HitGraph: High-throughput Graph Processing Framework on FPGA

Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, *Fellow, IEEE*, Guna Seetharaman, *Fellow, IEEE*, and Qing Wu

Abstract—This paper presents, HitGraph, an FPGA framework to accelerate graph processing based on the edge-centric paradigm. HitGraph takes in an edge-centric graph algorithm and hardware resource constraints, determines design parameters and then generates a Register Transfer Level (RTL) FPGA design. This makes accelerator design for various graph analytics transparent and user-friendly by masking internal details of the accelerator design process. HitGraph enables increased data reuse and parallelism through novel algorithmic optimizations, including (1) an optimized data layout that reduces non-sequential external memory accesses, (2) an efficient update merging and filtering scheme to reduce the data communication between the FPGA and external memory, and (3) a partition skipping scheme to reduce redundant edge traversals for non-stationary graph algorithms. Based on our design methodology, we accelerate Sparse Matrix Vector Multiplication (SpMV), PageRank (PR), Single Source Shortest Path (SSSP), and Weakly Connected Component (WCC). Experimental results show that HitGraph sustains a high throughput of 2076 Million Traversed Edges Per Second (MTEPS) for SpMV, 2225 MTEPS for PR, 2916 MTEPS for SSSP, and 3493 MTEPS for WCC, respectively. Compared with highly-optimized multi-core implementations, HitGraph achieves up to 37.9× speedup. Compared with state-of-the-art FPGA frameworks, HitGraph achieves up to 50.7× throughput improvement.

Index Terms—FPGA framework, graph processing, energy-efficient acceleration

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 machine learning [1], [2], [3], [4]. To facilitate the processing of large graphs, many graph processing frameworks have been developed based on general purpose processors. Representative examples include GraphChi [5], X-Stream [6], GridGraph [10], and GraphMat [11] based on multi-core general purpose processor, and Gunrock [12], nvGRAPH [13], and CuSha [42] based on general purpose graphics processing unit (GPGPU). However, general purpose processors are not the ideal acceleration platform for graph processing [21], [22], [23], [33]. They have 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), (2) ineffective on-chip memory usage due to the poor spatial and temporal locality of graph algorithms, and (3) expensive atomic operations (e.g., memory locks) to prevent the race condition due to concurrent updates by distinct threads. In order to address these inefficiencies, dedicated hardware accelerators for graph processing have gained lots of interest [17], [18], [19], [20], [21], [22], [23], [25], [33], [34], [40].

With the increased interest in energy-efficient acceleration, Field-Programmable Gate Array (FPGA) has become an attractive platform to develop accelerators [14], [15], [16]. FPGAs can achieve higher performance-per-watt than multi-core and GPU platforms [15], [16], and have been introduced into data centers to provide customized acceleration of computation-intensive tasks [27]. Due to the abundant user-controllable on-chip memory resources and the dense programmable logic elements, FPGAs can effectively overcome the inefficiencies inherent in the general purpose processors and thus have been widely explored to accelerate graph processing [28], [29], [40], [48]. However, most of the existing FPGA-based accelerators for graph processing are algorithm-specific and cannot be applied to general graph algorithms. In addition, the development of an optimized FPGA accelerator can involve the programming of Hardware Description Language (HDL), resulting in high development effort. GraphGen [20] and GraphOps [25] are FPGA-based graph processing frameworks that can accelerate general graph algorithms. They also provide design automation tools to facilitate the development of accelerators. However, these frameworks are designed based on the vertex-centric paradigm [3], in which the edges are traversed through pointers or vertex indices. This leads to massive random external memory accesses as well as accelerator stalls [25], [35].

- *S. Zhou and V.K. Prasanna are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089. E-mail: {shijiezh, prasanna}@usc.edu.*
- *R. Kannan is with the US Army Research Lab, Los Angeles, CA 90094. E-mail: Rajgopal.kannan.civ@mail.mil.*
- *G. Seetharaman is with the US Naval Research Laboratory, Washington, DC 20375. E-mail: guna@nrl.navy.mil.*
- *Q. Wu is with the Air Force Research Laboratory, Rome, NY 13441. E-mail: qing.wu.2@us.af.mil.*

Manuscript received 13 Aug. 2018; revised 4 Feb. 2019; accepted 14 Mar. 2019. Date of publication 11 Apr. 2019; date of current version 11 Sept. 2019.
(Corresponding author: Shijie Zhou.)
Recommended for acceptance by B. Veeravalli.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2910068

In this paper, we propose a graph processing framework, named HitGraph, to accelerate general graph algorithms using FPGA. HitGraph is designed based on the edge-centric paradigm [6], in which all the edges are sequentially traversed in a streaming fashion, making FPGA an ideal acceleration platform. Given an edge-centric graph algorithm and user-defined resource constraints (e.g., registers, Block RAMs, etc.) as inputs, HitGraph first determines the architecture parameters through design space exploration, and then generates the RTL design of a highly-optimized FPGA accelerator. Therefore, developers can easily and quickly generate graph processing accelerators using our framework. We summarize the main contributions of this paper below.

- We propose an FPGA framework called HitGraph to accelerate general graph algorithms based on the edge-centric paradigm. We demonstrate the applicability of HitGraph by accelerating Sparse Matrix Vector Multiplication (SpMV), PageRank (PR), Single Source Shortest Path (SSSP), and Weakly Connected Component (WCC).
- We develop a design automation tool, which can automatically produce the synthesizable `Verilog` code of an highly parallel accelerator based on user's inputs.
- In order to improve the performance of the generated accelerators, we propose several novel algorithmic optimizations, including:
  - graph partitioning to enable efficient vertex buffering for data reuse
  - highly parallel execution by exploiting both inter- and intra-partition parallelism
  - data layout optimization to improve external memory performance
  - update combining and filtering mechanism to reduce data communication
  - partition skipping scheme to reduce redundant edge traversals
- Experimental results show that our designs achieve an average throughput of 2,076 MTEPS for SpMV, 2,225 MTEPS for PR, 2,916 MTEPS for SSSP, and 3,493 MTEPS for WCC, respectively.
- Compared with several highly-optimized multi-core implementations, HitGraph achieves up to 20.5×, 35.5×, 5.0×, 37.9× speedup for SpMV, PR, SSSP, and WCC, respectively. Compared with state-of-the-art GPU implementations, HitGraph achieves comparative performance with 4.8× less memory bandwidth and over 20× less power consumption.
- Compared with two state-of-the-art FPGA-based graph processing frameworks, GraphOps [25] and ForeGraph [23], HitGraph demonstrates up to 50.7× and 2.0× throughput improvement, respectively.

The rest of the paper is organized as follows: Section 2 covers the background; Section 3 introduces the framework overview; Section 4 discusses our algorithmic optimizations; Section 5 describes the implementation details; Section 6 presents our design automation tool; Section 7 reports experimental results; Section 8 introduces related work; Section 9 concludes the paper.

2 Background

2.1 Edge-Centric Graph Processing

The edge-centric (EC) paradigm [6] is a well-known graph processing technique, flexible enough in representing a variety of graph algorithms with different graph structures, vertex attributes, and graph update functions [22], [44]. Generic EC computation follows a scatter-gather programming model as illustrated in Algorithm 1. The computation 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 *update* based on the source vertex of the edge. In the gather phase, each update produced in the previous scatter phase is applied to the corresponding destination vertex. The advantage of edge-centric paradigm is that it sequentially traverses all the edges (updates) in the scatter (gather) phase. Such streaming nature of edge-centric paradigm makes FPGA an ideal acceleration platform [39]. However, the edge-centric paradigm can result in redundant edge traversals for some algorithms in which not all the edges are necessary to be traversed in each iteration (e.g., SSSP). Such graph algorithms are called non-stationary graph algorithms (see Section 2.3).

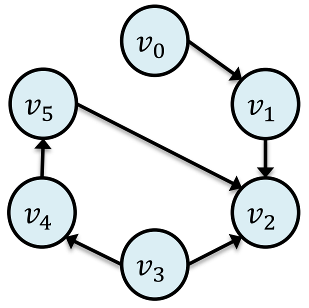
Algorithm 1. Edge-Centric Paradigm

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

The vertex-centric paradigm [4] is also widely used to design graph processing frameworks [20], [25]. It keeps track of a frontier consisting of all the *active vertices* whose attributes have recently been updated and need to be sent to their neighbor vertices. Each vertex keeps a pointer to locate its outgoing edges stored in the external memory. Then, in each iteration, only the active vertices traverse their outgoing edges to update the attributes of their neighbor vertices. However, one key issue of vertex-centric paradigm is that traversing the edges of active vertices requires random external memory accesses through pointers. FPGA has been shown to be inefficient for such pointer-based memory accesses which are too irregular to be efficiently handled by conventional memory controllers [25], [49]. In this scenario, it is very likely to gain no speedup [35]. Compared with the vertex-centric paradigm, the edge-centric paradigm completely eliminates the random memory accesses to the edges. Therefore, for large-scale graphs whose number of edges is much larger than the number of vertices, the edge-centric paradigm can lead to better performance than the vertex-centric paradigm [6], [32].

2.2 Data Structures

The edge-centric paradigm uses the coordinate (COO) format to store the input graph. In this format, all the edges are



Vertex array

v_{id}	attr
v_0	0.7
v_1	1.0
v_2	3.0
v_3	4.5
v_4	1.0
v_5	2.0

Edge array

src	dest	weight
v_0	v_1	2.0
v_1	v_2	3.0
v_3	v_2	1.0
v_3	v_4	0.2
v_4	v_5	0.4
v_5	v_2	3.0

Updates

dest	value
v_1	1.4
v_2	3.0
v_2	4.5
v_4	0.9
v_5	0.4
v_2	6.0

Fig. 1. Example graph and its associated data structures, assuming the value of each update is the product of the edge weight and the attribute of the source vertex of the edge.

stored in an edge array, with each edge represented as a $<$`src, dest, weight`$>$ tuple to specify 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 maintaining 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 associated with the update. Fig. 1 shows the data structures of an example graph .

