2017 29th International Symposium on Computer Architecture and High Performance Computing

Accelerating Graph Analytics on CPU-FPGA Heterogeneous Platform

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

***Abstract*—Hardware accelerators for graph analytics have gained increasing interest. Vertex-centric and edge-centric paradigms are widely used to design graph analytics accelerators. However, both of them have notable drawbacks: vertex-centric paradigm requires random memory accesses to traverse edges and edge-centric paradigm results in redundant edge traversals. In this paper, we explore the tradeoffs between vertex-centric and edge-centric paradigms and propose a hybrid algorithm which dynamically selects between them during the execution. We introduce the notion of active vertex ratio, based on which we develop a simple but efficient paradigm selection approach. We develop a hybrid data structure to concurrently support vertex-centric and edge-centric paradigms. Based on the hybrid data structure, we propose a graph partitioning scheme to increase parallelism and enable efficient parallel computation on heterogeneous platforms. In each iteration, we use our paradigm selection approach to select the appropriate paradigm for each partition. Further, we map our hybrid algorithm onto a state-of-the-art heterogeneous platform which integrates a multi-core CPU and a Field-Programmable Gate Array (FPGA) in a cache coherent fashion. We use our design methodology to accelerate two fundamental graph algorithms, breadth-first search (BFS) and single-source shortest path (SSSP). Experimental results show that our CPU-FPGA co-processing achieves up to 1.5× (1.9×) speedup for BFS (SSSP) compared with optimized baseline designs. Compared with the state-of-the-art FPGA-based designs, our design achieves up to 4.0× (4.2×) throughput improvement for BFS (SSSP). Compared with a state-of-the-art multi-core design, our design demonstrates up to 1.5× (1.8×) speedup for BFS (SSSP).**

I. Introduction

Emerging applications in broad areas including social networks, bioinformatics, and information networks require fast and efficient large-scale graph analytics [1]. To handle the graphs produced by these applications, many graph analytics engines have been proposed [2], [3], [4]. These engines are based on software and target general-purpose processors. Recently, accelerating graph analytics using hardware has been an area of growing interest in the community [5-19].

With the increased focus on energy-efficient acceleration, heterogeneous architectures integrating CPU and FPGA have become attractive platforms to deliver both high performance and low cost [13], [20], [21]. FPGA vendors have integrated general-purpose ARM processor and state-of-the-art FPGA into the same chip [22], [23]. A new trend is to couple CPU and FPGA through cache coherent interconnect. Intel and IBM have developed coherent memory interconnect technologies to provide coherent shared-memory access between CPU and FPGA [24], [25]. This enables FPGA to be a peer to CPU from a memory access standpoint, eliminating the need to move data back and forth between CPU and FPGA.

Graph algorithms have data-driven computation dictated by the vertices and edges of the graph. Vertex-Centric Paradigm (VCP) [2] and Edge-Centric Paradigms (ECP) [3] have been widely used to design graph processing engines. However, both of them have notable drawbacks, making it challenging to exploit FPGA to achieve efficient acceleration. VCP requires random memory accesses to traverse edges [3], [6]. Long-latency random memory accesses can result in accelerator stalls, potentially leading to limited performance gains [6]. ECP traverses edges in a streaming fashion, making FPGA an appropriate platform for acceleration [17]. However, ECP results in redundant edge traversals for non-stationary graph algorithms (e.g., BFS and SSSP) [29], [3]. In this paper, we explore the tradeoffs between VCP and ECP, and propose a hybrid algorithm to accelerate graph analytics on a CPU-FPGA heterogeneous platform. Our main contributions are:

- We conduct a detailed comparison between VCP and ECP. Based on their key characteristics, we propose a hybrid algorithm which dynamically selects between them during the execution.
- We propose a graph partitioning scheme to enable efficient concurrent execution on heterogeneous platforms. For each partition, we select the appropriate paradigm based on a simple paradigm selection approach.
- We develop an FPGA accelerator to accelerate our hybrid algorithm. We implement our approach on a state-of-the-art heterogeneous platform that supports coherent shared-memory between CPU and FPGA.
- Compared with state-of-the-art FPGA-based designs, our design achieves up to 4.0× (4.2×) throughput improvement for BFS (SSSP). Compared with a state-of-the-art multi-core design, our design achieves up to 1.5× (1.8×) speedup for BFS (SSSP).

