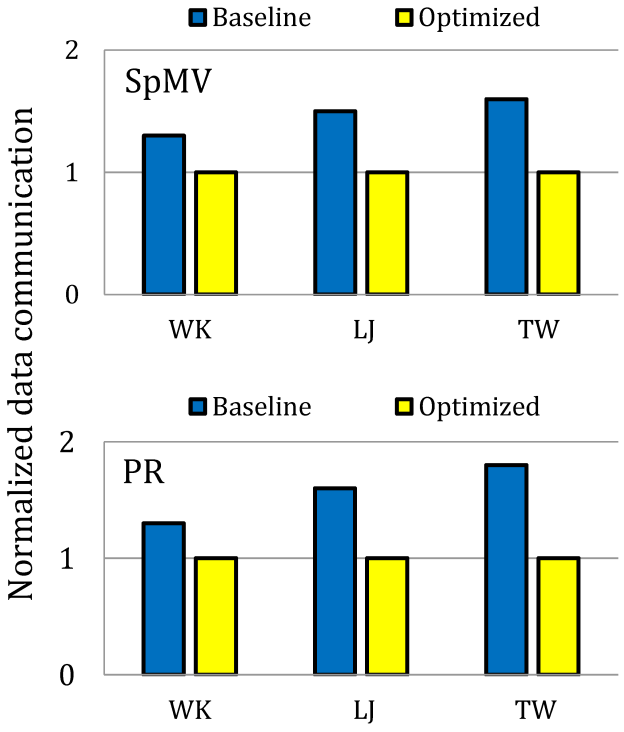


Figure 8: Data communication reduction

Table 6: Comparison with state-of-the-art multi-core designs

Algorithm	Dataset	Approach	Throughput (MTEPS)	Improvement
SpMV	LJ	[3]	93	1.0×
		This paper	1906	20.5×
PR	LJ	[3]	119	1.0×
		[6]	1530	12.9×
		This paper	2110	17.7×
	TW	[4]	408	1.0×
		[5]	716	1.8×
		[6]	815	2.0×
		This paper	2487	6.1×

6.6.2 Comparison with State-of-the-art FPGA-based Design. We further compare our proposed framework with two state-of-the-art FPGA designs [22, 23]. Both of the FPGA designs aim to accelerate general graph algorithms. Table 7 summarizes the results of the comparison. Compared with [23], our design achieves 5.3× throughput improvement for SpMV. Compared with [22], our design achieves 1.2× to 1.8× throughput improvement for PR.

Table 7: Comparison with state-of-the-art FPGA-based designs

Algorithm	Dataset	Approach	Throughput (MTEPS)	Improvement
SpMV	WK	[23]	190	5.3×
		This paper	1004	
PR	WK	[22]	965	1.2×
		This paper	1116	
	LJ	[22]	1193	1.8×
		This paper	2110	
	TW	[22]	1856	1.3×
		This paper	2487	

7 RELATED WORK

7.1 Software Graph Processing Frameworks

Many software-based graph processing frameworks have been developed, such as GraphChi [2], X-Stream [3], PowerGraph [4] and GraphMat [6] on multi-core, and CuSha [35], Gunrock [36], and Graphie [37] on GPU. These frameworks provide high-level programming models to allow programmers to easily perform graph analytics. They also focus on optimizing memory performance and exploiting massive thread-level parallelism. GraphChi [2] is the first graph processing framework on a single multicore platform. It is based on the vertex-centric paradigm and proposes a parallel sliding windows method to handle out-of-core graphs. X-Stream [3] proposes the edge-centric paradigm to maximize the sequential streaming of graph data from disk. GraphMat [6] maps vertex-centric graph computations to high-performance sparse matrix operations. CuSha [35] addresses the limitation of uncoalesced global memory data accesses for GPU graph processing. Gunrock [36] proposes a data-centric processing abstraction which accelerates the frontier operations using GPU. Graphie [37] implements the asynchronous graph-traversal model on GPU to reduce the data communication.

The optimizations proposed in this paper are also applicable to multi-core and GPU platforms. First, the partitioning approach can be performed based on the cache size of multi-core and GPU platforms to improve the cache performance [38]. Second, distinct thread blocks (i.e., groups of threads) can concurrently process distinct partitions, while inside each thread block, distinct threads can process distinct edges or updates in parallel. Third, the update combination mechanism can be performed using a parallel scan operation.

7.2 FPGA-based Graph Processing Accelerators

Using FPGA to accelerate graph processing has demonstrated great success. In [16, 19, 20], Breadth First Search (BFS) is accelerated on FPGA-HMC platforms. The designs achieve a high throughput of up to 45.8 GTEPS and power efficiency of up to 1.85 GTEPS/Watt for scale-free graphs. In [12], an FPGA accelerator for SpMV is proposed based on a specialized CISR encoding approach. The design achieves one third of the throughput performance of a GTX 580 GPU implementation with 9× lower memory bandwidth and 7× less energy. However, many existing FPGA-based accelerators [11, 12, 14–17] are algorithm-specific and cannot be easily extended to accelerate other graph algorithms. GraphGen [21] is an FPGA framework based on the vertex-centric paradigm to accelerate general graph applications. GraphGen pre-processes the input graph by partitioning it into subgraphs and then processes one subgraph at a time. It also provides a compiler for automatic HDL code generation. However, GraphGen requires the vertex data and the edge data of each subgraph to fit in the on-chip memory of FPGA. For large real-life graphs, this can lead to a large number of subgraphs and thus significantly increase the scheduling complexity. GraphOps [23] is a dataflow library for graph processing. It provides several commonly used building blocks for graph algorithms, such as reading the attributes from all the neighbor. However, GraphOps is based on the vertex-centric paradigm and thus suffers random memory accesses to the edges. ForeGraph [22] is a multi-FPGA-based graph processing framework. It partitions the graph and uses multiple FPGAs to process distinct partitions in parallel. However, the performance can be constrained by the communication overhead among the FPGAs.