Although the COO format requires more storage size than the Compressed Sparse Row (CSR) format [25], it leads to streaming accesses to the edges without any indirection such as pointers, and therefore is more favored by hardware accelerators [21], [22], [23]. In addition, the COO format enables data layout optimization of the edges (see Section 4.4) since it does not require the edges to be stored in any specific order.

2.3 Graph Algorithms

We define *active* vertex (in an iteration) as a vertex that has an updated attribute to propagate to its neighbors. Based on the characteristic of the number of active vertices across iterations, graph algorithms can be divided into two categories: *stationary* and *non-stationary* [7]. A graph algorithm is stationary if all the vertices are active in each iteration; otherwise, it is non-stationary. In this paper, we accelerate two representative stationary graph algorithms (i.e., SpMV and PR) and two representative non-stationary graph algorithms (i.e., SSSP and WCC). We have chosen these four algorithms because their computation patterns are general to represent a diverse range of algorithms. For example, the computation patterns of SpMV and PR cover general stationary graph algorithms (e.g., collaborative filtering) and probabilistic graphical models (e.g., belief propagation); the computation pattern of SSSP covers the non-stationary algorithms in which the number of active vertices may increase or decrease (e.g., graph traversal algorithms and graph labeling algorithms), and the computation pattern of WCC covers the non-stationary algorithms in which the number of active vertices keeps decreasing (e.g., community detection). This section briefly introduces the four target algorithms in this paper and shows how to map them to the edge-centric paradigm.

2.3.1 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication is a widely used computational kernel in scientific applications. Generalized SpMV iteratively computes $x^{t+1} = Ax^t = \oplus_{i=0}^{|V|-1} A_i \otimes x^t$, where A is a sparse $|V| \times |V|$ adjacency matrix of a weighted graph, A_i is the ith row vector of A $(0 \leq i < |V|)$, x is a dense vector with $|V|$ values (i.e., one value per vertex), $\oplus$ and $\otimes$ are algorithm specific operators, and t denotes the number of iterations that have been completed. SpMV is a stationary graph algorithm because in each iteration, each vertex updates the value associated with it and thus is active.

2.3.2 PageRank

The PageRank algorithm is a well-known graph analytics algorithm used to rank the importance of vertices in a directed graph. Each vertex maintains an attribute called `PageRank`, which indicates the likelihood that the vertex will be reached. In each iteration, each vertex v sends its `PageRank` to all the neighbors, and then updates its own `PageRank` based on Equation (1), in which d is a constant called damping factor, $|V|$ is the total number of vertices in the graph, v_{nbr} represents the neighbor of v such that v has an incoming edge from v_{nbr}, and L_{nbr} is the number of outgoing edges of v_{nbr}. PR is stationary because all the vertices update their `PageRank` values in each iteration and thus are active in each iteration.

$$PageRank(v) = \frac{1-d}{|V|} + d \times \sum \frac{PageRank(v_{nbr})}{L_{nbr}}. \quad (1)$$

2.3.3 Single Source Shortest Path

Single Source Shortest Path aims to find the shortest paths from a single source vertex to all the other vertices in a weighted graph. It is a key kernel for urban traffic simulation. In this algorithm, each vertex maintains an attribute to record the weight of the shortest path from the source vertex to itself. In the scatter phase of each iteration, all the active vertices send their updated attributes to their neighbors through outgoing edges. Then, in the gather phase, each vertex that receives update(s) from neighbor(s) updates its attribute if a shorter path to the source vertex is found. The algorithm terminates when none of the vertices updates its attribute in an iteration. SSSP is non-stationary because only partial vertices are active in each iteration.

2.3.4 Weakly Connected Component

A Connected Component (CC) of a graph is a subgraph such that (1) there is certain path connecting each pair of

TABLE 1
Mapping of Graph Algorithms to Edge-Centric Paradigm

Algorithm	Produce an *update* $u \leftarrow$ Process_edge(e, $v_{e.src}$)	Update *vertex* $v_{u.dest} \leftarrow$ Apply_update(u)
SpMV	$u.dest \leftarrow e.dest$; $u.value \leftarrow e.weight \otimes attr(v_{e.src})$	$attr(v_{u.dest}) \leftarrow attr(v_{u.dest}) \oplus u.value$
PR	$u.dest \leftarrow e.dest$; $u.value \leftarrow \frac{d \times attr(v_{e.src})}{Num\_of\_outgoing\_edges(v_{e.src})}$	$attr(v_{u.dest}) \leftarrow attr(v_{u.dest}) + u.value$
SSSP	$u.dest \leftarrow e.dest$; $u.value \leftarrow attr(v_{e.src}) + e.weight$	$attr(v_{u.dest}) \leftarrow \min(attr(v_{u.dest}), u.value)$
WCC	$u.dest \leftarrow e.dest$; $u.value \leftarrow attr(v_{e.src})$	$attr(v_{u.dest}) \leftarrow \min(attr(v_{u.dest}), u.value)$

vertices in the subgraph and (2) no additional vertices can be reached by the vertices in the subgraph. Weakly Connected Component aims to find all the connected components in an undirected graph. In this algorithm, each vertex maintains a CC identifier as the attribute to record the connected component that it belongs to. The CC identifier of a connected component is the vertex that has the smallest index in the connected component. In the scatter phase, each active vertex sends its CC identifier to its neighbors. In the gather phase, a vertex updates its CC identifier if it receives a smaller CC identifier than its current CC identifier. When the algorithm terminates, the vertices that have the same attribute (i.e., CC identifier) form a connected component. Since only partial vertices are active in each iteration, WCC is a non-stationary graph algorithm.

2.3.5 Mapping Algorithms to Edge-Centric Paradigm

Table 1 shows the mapping of the discussed graph algorithms to the edge-centric paradigm (Algorithm 1), where we use $attr(v)$ to denote the algorithm-specific attribute associated with the vertex v.

3 Framework Overview

Fig. 2 depicts a high-level view of the system architecture of HitGraph (more details are discussed in Section 5), which consists of external memory (i.e., DRAM) and FPGA. The external memory stores all the graph data including vertices, edges, and updates. On the FPGA, there are p ($p \geq 1$) Processing Engines (PEs) working in parallel to sustain high processing throughput. Each PE is customized based on the target graph algorithm and has a multi-pipelined architecture. The memory controller handles the external memory accesses performed by the PEs. In the scatter phase, the PEs read *edges* from the external memory and write *updates* into the external memory; in the gather phase, the PEs read *updates* from the external memory and write updated *vertices* into the external memory.

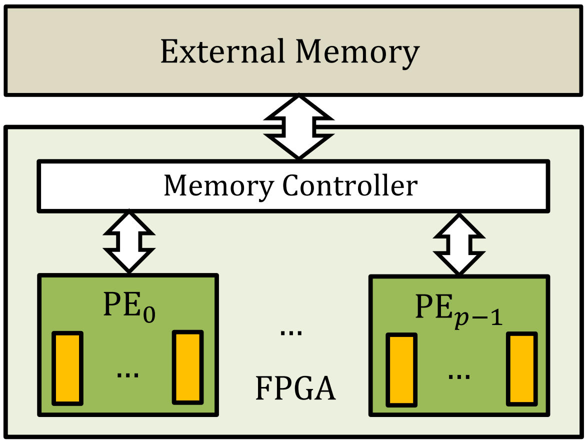


Fig. 2. Target system architecture of HitGraph.

Given a graph algorithm following the edge-centric paradigm, HitGraph maps it to the target architecture and generates the FPGA accelerator by a design automation tool. Fig. 3 illustrates the development flow of HitGraph. First, users provide the inputs to define the algorithm parameters of the edge-centric graph algorithm (e.g., data width of each edge and vertex, etc) and specify the hardware resource constraints of the FPGA design (e.g., how many Block RAMs can be used). Then, based on the user inputs, the framework performs design space exploration to determine the architecture parameters (e.g, the number of processing engines) to maximize the processing throughput. Finally, the design automation tool outputs the `Verilog` code of an FPGA accelerator, which automatically includes our optimizations proposed in Section 4.

4 Algorithmic Optimizations

In order to improve the performance of the generated accelerators by HitGraph, we propose several optimizations to (1) efficiently use the on-chip memory resources, (2) fully take advantage of the massive parallelism provided by FPGA, (3) optimize the performance of external memory, and (4) reduce the data communication between the FPGA and external memory.

4.1 Graph Partitioning and Vertex Buffering

Since the vertex data (i.e., $attr(v)$ in Table 1) are repeatedly accessed and updated in each iteration, we propose to buffer them using the on-chip RAMs of FPGA, which can offer fine-grained low-latency accesses to the PEs. For large graphs whose entire vertex array does not fit in the on-chip RAMs, we use a lightweight graph partitioning approach [22] to partition the graph, such that the vertex data of each partition fit in the on-chip RAMs.

Assuming the data of m vertices can be stored in the on-chip RAMs, we partition the input graph into $k = \lceil \frac{|V|}{m} \rceil$

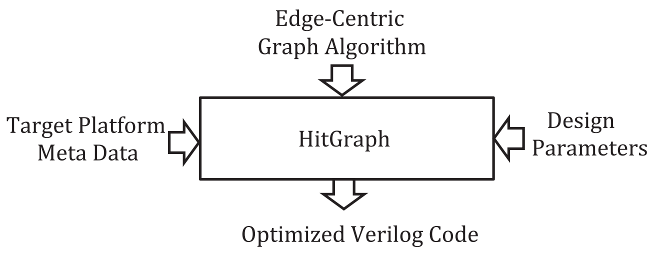
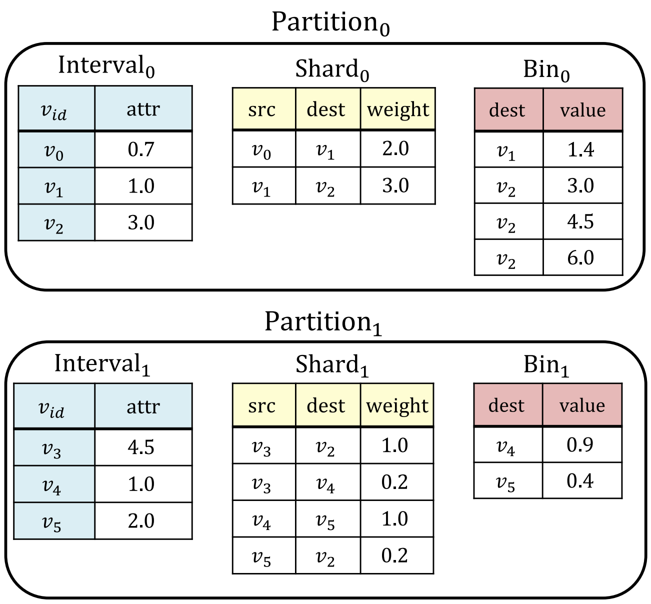


Fig. 3. Framework overview.