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. Equipment grant from the Intel Hardware Accelerator Research Program is gratefully acknowledged.

978-1-5090-1233-6/17 $31.00 © 2017 IEEE
DOI 10.1109/SBAC-PAD.2017.25

The rest of the paper is organized as follows. Section 2 covers background. Section 3 introduces our hybrid algorithm. Section 4 describes implementation details. Section 5 discusses experimental results. Section 6 presents related work. Section 7 concludes the paper.

II. Background

A. Vertex-centric and Edge-centric Paradigms

Vertex-centric paradigm (VCP) [2] follows a scatter-gather processing model. The computation is iterative, each iteration consisting of a scatter phase followed by a gather phase. Algorithm 1 shows the general computation template of VCP. In the scatter phase, the vertices that have updates (e.g., the attribute of a vertex has been updated in the previous iteration) send the updates to their neighbours. Such vertices are defined as active vertices. The outgoing edges of active vertices are defined as useful edges. In the gather phase, updates are performed and the vertices that are updated become active vertices in the next iteration. One key issue of VCP is that vertices require random memory accesses through indices or pointers to traverse their edges [3], [6]. The random memory accesses are highly irregular such that conventional prefetching and caching strategies are not able to efficiently handle them, resulting in dramatically higher memory access latency. In this scenario, accelerator stalls and the performance significantly deteriorates [3], [6].

Algorithm 1 Vertex-centric paradigm (VCP)

```
1:  while not done do
2:    Scatter:
3:    for each vertex v do
4:      if v has update u then
5:        send u to neighbours (through v's outgoing edges)
6:      end if
7:    end for
8:    Gather:
9:    for each update u do
10:     if update condition is met then
11:       update vertex u.dest
12:     end if
13:   end for
14: end while
```

Edge-centric paradigm (ECP) [3] also follows the iterative scatter-gather processing model. Algorithm 2 illustrates the general computation template of ECP. The gather phase of ECP is the same as VCP, but the scatter phase is quite different: ECP sequentially traverses **all** the edges in the scatter phase. This eliminates random memory accesses to edges and enables to read edges from external memory in a streaming fashion. However, for non-stationary graph algorithms [29], in which not all the vertices are active in every iteration, it is likely that there are only a few active vertices in an iteration; in this scenario, ECP leads to substantial redundant edge traversals since it traverses all the edges rather than just the useful edges.

Algorithm 2 Edge-centric paradigm (ECP)

```
1:  while not done do
2:    Scatter:
3:    for each edge e do
4:      if vertex e.src has update u then
5:        send u to vertex e.dest
6:      end if
7:    end for
8:    Gather:
9:    for each update u do
10:     if update condition is met then
11:       update vertex u.dest
12:     end if
13:   end for
14: end while
```

B. CPU-FPGA Heterogeneous Platform

Heterogeneous architectures integrating CPU and FPGA have become attractive platforms for high-performance computing with low cost [22], [23], [24], [25]. Since processing units optimized for fast sequential processing (CPU) and processing units optimized for massive parallelism (FPGA) coexist, such architectures can efficiently cope with the workloads that require variable amounts of parallelism across the execution. There has been a new trend to integrate CPU and FPGA through cache-coherent interconnect [24], [25]. Intel and IBM have developed server-class products that integrate CPU and FPGA using cache-coherent interconnect technologies [24], [25]. Such heterogeneous platforms allow FPGA to directly read from and write to the memory hierarchy of CPU, making FPGA a peer to the CPU from a memory access standpoint. Compared with conventional interconnect technologies (e.g., PCIe), cache-coherent interconnect eliminates the need to move data back and forth between CPU and FPGA, enabling to offload specific workloads to FPGA for acceleration in a fine-grained manner.