8 CONCLUSION

In this paper, we presented an FPGA framework to accelerate graph algorithms based on edge-centric paradigm. We partitioned the input graph to enable efficient on-chip buffering of vertex data and increase the parallelism. We further proposed an efficient update combination mechanism to reduce data communication. To facilitate non-FPGA-experts, we also developed a design automation tool for our framework. We accelerated SpMV and PR to study the performance of our framework. Experimental results showed that our framework achieved up to 20.5× and 17.7× speedup compared with highly-optimized multicore designs for SpMV and PR, respectively. Compared with state-of-the-art FPGA frameworks, our design achieved up to 5.3× and 1.8 × throughput improvement for SpMV and PR, respectively. In the future, we plan to evaluate our framework using more fundamental graph algorithms, such as finding connected components and single source shortest path.

ACKNOWLEDGMENTS

This work is supported by the U.S. National Science Foundation grants ACI-1339756 and CNS-1643351. This work is also supported in part by Intel Strategic Research Alliance funding.

REFERENCES

[1] 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 ACM SIGMOD International Conference on Management of data (SIGMOD), pp 135-146, 2010.

[2] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale Graph Computation on Just a PC," in Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 31-46, 2012.

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

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," in Proc. USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016.

[5] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An Efficient Graph Processing System on a Single Machine," in Proc. of International Conference on Data Engineering (ICDE), pp.409-420, 2016.

[6] N. Sundaram, N. Satish, M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High Performance Graph Analytics Made Productive," in Proc. of VLDB Endowment, vol. 8, no. 11, pp. 1214-1225, 2015.

[7] T. Ham, L. Wu, N. Sundaram, N. Satish, and Margaret Martonosi, "Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics," in Proc. of International Symposium on Microarchitecture (MICRO), pp. 1-13, 2016.

[8] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and Energy-efficient Graph Processing on FPGA," in Proc. of International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 103-110, 2016.

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

[10] D. Gregg, C. Mc Sweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA based Sparse Matrix Vector Multiplication using Commodity DRAM Memory", in Proc. of Field Programmable Logic and Applications (FPL), 2007.

[11] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating Large-Scale Single-Source Shortest Path on FPGA," in Proc. of International Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2015.

[12] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt, "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication," in Proc. of International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 36-43, 2014.

[13] 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), 2011.

[14] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing Memory Performance for FPGA Implementation of PageRank," in Proc. of International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2015.

[15] H. Giefers, P. Staar, R. Polig, "Energy-Efficient Stochastic Matrix Function Estimator for Graph Analytics on FPGA," in Proc. of Field Programmable Logic and Applications (FPL), 2016.

[16] J. Zhang, S. Khoram, and J. Li, "Boosting the Performance of FPGA-based Graph Processor using Hybrid Memory Cube: A Case for Breadth First Search," in Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA), 2017.

[17] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in Proc. of Application-Specific Systems, Architectures, and Processors (ASAP), pp. 8-15, 2012.

[18] 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 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW), pp. 228-235, 2014.

[19] J. Zhang and J. Li, "Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform," in Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 229-238, 2018.

[20] S. Khoram, J. Zhang, M. Strange, and J. Li, "Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform," in Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 239-248, 2018.

[21] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martinez, and C. Guestrin, "GraphGen: An FPGA Framework for Vertex-Centric Graph Computation," in Proc. of Field-Programmable Custom Computing Machines (FCCM), pp. 25-28, 2014.

[22] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture," in Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 217-226, 2017.

[23] T. Oguntebi and K. Olukotun, "GraphOps: A Dataflow Library for Graph Analytics Acceleration," in Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 111-117, 2016.

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

[25] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Xiao, D. Burger, J. Larus, G. Gopal, and S. Pope, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in Proc. of International Symposium on Computer Architecture (ISCA), 2014.

[26] "Virtex UltraScale+ FPGA Data Sheet," https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf

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

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

[29] R. Pearce, M. Gokhale, and N. M. Amato, "Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates," in Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 549-559, 2014.

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

[31] J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Predicting Positive and Negative Links in Online Social Networks," in Proc. of the 20th international conference on World Wide Web (WWW), 2011.

[32] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters," in Internet Mathematics 6(1), 2009.

[33] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, A Social Network or a News Media?" in Proc. of the 19th international conference on World wide web (WWW), 2010.

[34] R. Nasre, M. Burtscher, and K. Pingali, "Data-Driven Versus Topology-driven Irregular Computations on GPUs," in Proc. of 27th International Symposium on Parallel & Distributed Processing (IPDPS), 2013.

[35] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric Graph Processing on GPUs," in Proc. of International Symposium on High-performance Parallel and Distributed Computing (HPDC), pp. 239-252, 2014.

[36] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A High-performance Graph Processing Library on the GPU," in Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2016.

[37] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU," in Proc. of International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 233-245, 2017.

[38] K. Lakhotia, R. Kannan, and V. K. Prasanna, "Accelerating PageRank using Partition-Centric Processing," https://arxiv.org/abs/1709.07122