Partition_0

Interval_0

v_{id}	attr
v_0	0.7
v_1	1.0
v_2	3.0

Shard_0

src	dest	weight
v_0	v_1	2.0
v_1	v_2	3.0

Bin_0

dest	value
v_1	1.4
v_2	3.0
v_2	4.5
v_2	6.0

Partition_1

Interval_1

v_{id}	attr
v_3	4.5
v_4	1.0
v_5	2.0

Shard_1

src	dest	weight
v_3	v_2	1.0
v_3	v_4	0.2
v_4	v_5	1.0
v_5	v_2	0.2

Bin_1

dest	value
v_4	0.9
v_5	0.4

Fig. 4. Data layout after graph partitioning.

non-overlapping partitions, where $|V|$ denotes the total number of vertices in the graph. We first partition the vertex array into k vertex sub-arrays, such that the ith vertex sub-array contains m vertices whose vertex indices are consecutive and between $i \times m$ and $(i+1) \times m - 1$ $(0 \leq i < k)$. We define each vertex sub-array as an *interval*. After partitioning the vertex array, the edge array is partitioned into k edge sub-arrays, each of which is defined as a *shard*; the ith shard contains all the edges whose source vertices belong to the ith interval (i.e., $\forall$ *edge* $e \in \text{Shard}_i$, $v_{e.src} \in \text{Interval}_i$). The ith shard and the ith interval constitute the ith *partition*. Each partition also maintains an array called *bin* to store the updates whose destination vertices belong to the interval of partition (i.e., $\forall$ *update* $u \in \text{Bin}_i$, $v_{u.dest} \in \text{Interval}_i$). The complexity of our partitioning approach is $O(E)$. During the processing, the data of each shard (i.e., edges) remain fixed;[1] the data of each bin (i.e., updates) are recomputed in each scatter phase; the data of each interval (i.e., vertices) are updated in each gather phase.

Fig. 4 shows the data layout after the graph in Fig. 1 is partitioned into 2 partitions (i.e., $k = 2$) with each partition having 3 vertices (i.e., $m = 3$). Note that the size of each shard depends on the number of edges whose *source* vertices are in the corresponding interval; the size of each bin depends on the number of edges whose *destination* vertices are in the corresponding interval.

Algorithm 2 illustrates the computation of edge-centric paradigm after the input graph is partitioned. 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 into the on-chip RAMs (Lines 4 and 12). Then, edges (updates) are sequentially read from the external memory during the scatter (gather) phase (Lines 5 and 13). Due to the vertex buffering, the processing engines on the FPGA can access the vertex data directly from the on-chip RAMs to process edges and updates (Lines 6 and 14).

1. We assume the edges of the input graph do not change.

4.2 Partition Skipping

One key issue of the edge-centric paradigm is that it requires to traverse all the edges of the graph in the scatter phase of each iteration. This can result in lots of redundant edge traversals for non-stationary graph algorithms, in which traversing the edges of non-active vertices in an iteration is unnecessary. In order to address this issue, we propose a *partition skipping scheme* to reduce redundant edge traversals for non-stationary graph algorithms. We define *active partition* (in an iteration) as a partition that has at least one active vertex in its interval. For each partition, we maintain a 1-bit `status` flag to indicate if the partition is active or not. In the scatter phase, we check the `status` flag of each partition. If a partition is active, the edges in its shard are traversed; otherwise, this partition is directly skipped. In the gather phase, when the attribute of a vertex is updated, the corresponding partition that this vertex belongs to will be marked as active for the next iteration.

Algorithm 2. Edge-Centric Graph Processing based on Graph Partitioning

```
1:  while not done do
2:    Scatter phase:
3:    for $i$ from 0 to $k-1$ do
4:      Store $\text{Interval}_i$ in on-chip RAMs //vertex buffering
5:      for each edge $e \in \text{Shard}_i$ do
6:        $u \leftarrow \text{Process\_edge}(e, v_{e.src})$
7:        Write $u$ into $\text{Bin}_{\lfloor u.dest/m \rfloor}$
8:      end for
9:    end for
10:   Gather phase:
11:   for $i$ from 0 to $k-1$ do
12:     Store $\text{Interval}_i$ in on-chip RAMs //vertex buffering
13:     for each update $u \in \text{Bin}_i$ do
14:       $v_{u.dest} \leftarrow \text{Apply\_update}(u)$
15:     end for
16:     Write $\text{Interval}_i$ into external memory
17:   end for
18: end while
```

4.3 Parallelizing Edge-Centric Graph Processing

To fully utilize the massive parallelism offered by the FPGA, we parallelize the execution of Algorithm 2 using two levels of parallelism, including *inter-partition* and *intra-partition* parallelism.

4.3.1 Inter-Partition Parallelism

Since HitGraph employs p $(p \geq 1)$ PEs on the FPGA, up to p partitions can be processed by the PEs in parallel. We define the parallelism to process distinct partitions by distinct PEs in parallel as *inter-partition parallelism* and denote it as p. We use a centralized load balancing scheme [45] to allocate the computation tasks of partitions to the PEs. In each phase (i.e., scatter phase or gather phase), the computation tasks of all the partitions are maintained in a task pool. When a PE completes the computation of a partition, it is dynamically assigned another partition that has not been processed. When all the tasks in the task pool have been completed, the algorithm steps to the next phase. The parallelized algorithm for edge-centric graph processing is illustrated in Algorithm 3.

4.3.2 *Intra-Partition Parallelism*

Inside each processing engine, we employ q ($q \geq 1$) parallel processing pipelines (see Section 5.2). In the scatter (gather) phase, these q processing pipelines concurrently process q distinct edges (updates) of the same shard (bin) in a pipelined fashion. We define the parallelism to concurrently process distinct edges or updates inside each PE as *intra-partition parallelism* and denote it as q. Since there are p PEs on the FPGA, the total number of processing pipelines is $p \times q$. Hence, up to $p \times q$ edges or updates can be concurrently processed by the accelerator in each clock cycle.

Algorithm 3. Parallel Edge-Centric Graph Processing

```
Let Partition_i denote the ith partition, $0 \leq i < k$
Let PE_j denote the jth PE on the FPGA, $0 \leq j < p$
 1: while not done do
 2:   Scatter phase:
 3:   for each Partition_i parallel do
 4:     if Partition_i is active then
 5:       if $\exists$ PE_j: PE_j is idle then
 6:         PE_j $\leftarrow$ Scatter phase of Partition_i
 7:       else
 8:         Wait until some PE becomes idle
 9:       end if
10:     end if
11:   end for
12:   Barrier
13:   Gather phase:
14:   for each Partition_i parallel do
15:     if Bin_i is non-empty then
16:       if $\exists$ PE_j: PE_j is idle then
17:         PE_j $\leftarrow$ Gather phase of Partition_i
18:       else
19:         Wait until some PE becomes idle
20:       end if
21:     end if
22:   end for
23:   Barrier
24: end while
```

4.4 Data Layout Optimization

Let $r_0, r_1, \ldots, r_{h-1}, r_h, r_{h+1}, \ldots,$ denote a sequence of external memory accesses. We define a memory access r_h as a *sequential memory access* if the memory location accessed by r_h is contiguous to the memory location accessed by r_{h-1}; otherwise, r_h is defined as a *non-sequential* memory access. Non-sequential memory accesses can result in additional access latency as well as additional memory power consumption [28], [43]. Therefore, it is desirable to optimize the data layout to reduce the number of non-sequential memory accesses.

In Algorithm 2, reading vertices (Lines 4 and 12), edges (Line 5), and updates (Line 13) from external memory and writing vertices (Line 16) into external memory all result in sequential memory accesses. However, writing updates into external memory (Line 7) results in non-sequential memory accesses. This is because the produced updates need to be written into the bins based on their destination vertices. It is likely that the destination vertices of consecutively produced updates are in distinct intervals. In this scenario, these updates are written into distinct bins stored in discontinuous external memory locations, thus resulting in non-sequential memory accesses. In the worst case, each update can result in a non-sequential external memory access. Therefore, assuming $|S|$ is the number of edges in a shard, processing all the edges in the shard can result in $O(|S|)$ non-sequential external memory accesses in the scatter phase. In order to minimize the number of non-sequential external memory accesses, we propose an *optimized data layout* that is achieved by sorting the edges in each shard based on their destination vertices. Because the average number of edges in each shard is $\frac{E}{k}$, the average-case complexity to generate our proposed data layout is $O(E \log \frac{E}{k})$.

We demonstrate that the optimized data layout significantly reduces the non-sequential external memory accesses.

Theorem 1. *In the scatter phase, based on our optimized data layout, processing each shard results in $O(k)$ non-sequential external memory accesses, where k is the total number of partitions.*

Proof. The destination vertices of the updates are the same as the destination vertices of the traversed edges (see Table 1). Since we have sorted each shard based on the destination vertices, the updates are produced in a sorted order as well. Therefore, the updates whose destination vertices belong to the same interval are produced consecutively and thus are written into the same bin. Non-sequential memory access only occurs when an update belonging to a different bin (i.e., other than the bin that the previous update is written into) is produced. Therefore, writing the updates produced by traversing one shard results in $O(k)$ non-sequential external memory accesses, which is far less than $O(|S|)$ (i.e., $|S| > 1000k$). □

4.5 Data Communication Reduction

Since traversing each edge will produce an update, the total number of updates produced in the scatter phase is equal to the number of edges (i.e., $|E|$). Therefore, $|E|$ updates are written into the external memory in the scatter phase, and read from the external memory in the following gather phase. This results in $|E|$ updates transferred back and forth between the FPGA and external memory in each iteration. In order to reduce such data communication, we propose two optimizations: update combining and update filtering.