III. Hybrid Algorithm

A. Motivation

We define **active vertex** (in an iteration) as a vertex that has an update to send to its neighbors, and **active vertex ratio** as the number of active vertices over the total number of vertices. Our hybrid algorithm targets non-stationary graph algorithms, in which only a subset of the vertices are active in each iteration. Example algorithms include breadth-first search (BFS), single-source shortest path (SSSP), weakly connected component, and community detection. Our hybrid algorithm is motivated by the fact that for such graph algorithms, the active vertex ratio varies over the iterations, especially when the input graphs follow power-law structure [10], [13]. The key idea of our hybrid algorithm is: (1) when the active vertex ratio in an iteration is low, we adopt VCP to traverse edges (small amount of random memory accesses are favored over large amount of redundant edge traversals); (2) when the active vertex ratio in an iteration is high, we adopt ECP to traverse edges (small

amount of redundant edge traversals are favored over large amount of random memory accesses).

B. Hybrid Data Structure

We assume the input graph is initially stored based on the coordinate (COO) format [26], which is a widely used storage format for graphs [3], [18]. VCP and ECP have different data structure requirements [2], [3], [18]. We propose a hybrid data structure to concurrently support VCP and ECP.

Given a graph $G = (V, E)$ with $|V|$ vertices and $|E|$ edges, the COO format stores the graph as an edge array with $|E|$ elements; each edge is represented as a *<src, dest, weight>* tuple, which specifies the source vertex, destination vertex, and weight of the edge; the edge array has been sorted based on source vertices. Besides the edge array, our design maintains a vertex array with $|V|$ elements. Each vertex has an algorithm-specific '*attribute*', which records the attribute value of the vertex. For example, for BFS, the attribute refers to the BFS level of the vertex; for SSSP, the attribute refers to the shortest path length between the vertex and the source vertex. Further, we partition the vertex array into sub-arrays of equal size, each of which is defined as an **interval**. Assuming the vertex array is partitioned into P intervals, the i-th interval ($0 \leq i < P$) includes the vertices with indices from $i \times \frac{|V|}{P}$ to $(i+1) \times \frac{|V|}{P} - 1$. All the edges whose source vertices belong to the same interval (i.e., outgoing edges of the vertices in the interval) constitute a **shard** of the interval. Note that since the edge array has been sorted based on source vertices, each shard is a continuous sub-array of the edge array. For each interval, we maintain an **update bin** which is an array to store the updates whose destination vertices belong to the interval. An update consists of a *<dest, value>* pair, in which '*dest*' refers to the destination vertex of the update and '*value*' is used to update the attribute of destination vertex. The partitioning scheme increases the available parallelism because the computations of different intervals can be performed in parallel. In addition, we can choose the interval size $|I|$ such that the vertex data of each interval fit in on-chip memory (e.g., cache); as a result, the vertex data can be repeatedly accessed from on-chip memory when processing the shard of the interval.

The above data structure supports ECP, but does not support VCP. The reason is that given a vertex, the memory location of its outgoing edges is unknown; therefore, the vertex is not able to access its edges. To resolve this issue, for each vertex, we keep a '*pointer*' to record the index of its first outgoing edge in the edge array, through which each vertex can quickly locate its edges. To indicate whether vertices are active or not in an iteration, we assign an '*active_tag*' to each vertex, which records the most recent iteration in which the attribute of the vertex was updated. The vertices with '*active_tag*' value of i become active vertices in the $(i+1)$-th iteration. In Figure 1, we show the data structures for an example graph, assuming the vertex array is partitioned into 2 intervals ($|I| = 2$).