4.5.1 *Update Combining*

In the scatter phase, we propose to combine the updates that have the same destination vertex before writing them into the external memory. For example, for PR, combining multiple updates can be performed by summing them up. Note that the update combining scheme is enabled by our data layout optimization discussed in Section 4.4. Since the proposed data layout sorts each shard based on the destination vertices, in the scatter phase, the updates that have the same destination vertex are produced consecutively. These consecutive updates that have the same destination vertex can be easily combined as one update. As a result, the number of updates to be written into the external memory in the scatter phase is reduced. Consequently, the number of updates to be processed in the following gather phase is reduced as well.

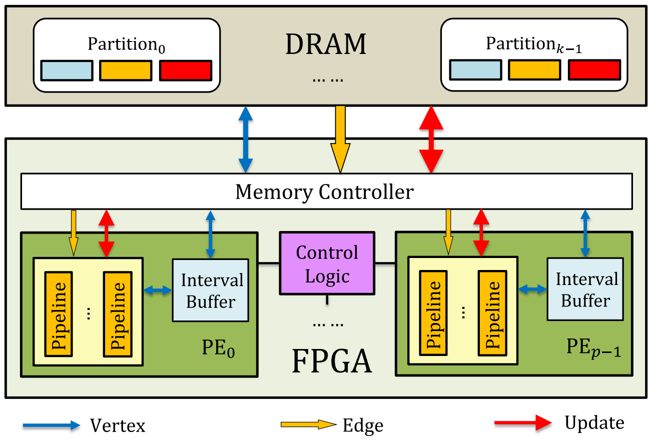


Fig. 5. Overall architecture.

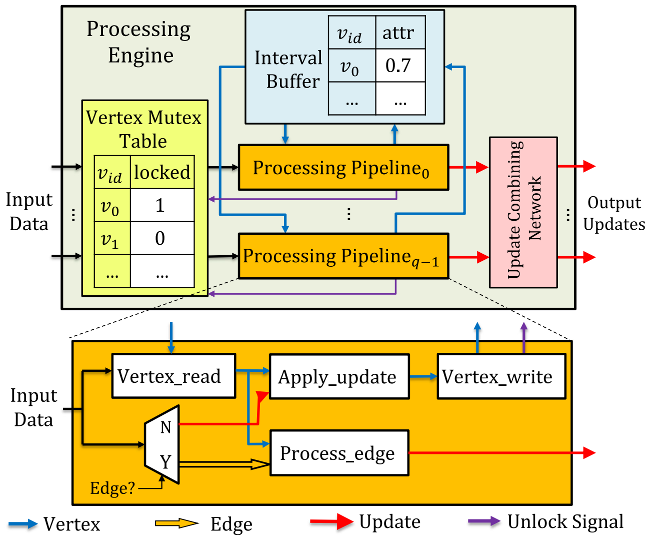


Fig. 6. Architecture of processing engine.

4.5.2 *Update Filtering*

We further propose an update filtering scheme for non-stationary graph algorithms. We assign each vertex an additional `active_tag` to indicate whether the vertex is active or not in the current iteration. In the scatter phase, for each produced update, we check the `active_tag` of the source vertex that the update is produced based on. If an update is produced based on an active vertex, it is marked as a valid update; otherwise, it is invalid. All the invalid updates are discarded and will not be written into the external memory, thus reducing the data communication. Note that the update filtering optimization is not applicable to stationary graph algorithms in which all the vertices are active in each iteration.

5 Implementation Detail

5.1 Overall Architecture

We show the overall architecture of our design in Fig. 5. The DRAM connected to the FPGA is the external memory to store all the intervals, shards, and bins. There are p processing engines on the FPGA, which process p distinct partitions in parallel. Each PE has an individual interval buffer and multiple processing pipelines. The interval buffer is constructed by on-chip UltraRAMs and used to buffer the interval data of the partition being processed by the PE. The processing pipelines of each PE concurrently process distinct edges (updates) of the same shard (bin) in a pipelined fashion in the scatter (gather) phase. The control logic is responsible for scheduling the execution and allocating the computation of partitions to the PEs. The memory controller handles the external memory accesses made by the PEs. 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

Fig. 6 depicts the architecture of each PE. As shown, each PE employs q processing pipelines ($q \geq 1$), thus is able to concurrently process q input data in each clock cycle. The update combining network is used to perform the update filtering and update combining optimizations (Section 4.5) in the scatter phase. The vertex mutex table is used to prevent read-after-write data hazard due to the data dependencies in the gather phase.

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 module reads the attribute of the source vertex of the edge from the interval buffer. The process-edge module produces an update based on the edge weight and the attribute of the source vertex. Each produced update is assigned a `validity` flag to indicate whether it is produced based on an active vertex or not. Note that the vertex-write module and vertex mutex table are not used during the scatter phase; this is because there are only read accesses to the vertices in the scatter phase. All the updates produced by the q processing pipelines are fed into the update combining network, which employs parallel Sort-and-Combine (SaC) units to combine the input updates based on their destination vertices. Each SaC unit takes two updates as input and compares their destination vertices. If the two updates are both valid and have the same destination vertex, they are combined and output as one valid update; otherwise, they are sorted based on their destination vertices and output to the next pipeline stage. The update combining network arranges the SaC units in a pipelined bitonic sorter fashion [30]. Therefore, it sustains a throughput of combining q updates per clock cycle. For an efficient implementation of the combining network, we set the value of q as a power of 2, resulting in $(1+\log q) \cdot \log q \cdot q/4$ SaC units in total. Fig. 7 depicts the architecture of the update combining network for $q = 4$. Note that the invalid updates output by the update combining network are discarded and will not be written into the external memory.

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 module reads the attribute of the destination vertex of the update from the interval buffer. The apply-update module computes the updated attribute of the destination vertex. At last, the vertex-write module writes the updated attribute of the destination vertex into the interval buffer. Since there are both read and write accesses to the vertex attributes in the gather phase, Read-After-Write (RAW) data hazard (i.e., the vertex-read

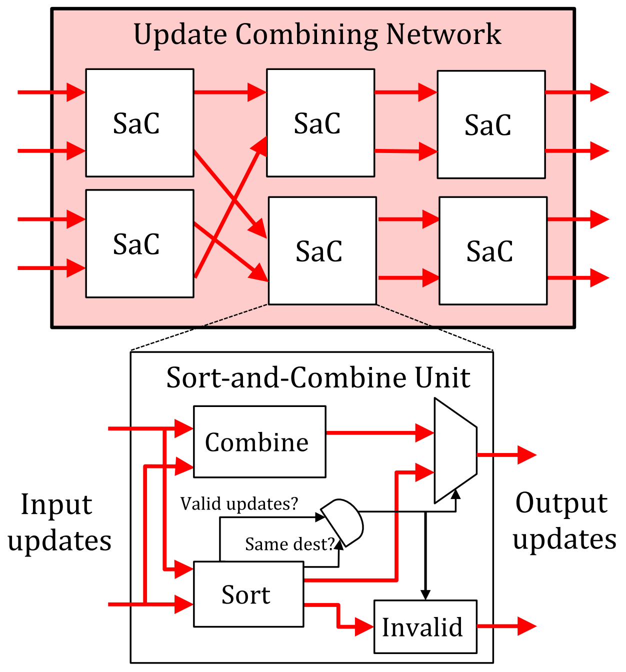


Fig. 7. Update combining network for $q = 4$.

module reads the attribute of a vertex that is being computed) may occur. In order to handle the possible RAW data hazard, we develop a Vertex Mutex Table (VMT) based on a fine-grained locking mechanism [45]. The VMT uses Block RAMs (BRAMs) to maintain a 1-bit lock for each vertex of the partition being processed. A lock with value 1 means that 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 VMT checks the lock status of its destination vertex: if the lock value is 0 (i.e., unlocked), the update is fed into the processing pipeline and the lock value is set to 1 (i.e., locked); otherwise, the pipeline stalls until the lock value becomes 0. Note that when any processing pipeline writes an updated vertex attribute into the interval buffer, it also generates an unlock signal (see Fig. 6) to the VMT to unlock the corresponding vertex. Therefore, deadlock will not occur. For the graph algorithms whose `Apply_update` function can be performed within a single clock cycle (e.g., SSSP and WCC), we propose to replace the VMT with data forwarding circuits to avoid the pipeline stalls due to data hazards. As shown in Fig. 8, the data forwarding circuits forward the vertex attribute that is output by each processing pipeline to all the processing pipelines. Each apply-update module checks whether the attribute of the destination vertex of its input update is among the forwarded data; if it is, the apply-update module

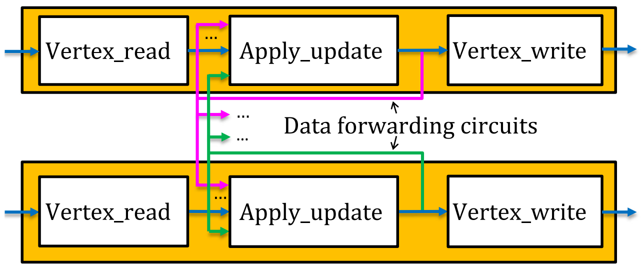


Fig. 8. Data forwarding circuits.

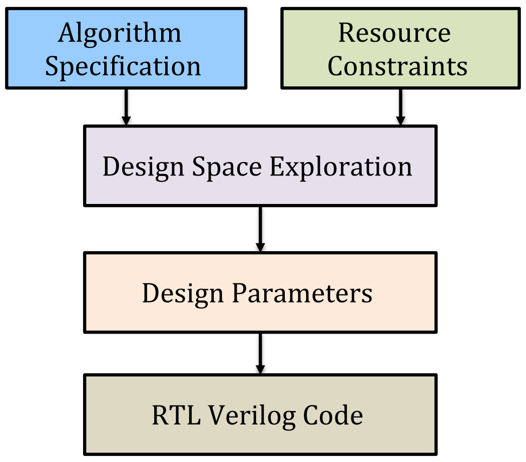


Fig. 9. Workflow of the design automation tool.

uses the forwarded data rather than the data read from the interval buffer. To support q pipelines in a PE, q^2 data forwarding circuits are needed.

6 Design Automation Tool

6.1 Tool Workflow

We have built a design automation tool to allow users to rapidly generate the FPGA accelerators based on our design methodology. Fig. 9 illustrates the workflow of our design automation tool.

As shown, users need to provide the edge-centric algorithm specification and hardware resource constraints to the tool. For example, users can specify the data type and data width of each edge, vertex, and update, and hardware resource constraints such as the available on-chip RAMs, logic resources, DSP resources, and the external memory bandwidth for implementing the target accelerator. Our tool uses these constraint inputs to determine the design parameters of the accelerator, including inter-partition parallelism (p), intra-partition parallelism (q), and the capacity of each interval buffer in terms of vertices (m). Users can also manually choose these design parameters. Based on the selected design parameters, the tool generates all the design modules (i.e., vertex mutex table, update combining network, processing engines, etc.) and automatically connects them to produce the Register Transfer Level (RTL) Verilog code of the FPGA accelerator.

6.2 Parameter Selection

The selection of p, q, and m is through design space exploration as shown in Algorithm 4 with the assumption that the resulting accelerator operates at 200 MHz. Because our framework traverses edges in a streaming fashion, DRAM bandwidth is the main performance constraint. Our tool selects p and q which can maximally utilize the available DRAM bandwidth and maintain low architecture complexity as well. Our tool first sets p to the number of DRAM channels. This is because a larger value of p leads to the scenario in which multiple PEs share the same DRAM channel and currently update the vertices belonging to the same partition, resulting in data hazards; a smaller value of p requires a larger q in order to increase DRAM bandwidth utilization, quadratically increasing the architecture complexity of each

TABLE 2
Graph Datasets

Dataset	$\vert V\vert$	$\vert E\vert$	Diam.	Graph type
BKstan	0.7 M	7.6 M	514	Web graph
WKtalk	2.4 M	5.0 M	9	Communication
CAroad	2.0 M	5.5 M	849	Road network
LJounal	4.8 M	69.0 M	16	Social network
Twitter	41.6 M	1468.4 M	15	Social network
RMat21	2.1 M	182.1 M	6	Synthetic graph
RMat24	16.8 M	263.0 M	6	Synthetic graph

PE (see Section 5.2). Then, our tool selects the maximum feasible value of q that satisfies the hardware resource requirement. After p and q are determined, our tool selects m based on the available on-chip RAM resources. Note that m is constrained by on-chip RAMs and our tool selects the largest feasible m. This is because a larger value of m results in fewer partitions (i.e., k), and thus fewer non-sequential external memory accesses (see Section 4.4).

Algorithm 4. Design Space Exploration

```
 1: Inputs: hardware resource constraints
 2: $p \leftarrow$ number of DRAM channels
 3: $q \leftarrow 1$
 4: **while** true **do**
 5:     **if** any resource is insufficient **then**
 6:         $q \leftarrow q/2$
 7:         Break
 8:     **else**
 9:         $q \leftarrow q \times 2$
10:     **end if**
11: **end while**
12: $m \leftarrow m_{max}$ s.t. $m_{max}$ satisfies on-chip RAM constraint
13: Outputs: $p, q, m$
```

7 Performance Evaluation

7.1 Experimental Setup

We conduct experiments using the Xilinx Virtex UltraScale+ xcvu5pflva2104 FPGA. The target FPGA device has 600,577 slice LUTs, 1,201,154 slice registers, 3,474 DSPs, 36 Mb of BRAMs, and 132 Mb of UltraRAMs. We synthesize, place-and-route, and simulate our designs using Xilinx Vivado Design Suite 2018.1 [31]. We use four Micron 8 GB DDR3-1600 MT41K1G8 chips as the external memory. Each DRAM chip runs at 800 MHz and has a peak data transfer rate of 15 GB/s. Therefore, the peak external memory bandwidth is 60 GB/s. We evaluate the DRAM performance using DRAMSim2 [36], a widely used tool to evaluate DRAM performance for the target platform [23], [37]. A broad range of graph datasets, including both real-life and synthetic graphs, are used in the experiments. Table 2 summaries the key characteristics of these datasets. The real-life graphs are obtained from Stanford network dataset repository [38] and the synthetic graphs are generated using the Graph500 graph generator [1], respectively. For SpMV, standard addition and multiplication operators are used; for PR, we set the value of damping factor (i.e., d in Equation (1)) to 0.85; for SSSP, we randomly choose the source vertex for 20 runs and report the average performance in the following sections.

7.2 Performance Metrics

We evaluate our designs using the following performance metrics:

- Resource utilization: the utilization of FPGA resources, including logic slices, registers, on-chip RAMs, and DSPs
- Clock rate: the clock rate sustained by the FPGA accelerator
- Power consumption: the total power consumed by the FPGA accelerator, including both the dynamic power and static power
- Execution time: for stationary algorithms (i.e, SpMV and PR), the execution time refers to the average execution time per iteration; for non-stationary algorithms (i.e, SSSP and WCC), the execution time refers to the total execution time of the algorithm
- Throughput: the number of Traversed Edges Per Second (TEPS), computed as the total number of traversed edges divided by the execution time

7.3 Clock Rate, Resource Utilization, and Power Consumption

We generate various FPGA designs with different architecture parameters using our design automation tool. Table 3 shows the resource utilization, clock rate, and power consumption of the generated FPGA accelerators. All the reported results are post-place-and-route results evaluated using Xilinx Vivado Design Suite 2018.1 [31]. Each interval buffer is able to store the data of 256K vertices (i.e., $m = 256$ K). We observe that the clock rate slightly deteriorates as p and q increase. The deterioration is due to the increasing routing complexity of the accelerator as more hardware resources are consumed. Based on Algorithm 4 and the available resources of the target FPGA device, we empirically set the number of PEs to 4 ($p = 4$), the number of pipelines in each PE to 8 ($q = 8$), and the capacity of each interval buffer to 256K vertices ($m = 256$ K) for the experiments in the rest of the paper.

7.4 Execution Time and Throughput

We show the execution time and throughput performance for various graph datasets in Table 4. On average, our FPGA accelerators achieve a high throughput of 2,076 MTEPS for SpMV, 2,225 MTEPS for PR, 2,916 MTEPS for SSSP, and 3,493 MTEPS for WCC, respectively. We also observe that the achieved throughput for the dataset WKtalk is much less than the average for all the four graph algorithms. This is because approximately 90 percent of the edges are grouped into the same shard after the graph is partitioned. As a result, the distribution of the computation load among the PEs is extremely unbalanced in the scatter phase (i.e., one PE traverses 90 percent of the edges while the other PEs traverse only 10 percent of the edges). Previous studies [46], [47] have shown that reordering and relabeling vertices of such graphs can improve load balancing. For the other datasets, we do not observe any load balancing issue; the computation load is equally distributed among the PEs and each PE sustains an average throughput of 563 MTEPS for SpMV, 602 MTEPS for PR, 760 MTEPS for SSSP, and 949 MTEPS for WCC, respectively.

TABLE 3
Resource Utilization, Clock Rate, and Power Consumption

Algorithm	p	q	LUT (%)	Reg (%)	DSP (%)	On-chip RAM (%)		Clock rate (MHz)	Power (Watt)
						Block RAM	UltraRAM		
SpMV	1	1	8.5	4.6	0.1	1.7	13.6	212	4.0
		2	9.3	4.7	0.1	1.9	13.6	211	4.2
		4	10.9	5.2	0.2	2.1	13.6	209	5.2
		8	15.9	6.6	0.5	2.6	13.6	208	6.1
	2	1	17.2	9.1	0.1	3.4	27.2	212	6.3
		2	18.9	9.4	0.2	3.7	27.2	211	6.5
		4	21.8	10.3	0.5	4.2	27.2	208	7.7
		8	31.7	13.2	0.9	5.3	27.2	206	9.5
	4	1	34.5	18.1	0.2	6.8	54.5	207	10.0
		2	37.8	18.7	0.5	7.4	54.5	207	10.2
		4	43.7	20.6	0.9	8.4	54.5	207	13.1
		8	63.5	26.3	1.8	10.6	54.5	201	17.5
PR	1	1	8.7	4.5	0.1	1.7	13.6	217	3.1
		2	9.7	4.7	0.1	1.8	13.6	217	3.3
		4	11.4	5.2	0.2	2.0	13.6	215	3.6
		8	16.9	6.6	0.5	2.3	13.6	208	4.1
	2	1	17.5	9.1	0.1	3.3	27.2	209	4.5
		2	19.5	9.3	0.2	3.5	27.2	209	4.6
		4	22.8	10.3	0.5	3.9	27.2	208	5.0
		8	34.0	13.1	0.9	4.6	27.2	204	5.8
	4	1	34.6	18.1	0.2	6.6	54.5	208	6.9
		2	39.0	18.6	0.5	7.0	54.5	208	7.1
		4	45.6	20.5	0.9	7.8	54.5	202	7.6
		8	68.1	26.1	1.8	9.2	54.5	200	10.7
SSSP	1	1	5.1	2.7	0	0.1	13.6	222	2.8
		2	5.2	2.7	0	0.2	13.6	216	2.9
		4	6.5	3.2	0	0.4	13.6	215	3.0
		8	8.0	3.5	0	0.8	13.6	208	3.1
	2	1	10.0	5.3	0	0.2	27.2	220	3.1
		2	10.3	5.4	0	0.4	27.2	212	3.1
		4	12.1	6.1	0	0.8	27.2	207	3.7
		8	16.1	7.0	0	1.5	27.2	206	5.0
	4	1	18.2	10.5	0	0.4	54.5	210	5.0
		2	20.2	10.7	0	0.8	54.5	207	5.2
		4	24.3	12.1	0	1.6	54.5	205	6.2
		8	32.3	13.9	0	2.9	54.5	200	8.2
WCC	1	1	5.7	2.9	0	0.1	13.6	223	2.5
		2	5.7	3.0	0	0.2	13.6	221	2.6
		4	6.7	3.2	0	0.4	13.6	219	2.9
		8	8.7	3.8	0	0.8	13.6	213	3.2
	2	1	11.4	5.6	0	0.2	27.2	216	3.5
		2	11.6	5.9	0	0.4	27.2	212	3.6
		4	13.5	6.3	0	0.8	27.2	208	3.9
		8	19.3	7.6	0	1.5	27.2	205	4.4
	4	1	23.6	11.6	0	0.4	54.5	210	5.0
		2	23.9	11.8	0	0.8	54.5	207	5.4
		4	27.0	12.6	0	1.6	54.5	203	5.9
		8	34.5	15.1	0	2.9	54.5	200	7.5

7.5 Impact of the Optimizations

To show the effectiveness of the proposed optimizations in Section 4, we compare the optimized designs with various non-optimized FPGA-based baseline designs.