C. Hybrid Algorithm

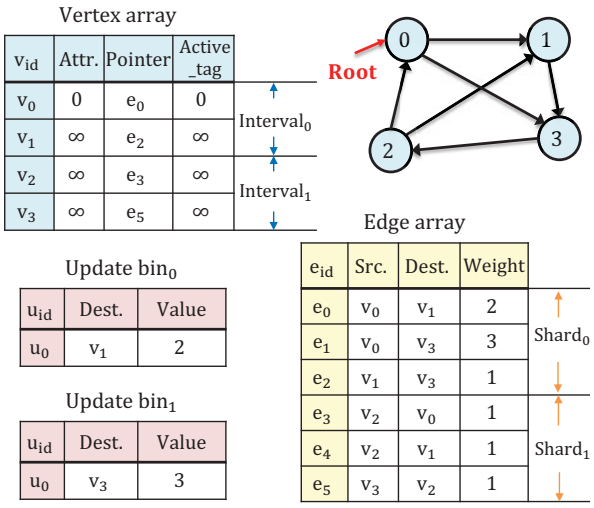

Figure 1: Example graph and its associated data structures

1) Paradigm Selection: The scatter phase includes 3 types of operations, namely reading edges from memory, computing updates, and writing updates into the update bins in memory. Since the execution for computing updates and writing updates are similar between VCP and ECP, reading edges results in the most significant performance difference [3]. **In each iteration**, we select the appropriate paradigm **for each interval** based on the **active vertex ratio of the interval**. Let $|I|$ denote the number of vertices in an interval, BW_{VCP} (BW_{ECP}) denote the sustained memory bandwidth for reading edges based on VCP (ECP), and r denote the active vertex ratio of an interval (i.e., the number of active vertices in the interval divided by $|I|$).

Proposition 3.1: In the scatter phase, if $r > BW_{VCP}/BW_{ECP}$, ECP results in lower execution time; otherwise, VCP results in lower execution time.

Proof. Let $|S|$ denote the total number of edges in the shard of the interval, and D_e denote the number of bytes required to represent an edge. The execution time for reading all the edges based on ECP can be estimated as:

$$T_{ECP} = |S| \times D_e / BW_{ECP} \quad (1)$$

Let m denote the average degree of vertices ($m = |S|/|I|$) in the interval. The execution time for reading all the edges of active vertices based on VCP can be estimated as:

$$T_{VCP} = r \times |I| \times m \times D_e / BW_{VCP} \quad (2)$$

By comparing Equations (1) and (2), we can obtain that when $r > BW_{VCP}/BW_{ECP}$, T_{ECP} is smaller; otherwise, T_{VCP} is smaller. □

Let r_{thold} denote the threshold for determining whether to select VCP or ECP (i.e., $r_{thold} = BW_{VCP}/BW_{ECP}$). We use the first two iterations to estimate BW_{VCP} and BW_{ECP} in order to determine r_{thold}: in the first iteration, we enforce

to select ECP to estimate BW_{ECP}; in the second iteration, we enforce to select VCP to estimate BW_{VCP}. We estimate the sustained memory bandwidth for reading edges based on Eq. (3), in which D_e denotes the number of bytes required to represent an edge and $T_{scatter}$ denotes the execution time of the scatter phase in the corresponding iteration.

$$BW = \#\_of\_accessed\_edges \times D_e / T_{scatter} \quad (3)$$

After r_{thold} is determined, at the beginning of each iteration, we select the appropriate paradigm for each interval based on Algorithm 3. The scatter phase of the intervals that are added into the VCP_queue (ECP_queue) will be performed based on VCP (ECP).

Algorithm 3 Paradigm selection

Let P denote the total number of intervals
Let I_i denote the i-th interval $(0 \leq i < P)$

```
for i from 0 to P − 1 do
    if I_i.no_of_active_vertices > |I| × r_thold then
        ECP_queue.enqueue(I_i)
    else if I_i.no_of_active_vertices > 0 then
        VCP_queue.enqueue(I_i)
    end if
end for
```

2) Scatter Phase: The scatter phase of the intervals in VCP_queue and ECP_queue is performed based on Algorithms 4 and 5, respectively. We accelerate the scatter phase by CPU-FPGA co-processing. To coordinate the execution between CPU and FPGA, we develop a runtime system. As shown in Figure 2, (1) when both VCP_queue and ECP_queue are not empty, CPU and FPGA concurrently execute the scatter phase of the intervals in VCP_queue and ECP_queue, respectively; (2) when VCP_queue is empty but there are still remaining intervals in ECP_queue, the runtime system employs a work-stealing strategy to achieve load balancing between CPU and FPGA; in this scenario, CPU 'steals' an interval from ECP_queue and executes its scatter phase. We use FPGA to accelerate the scatter phase of the intervals in ECP_queue because (1) we observe that the total execution time is dominated by executing the scatter phase of the intervals with high active vertex ratio and (2) the streaming nature of ECP makes FPGA suitable for acceleration [3], [17].