7.5.1 Impact of Partition Skipping

We first study the impact of the partition skipping optimization for SSSP and WCC. The baseline design does not have this optimization and thus traverses the edges of both active partitions and non-active partitions in each iteration. Fig. 10 shows the reduction of edge traversals due to the partition skipping optimization. On average, this optimization reduces the number of edge traversals by 1.4× for SSSP and 1.3× for WCC, respectively. We also observe that the partition skipping optimization is very effective when the ratio of active vertices (i.e., the number of active vertices over the total number of vertices) in an iteration is very low (e.g., the first iteration of SSSP and the last iteration of WCC); in such

TABLE 4
Execution Time and Throughput

Algorithm	Metrics	Dataset							
		BKstan	WKtalk	CAroad	LJounal	Twitter	RMat21	RMat24	Average
SpMV	Execution time (ms)	3.2	5.0	2.8	36.2	652.5	56.7	143.5	–
	Throughput (MTEPS)	2,361	1,004	1,964	1,906	2,250	3,217	1,832	2,076
PR	Execution time (ms)	3.0	4.5	2.7	32.7	590.4	53.4	140.3	–
	Throughput (MTEPS)	2,533	1,116	2,037	2,110	2,487	3,410	1,875	2,225
SSSP	Execution time (ms)	782.4	25.5	1113.3	592.1	5576.8	967.1	921.3	–
	Throughput (MTEPS)	3,109	2,156	2,441	3,111	2,869	4,304	2,419	2,916
WCC	Execution time (ms)	1769.0	46.2	1480.1	412.9	6617.1	450.3	1107.9	–
	Throughput (MTEPS)	4,949	1,665	3,652	3,322	3,395	4,852	2,619	3,493

iterations, many partitions do not have any active vertices and thus can be skipped. However, when the ratio of active vertices in an iteration is very high (e.g., the first iteration of WCC), it is highly likely that all the partitions are active; in this scenario, none of the partitions can be skipped.

7.5.2 Impact of Update Combining and Filtering

We further explore the effectiveness of the update combining and filtering optimization (Section 4.5) to reduce the data communication between FPGA and external memory. For comparison purpose, we implement a baseline design which has the partition skipping optimization and uses the optimized data layout, but does not have the update combining or filtering optimization. Fig. 11 illustrates the effectiveness of the optimization. We compute the reduction factor of updates as the total number of updates written into DRAM for the basedline design divided by the total number of updates written into DRAM for the optimized design. Therefore, a higher reduction factor corresponds to better performance. We observe that the total number of updates written into the DRAM is reduced by 2.3× to 12.5× for SpMV, 2.7× to 14.7× for PR, 10.6× to 548.2× for SSSP, and 6.9× to 1253.1× for WCC, respectively. This optimization has higher impact on non-stationary graph algorithms (i.e., SSSP and WCC) because non-stationary graph algorithms employ both the update combining and update filtering schemes, while stationary graph algorithms (i.e, SpMV and PR) only employ the update combining scheme. On average, this optimization reduces the number of updates written into DRAM by 6.5× for SpMV, 7.5× for PR,

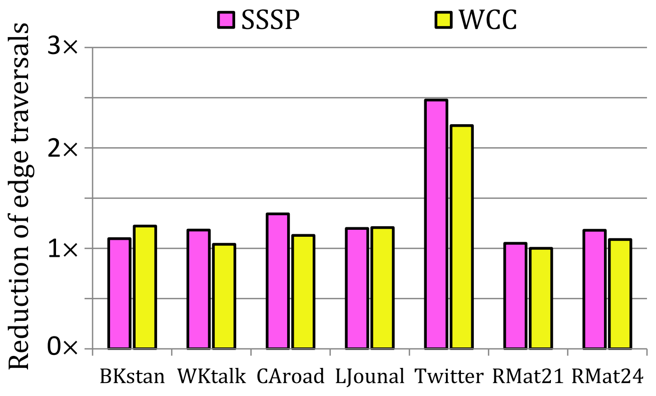


Fig. 10. Reduction of edge traversals due to partition skipping.

104.1× for SSSP, and 218.9× for WCC, respectively. Because of this optimization, the execution time is reduced by 1.9× to 4.7× for SpMV, 2.0× to 5.0× for PR, 2.8× to 11.4× for SSSP, and 1.4× to 4.9× for WCC, respectively.

7.5.3 Impact of Data Layout Optimization

Lastly, we study the impact of our data layout optimization (Section 4.4). The baseline design for the comparison employs the partition skipping and communication reduction optimizations, but uses the data layout described in Section 2.2 which does not have our data layout optimization. Fig. 12 shows the reduction factor of non-sequential DRAM accesses, which is computed as the number of non-sequential DRAM accesses performed by the baseline design divided by the number of non-sequential DRAM accesses performed by the optimized design. We observe that this optimization reduces the number of non-sequential DRAM accesses by 2.1× to 12.2× for SpMV, 2.4× to 15.3× for PR, 2.4× to 8.2× for SSSP, and 2.2× to 12.2× for WCC, respectively. As a result, the optimized designs can sustain a high DRAM bandwidth of 36.7 GB/s to 46.8 GB/s, while the baseline designs can only sustain 10.3 GB/s to 28.5 GB/s.

7.6 Comparison with State-of-the-Art

7.6.1 Comparison with Multi-Core Designs

We first compare the performance of our design with several highly-optimized multi-core implementations, including X-Stream [6], NXgraph [9], GraphX [8], and GraphMat [11]. X-Stream [6] runs on a 32-core AMD Opteron 6,272 processor with 25 GB/s DRAM bandwidth. NXgraph [9] runs on a hexa-core Intel i7 processor with 160 GB/s DRAM bandwidth. GraphX [8] runs on a cluster consisting of 16 computing nodes; each node has 8 cores. GraphMat [11] runs on a 24-core Intel Xeon E5-2697 processor with 80 GB/s DRAM bandwidth. Since these works do not report the throughput performance, we conduct the comparison based on the execution time performance. Table 5 summarizes the results of the comparison using the same datasets. Our FPGA designs achieve up to 20.5×, 35.5×, 5.0× and 37.9× speedup for SpMV, PR, SSSP, and WCC, respectively. In addition, the power consumption of our FPGA designs (<20 Watt) are much lower than the multi-core platforms (typically >80 Watt). Hence, from energy-efficiency perspective (i.e., performance-per-watt), our framework achieves even larger improvement.

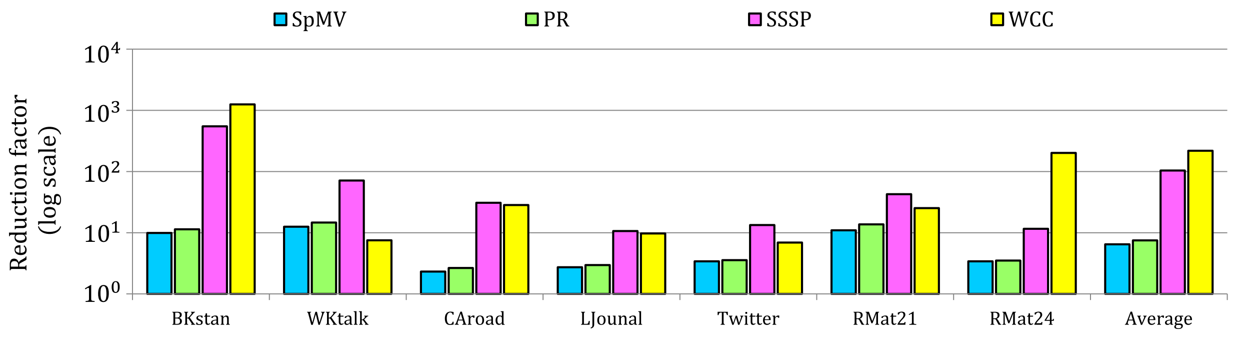


Fig. 11. Reduction factor of updates written into DRAM due to update combining and filtering.

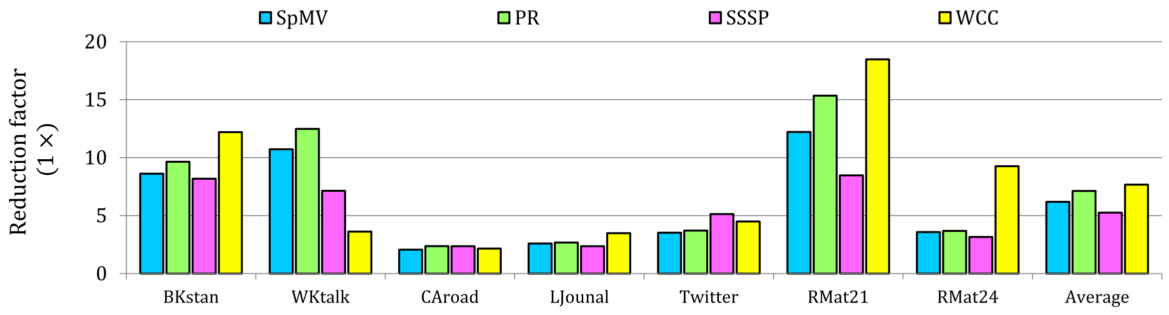


Fig. 12. Reduction factor of non-sequential DRAM accesses due to data layout optimization.