Note that when updates are written into the update bins in the memory, atomic operations (e.g., exclusive access to shared data) are required. This is because the updates are written based on the destination vertices; it is likely that multiple processing units (i.e., CPU cores and FPGA) concurrently write updates into the same update bin. Thus, one shared counter is required for each update bin to keep track of the number of updates stored in each update bin.

3) Gather Phase: The gather phase of distinct intervals can be **independently** executed in parallel. This is because the updates stored in an update bin will be only performed on the vertices in the corresponding interval. In the gather phase, we also keep track of the number of updated vertices in each interval, which can be used to compute the active vertex ratio of the interval in the next iteration.

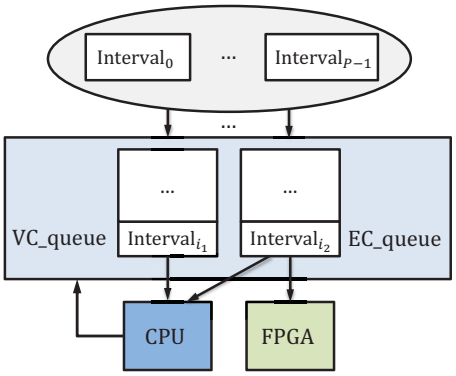

Figure 2: Coordination between CPU and FPGA by runtime system

Algorithm 4 VCP-based scatter

```
function VCP_scatter(Interval):
for each vertex v in Interval do
    if v is active with update u then
        for each outgoing edge e of v do
            write u into the Update Bin of I_⌊e.dest×P/|V|⌋
        end for
    end if
end for
```

Algorithm 5 ECP-based scatter

```
function ECP_scatter(Interval):
for each edge e in the Shard of Interval do
    if vertex e.src is active with update u then
        write u into the Update Bin of I_⌊e.dest×P/|V|⌋
    end if
end for
```

IV. Implementation

Our approach targets a heterogeneous platform with coherent shared-memory between CPU and FPGA. Examples include Intel-Altera Heterogeneous Architecture Research Platform (HARP) [24]. The platform integrates an Intel Xeon multi-core processor with an Altera FPGA through cache-coherent QuickPath Interconnect (QPI) technology [24]. On FPGA, a control unit for receiving control signals from CPU and a 4KB device status memory for storing the status of FPGA are provided. Users can implement customized Accelerator Function Unit (AFU) on FPGA, which is able to coherently access the CPU's last-level cache (i.e., L3 cache) and the DRAM attached to the CPU through QPI.

A. Overall Architecture

Figure 3 depicts the overall architecture of our design. The vertex array, edge array, and update bins are stored in DRAM.

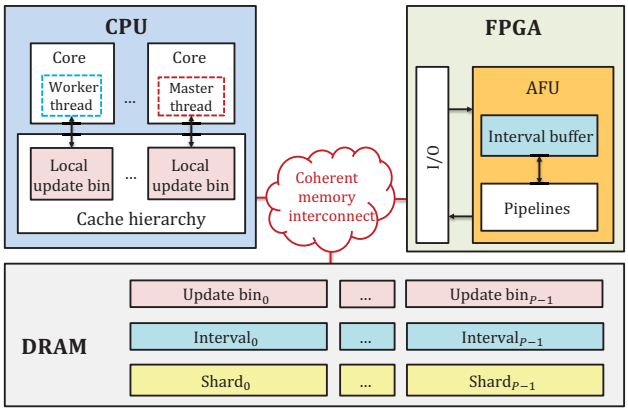

Figure 3: Overall architecture

On CPU, a master thread is created to schedule the execution and coordinate with FPGA. The master thread creates a group of worker threads to execute the computations for distinct intervals in parallel. Each thread maintains a local update bin; in the scatter phase, the produced updates are first written into the local update bins; when a local update bin becomes full, the corresponding thread will write the updates into DRAM. Having the local update bins in cache is in order to avoid frequent expensive atomic operations for writing updates into the update bins in DRAM.