7.6.2 Comparison with GPU Designs

We further compare our FPGA framework with three state-of-the-art GPU-based graph processing frameworks, including nvGRAPH [13], CuSha [42], and Gunrock [12]. The results of the comparison are shown in Table 6. It can be observed that our FPGA-based designs achieve comparative performance with the GPU-based designs. Note that the external memory bandwidth of the GPU platforms (288 GB/s) is 4.8× higher than our target platform (60 GB/s). If we scale these GPU results by assuming a peak bandwidth of only 60 GB/s, HitGraph will outperform the GPU designs by 2.2× to 7.2×. In addition, the thermal design power of the GPU platforms is over 20× higher than the power consumption of our FPGA-based accelerators.

TABLE 5
Comparison with Multi-Core Implementations

Algorithm	Dataset	Approach	Exec. time (ms)	Speedup
SpMV	LJournal	[6]	740	20.5×
		This paper	36	
PR	LJournal	[6]	580	1.0×
		[9]	100	5.8×
		[11]	45	12.9×
		This paper	33	17.6×
	Twitter	[8]	20950	1.0×
		[9]	2050	10.2×
		[11]	1800	11.6×
		This paper	590	35.5×
SSSP	CAroad	[11]	5500	5.0×
		This paper	1113	
	RMat24	[11]	1900	2.1×
		This paper	921	
WCC	LJournal	[6]	7220	17.5×
		This paper	413	
	Twitter	[8]	251000	37.9×
		This paper	6617	

7.6.3 Comparison with FPGA Designs

Lastly, we compare our proposed framework with two state-of-the-art FPGA frameworks for accelerating general graph algorithms, including GraphOps [25] and ForeGraph [23]. GraphOps [25] is a hardware library to construct FPGA-based accelerators for graph analytics. Its target platform is a CPU-FPGA heterogeneous platform consisting of a 12-core Intel Xeon X5650 host processor and a Xilinx Virtex-6 FPGA. The host processor operates at 2.66 GHz and has 192 Mb cache; its peak DRAM bandwidth is 64 GB/s. The FPGA has 475k logic cells and 37 Mb of BRAMs; its peak DRAM bandwidth is 38.4 GB/s. Table 7 summarizes the results of the comparison with GraphOps, showing that our framework improves the throughput performance by up to 27.6× and 50.7× for SpMV and PR, respectively.

ForeGraph [23] is a graph processing framework based on four Virtex UltraScale FPGAs that are interconnected in the Microsoft Catapult fashion [27]. Each FPGA device has 1,074K LUTs, 2,148K registers, 133 Mb of BRAMs, and peak DRAM bandwidth of 19.2 GB/s. Table 8 shows the results of the comparison with ForeGraph. Our framework achieves 1.3× and 2.0× higher throughput for PR and WCC, respectively. Note that ForeGraph uses four FPGAs while our framework uses a single FPGA.

8 Related Work

8.1 Software Graph Processing Frameworks

Many software-based graph processing frameworks have been developed, such as GraphChi [5], X-Stream [6], NXgraph [9], and GraphMat [11] on multi-core, and CuSha

TABLE 6
Comparison with GPU-based Implementations

Algor.	Dataset	Approach	Platform	Bandwidth (GB/s)	# of cores/ pipelines	Frequency (MHz)	Power (Watt)	Exec. Time (ms)
PR	RMat21	[12]	NIVIDIA Tesla K40c	288	2,880	745	245.0	80.4
		This paper	Xilinx UltraScale+	60	32	200	10.7	53.4
	Twitter	[13]	NIVIDIA Tesla M40	288	3,072	1,140	250.0	850.0
		This paper	Xilinx UltraScale+	60	32	200	10.7	590.1
SSSP	LJournal	[42]	NIVIDIA GeForce GTX780	288	2,304	863	250.0	346.0
		This paper	Xilinx UltraScale+	60	32	200	8.2	592.1
WCC	RMat21	[12]	NIVIDIA Tesla K40c	288	2,880	745	245.0	428.9
		This paper	Xilinx UltraScale+	60	32	200	7.5	450.3
	LJournal	[42]	NIVIDIA GeForce GTX780	288	2,304	863	250.0	190.0
		This paper	Xilinx UltraScale+	60	32	200	7.5	412.9

[42], Gunrock [12], nvGRAPH [13], Medusa [50], and Graphie [44] 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 [5] is the first graph processing framework developed based on a single multi-core processor. It stores all the graph data in solid-state drive (SSD) and develops a parallel sliding-window method to reduce the amount of random accesses to the SSD. X-Stream [6] is designed based on the edge-centric paradigm. It proposes a streaming partition approach to maximize the sequential accesses to the graph data stored in disk. GraphMat [11] maps vertex-centric computations to high-performance sparse matrix operations. NXgraph [9] develops a fine-grained partitioning approach to break graphs into 1D-partitioned vertex blocks and 2D-partitioned edge blocks. The objective is to enable on-the-fly vertex updates and reduce the I/O amount. CuSha [42] focuses on addressing the limitations of uncoalesced global memory accesses for GPU-based graph processing. nvGraph [13] is a graph analytics library developed by NVIDIA based on Compute Unified Device Architecture (CUDA). Medusa [50] develops six programming APIs, which allow developers to define their own data structures and graph kernels. Gunrock [12] proposes a data-centric processing abstraction which leverages GPU to accelerate the frontier computations (i.e., the computations of active vertices). Graphie [44] implements the asynchronous graph-traversal model on a single GPU to reduce the data communication.

TABLE 7
Comparison with GraphOps

Algorithm	Dataset	Approach	Throughput (MTEPS)	Improv.
SpMV	BKstan	[25]	162	14.7×
		This paper	2361	
	WKtalk	[25]	37	27.6×
		This paper	1004	
	RMat24	[25]	165	11.1×
		This paper	1832	
PR	BKstan	[25]	190	13.3×
		This paper	2533	
	WKtalk	[25]	37	29.8×
		This paper	1116	
	RMat24	[25]	37	50.7×
		This paper	1875	

8.2 FPGA-based Graph Processing Accelerators

Using FPGA to accelerate graph processing has demonstrated great success. In [34], [40], [41], Breadth First Search (BFS) is accelerated based 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 [29], G. Lei et al. accelerate the Dijkstra algorithm for SSSP using FPGA. Compared with a CPU implementation running on the AMD Opteron 6,376 processor, the FPGA accelerator achieves up to 5× speedup. In [48], 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. In [26], B. Betkaoui et al. accelerate the all-pairs shortest-paths algorithm using a CPU-FPGA heterogeneous platform; the design achieves 10× speedup over a quad-core CPU implementation and 5× speedup over a AMD Cypress GPU implementation, respectively. However, these FPGA accelerators [29], [34], [35], [40], [41], [48] are algorithm-specific and cannot be used to accelerate other graph algorithms.

GraphGen [20] is a vertex-centric framework to accelerate general graph applications. It partitions the input graph into subgraphs and then processes one subgraph at a time. It also provides a compiler for automatic HDL code generation. However, GraphGen requires both the vertex data and the edge data of each subgraph to fit in the on-chip memory of

TABLE 8
Comparison with ForeGraph

Algorithm	Dataset	Approach	Throughput (MTEPS)	Improv.
PR	Twitter	[23]	1856	1.3×
		This paper	2487	
WCC	Twitter	[23]	1727	2.0×
		This paper	3395	

FPGA. For large input graphs, this requirement can lead to a large number of subgraphs and thus significantly increase the scheduling complexity. GraphOps [25] is an FPGA-based dataflow library for graph processing. It provides several commonly used building blocks for graph processing, such as reading the attributes of all the neighbors of a vertex. The target platform of GraphOps is a CPU-FPGA heterogeneous architecture, in which the FPGA is used to accelerate edge traversals and the CPU is responsible for updating vertex attributes, respectively. However, GraphOps is based on the vertex-centric paradigm and thus suffers random memory accesses to the edges. As a result, GraphOps sustains only 18.3 percent of the external memory bandwidth. ForeGraph [23] is a multi-FPGA-based graph processing framework. It uses the 2-D graph partitioning technique of [10] to partition the input graph into 2-D edge blocks and uses multiple FPGAs to process distinct edge blocks in parallel. However, the performance can be constrained by the communication overhead among the FPGAs.

9 Conclusion and Future Work

In this paper, we presented an FPGA framework to accelerate general graph algorithms based on the edge-centric paradigm. We partitioned the graph to enable efficient on-chip buffering of vertex data and increase the parallelism. We further optimized the data layout to reduce non-sequential external memory accesses and data communication. We also developed a design automation tool to facilitate the generation of the `Verilog` code using our framework. We accelerated four fundamental graph algorithms, including SpMV, PR, SSSP, and WCC, to study the performance of our framework. Experimental results showed that our framework achieved up to 37.9× speedup compared with state-of-the-art multi-core designs, and up to 50.7× throughput improvement compared with state-of-the-art FPGA designs.

The optimizations proposed in this paper are also applicable to multi-core and GPU implementations to improve their performance. First, the graph partitioning approach can be performed based on the cache size of multi-core and GPU platforms to improve the cache performance. Second, the computations can be parallelized by employing parallel thread blocks (i.e., groups of threads) such that distinct thread blocks concurrently process distinct partitions and distinct threads in each thread block process distinct edges or updates in parallel. Third, our optimized data layout can be directly used to improve the memory performance. Fourth, the update combining scheme can be implemented using parallel scan operation on the multi-core and GPU platforms to reduce the data communication.

HitGraph has supported the four algorithms studied in this paper. In the future, we plan to extend it to support more algorithms. Note that to support a new algorithm, only the functions of `Process_edge` and `Apply_update` need to be defined; while all the other design components is applicable to general algorithms. We will maintain the updates of HitGraph at http://www-scf.usc.edu/~shijiezh/HitGraph/. We are also interested in integrating HitGraph with emerging memory technologies such as 3D stacked high-bandwidth memory 2 (HBM2) [51]. This memory technology can provide 256 GB/s bandwidth to enable HitGraph to employ 4× more PEs (i.e., 16 PEs) to saturate the memory bandwidth. We anticipate that HitGraph will achieve 4× higher throughput in this scenario.

Acknowledgments

This work is supported by Intel Strategic Research Alliance funding. This work is also supported in part by the U.S. National Science Foundation under grants ACI-1339756 and CNS-1643351 and Air Force Research Laboratory under grant FA8750-18-2-0034.

References

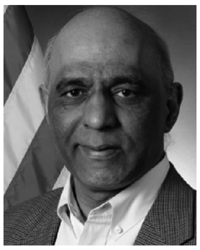
[1] Graph 500, 2017. [Online.] Available: https://graph500.org/
[2] S. Zhou, R. Kannan, H. Zeng, and V. K. Prasanna, "An FPGA framework for edge-centric graph processing," in *Proc. Comput. Frontiers*, 2018, pp. 69–77.
[3] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proc. Int. Conf. Manage. Data*, 2010, pp 135–146.
[4] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, 2015.
[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-Stream: Edge-centric graph processing using streaming partitions," in *Proc. ACM Symp. Operating Syst. Principles*, 2013, pp. 472–488.
[7] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 169–182.
[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 599–613
[9] 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. Int. Conf. Data Eng.*, 2016, pp. 409–420.
[10] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Annu. Tech. Conf.*, 2015, pp. 375–386.
[11] 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," *Proc. VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, 2015.
[12] 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 *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, Art. no. 11.
[13] nvGRAPH, 2019. [Online.] Available: https://developer.nvidia.com/nvgraph
[14] M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo, and M. D. Santambrogio, "Heterogeneous exascale supercomputing: The Role of CAD in the exaFPGA project," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, 2017, pp. 410–415
[15] L. D. Tucci, K. Brien, M. Blott, and M. D. Santambrogio, "Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, 2017, pp. 716–721
[16] GPU vs FPGA performance comparison, 2016. [Online.] Available: http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf
[17] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proc. Int. Symp. Comput. Archit.*, 2016, pp. 166–177.
[18] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 531–543.
[19] T. Huang, G. Dai, Y. Wang, and H. Yang, "HyVE: Hybrid vertex-edge memory hierarchy for energy-efficient graph processing," in *Proc. Des. Autom. Test Eur. Conf. Exhibition*, 2018, pp. 973–978.

[20] 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. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 25–28.
[21] T. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. Int. Symp. Microarchitecture*, 2016, pp. 1–13.
[22] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *Proc. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2016, pp. 103–110.
[23] 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. Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 217–226.
[24] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for FPGA acceleration of large graph problems: Graphlet counting case study," in *Proc. Int. Conf. Field Programmable Technol.*, 2011, pp. 1–8.
[25] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 111–117.
[26] 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. 22nd Int. Conf. Field Programmable Logic Appl.*, 2012, pp. 99–104.
[27] 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.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proc. Int. Symp. Comput. Archit.*, 2014, pp. 13–24.
[28] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing memory performance for FPGA implementation of PageRank," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs*, 2015, pp. 1–6.
[29] G. Lei, Y. Dou, R. Li, and F. Xia, "An FPGA implementation for solving the large single-source-shortest-path problem," *IEEE Trans. Circuits Syst. II: Express Briefs*, vol. 63, no. 5, pp. 473–477, May 2016.
[30] K. E. Batcher, "Sorting networks and their applications," in *Proc. Spring Joint Comput. Conf.*, 1968, vol. 32, pp. 307–314.
[31] Vivado design suite, 2019. [Online.] Available: https://www.xilinx.com/products/design-tools/vivado.html
[32] M. Kim, K. An, H. Park, H. Seo, and J. Kim, "GTS: A fast and scalable graph processing method based on streaming topology to GPUs," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 447–461.
[33] J. Zhou, S. Liu, Q. Guo, X. Zhou, T. Zhi, D. Liu, C. Wang, X. Zhou, Y. Chen, and T. Chen, "TuNao: A high-performance and energy-efficient reconfigurable accelerator for graph processing," in *Proc. Int. Symp. Cluster Cloud Grid Comput.*, 2017, pp. 731–734.
[34] 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. Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 207–216.
[35] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A reconfigurable computing approach for efficient and scalable parallel graph exploration," in *Proc. Int. Conf. Appl.-Specific Syst. Archit. Processors*, 2012, pp. 8–15.
[36] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *Comput. Archit. Lett.*, vol. 10, pp. 16–19, 2011.
[37] Z. Dai, "Application-driven memory system design on FPGAs," PhD thesis, Dept. Electr. Comput. Eng., University of Toronto, Toronto, ON, 2013.
[38] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," 2014. [Online.] Available: https://snap.stanford.edu/data
[39] S. Neuendorffer and K. Vissers, "Streaming systems in FPGAs," in *Proc. Int. Workshop Embedded Comput. Syst.*, 2008, pp. 147–156.
[40] J. Zhang and J. Li, "Degree-aware hybrid graph traversal on FPGA-HMC platform," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2018, pp. 229–238.
[41] 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. Int. Symp. Field-Programmable Gate Arrays*, 2018, pp. 239–248.
[42] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "CuSha: Vertex-centric graph processing on GPUs," in *Proc. Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 239–252.
[43] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
[44] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 2017, pp. 233–245.
[45] A. Grama, G. Karypis, V. Kumar, and A. Gupta, "Introduction to parallel computing," *Pearson*, 2003, p. 656.
[46] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna, "ReCALL: Reordered cache aware locality based graph processing," in *Proc. Int. Conf. High Perform. Comput.*, 2017, pp. 273–282.
[47] V. Balaji and B. Lucia, "When is graph reordering an optimization? Studying the effect of lightweight graph reordering across applications and input graphs," in *Proc. Int. Symp. Workload Characterization*, 2018, pp. 203–214.
[48] 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. Int. Symp. Field-Programmable Custom Comput. Mach.*, 2014, pp. 36–43.
[49] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-FPGA systems," in *Proc. Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 264–273.
[50] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.
[51] Intel Stratix 10 MX FPGAs, 2019. [Online.] Available: https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html

Shijie Zhou received the BS degree in electrical engineering from Zhejiang University, in 2010, and the PhD degree in electrical engineering from the University of Southern California (USC), in 2018. His research interests include parallel computing, graph algorithms, and accelerator design. He received the best paper award at the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). He is a member of the ACM.

Rajgopal Kannan received the BTech degree in computer science and engineering from IIT-Bombay, in 1991 and the PhD degree in computer science from the University of Denver, in 1996. He is currently a computer scientist at the Army Research Lab in the Computing Architectures Branch and a research adjunct professor in electrical engineering at the University of Southern California. He was formerly a professor with the Department of Computer Science, Louisiana State University (2000-2015). His academic research was funded by DARPA, NSF and DOE and he has published more than 150 research papers in international journals and conferences with two patents awarded in the area of network optimization. His research interests are at the intersection of graph analytics, machine learning and edge computing - enabling application acceleration at the edge on low power devices, for example using Software-Defined Memory for memory bound applications. He is also interested in cyber-physical systems, especially data-driven models and analytics driving Smartgrid optimization and control.

Viktor K. Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is Charles Lee Powell chair in engineering in the Ming Hsieh Department of Electrical Engineering and professor of computer science with the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He is the executive director of the USC-Infosys Center for Advanced Software Technologies (CAST) and was an associate director of the USC Chevron Center of Excellence for Research and Academic Training on Interactive Smart Oilfield Technologies (Cisoft). He also serves as the director of the Center for Energy Informatics, USC. He served as the editor-in-chief of the *IEEE Transactions on Computers* during 2003-06. Currently, he is the editor-in-chief of the *Journal of Parallel and Distributed Computing*. He was the founding chair of the *IEEE Computer Society Technical Committee on Parallel Processing*. He is the steering cochair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. His work on regular expression matching received one of the most significant papers in FCCM during its first 20 years award in 2013. He is a fellow of the IEEE, the ACM, and the American Association for Advancement of Science (AAAS).



Guna Seetharaman is the Navy senior scientist (ST) for Advanced Computing Concepts, and the chief scientist for Computation, Center for Computational Science, Navy Research Lab. He also serves as a senior scientific advisor, for the Information Systems and Cyber Technology Directorate, of the Office of Under Secretary of Defense, Research Development and Engineering. He joined NRL, in June 2015, where he leads a team effort on: Video Analytics, High performance computing, lowlatency, high-throughput, on-demand scalable geo-dispersed computer-networks. He worked as Principal engineer at the Air Force Research Laboratory, Information Directorate, 2008-2015, where he led research and development in Video Exploitation, Wide Area Motion Imagery, Computing Architectures and Cyber Security. He holds several US Patents in related areas. His team won the best algorithm award at IEEE CVPR-2014 Video Change Detection challenge, featuring a semantic segmentation of dynamic scene to detect change in the midst of dynamic clutters. He served as a tenured associate professor at the Air Force Institute of Technology, and University of Louisiana at Lafayette, before joining AFRL. He and his colleagues cofounded Team CajubBot and successfully fielded two unmanned vehicles at the DARPA Grand Challenges 2004, 2005 and 2007. He also co-edited a special issue of IEEE Computer dedicated to Unmanned Vehicles, and special issue of The European Journal Embedded Systems focused on intelligent autonomous vehicles. He was elected as fellow of the IEEE, in 2014, for his contributions in high performance computer vision algorithms for airborne applications. He also served as the elected chair of the IEEE Mohawk Valley Section, Region 1, FY 2013 and FY 2014. He is a member of the honor societies: Tau Beta Pi, Eta Kappa Nu, Upsilon Pi Epsilon and Phi Beta Delta; and, a Paul Harris fellow of Rotary International.

Qing Wu received the BS and MS degrees from the Department of Information Science and Electronic Engineering, Zhejiang University, Hangzhou, China, in 1993 and 1995, respectively, and the PhD degree from the Department of Electrical Engineering, University of Southern California, Los Angeles, CA, in 2002. He was an assistant professor with the Department of Electrical and Computer Engineering, State University of New York at Binghamton, Binghamton, NY. He is currently a Principal Electronics engineer with the United States Air Force Research Laboratory, Information Directorate, Rome, NY. He has authored or co-authored more than ninety research papers in international journals and conferences. His current research interests include neuromorphic computing architectures, high-performance computing architectures, deep neural networks and memristor-based neuromorphic circuits and systems.