The master thread controls FPGA by sending control signals to the control unit on FPGA. Based on the control signals, FPGA obtains the memory addresses of the data to be processed and starts processing. During processing, FPGA sets its device status to '*busy*'; when FPGA completes processing, it sets its device status to '*free*' to indicate that it is ready to process another interval. In the scatter phase, the updates produced by FPGA are first written into the local update bin of the master thread. The master thread is responsible for writing them into DRAM.

B. Accelerator Function Unit Design

FPGA accesses the shared-memory in blocks of cache lines. The cache line size (e.g., 64 bytes) can be much larger than the data size of an edge. In order to fully utilize the data in a cache line, we design a multi-pipeline architecture for the AFU. Assuming the cache line size is CL bytes and each edge is represented using D_e bytes, the AFU has CL/D_e pipelines working in parallel. Figure 4 depicts the AFU architecture. All the pipelines connect to an interval buffer which is composed of on-chip Block RAMs (BRAMs). When FPGA executes the scatter phase for an interval, the vertex data of the interval are prefetched into the interval buffer; as a result, the pipelines can access the vertex data only from the interval buffer other than from DRAM. Each pipeline consists of a BRAM access module and a compute module. The BRAM access module reads the vertex data from the interval buffer when edges are streamed in. The compute module is responsible for computing the update based on the attribute of vertex and edge weight. The update filter is used to filter out the invalid updates produced by non-active vertices. An invalid update can be identified by checking the '*active_tag*' of the source vertex that produces the update. Valid updates are first written into an output buffer on FPGA whose size is equal to the cache line size. When the output buffer becomes full, FPGA issues a memory write request to write the buffered updates into the local update bin of the master thread.

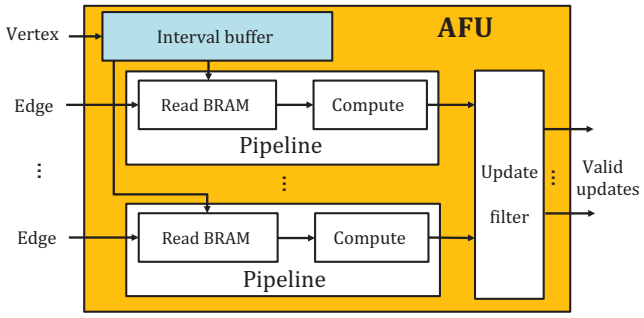

Figure 4: AFU architecture

V. Experimental Results

A. Experimental Setup

We implemented our designs on an Intel-Altera Heterogeneous Architecture Research Platform (HARP) [24]. The target platform integrates a 14-core Intel Xeon E5-2680 processor with an Altera Arria 10 GX1150 FPGA through QuickPath Interconnect (QPI) technology. Each CPU core operates at 2.4 GHz. The FPGA has 1,150,720 Adaptive Logic Modules (ALM) and up to 6.62 MB of on-chip BRAMs. The heterogeneous platform is equipped with 64 GB DDR3-1600 main memory. CPU can assess the main memory with the peak bandwidth of 30 GB/s. QPI enables the FPGA to assess the main memory with the peak bandwidth of 12.8 GB/s [24].

We report the resource utilization of our FPGA accelerators in Table I. They are evaluated using Quartus design software. For BFS (SSSP), each interval has 512K (128K) vertices and each vertex has a 8-bit (32-bit) attribute. We could not further increase the interval size because the accelerators have consumed up to 62.6% of the BRAMs in the FPGA device. For both BFS and SSSP, the FPGA accelerator runs at 200 MHz and the AFU contains 8 parallel pipelines to saturate the bandwidth available to the FPGA.

Table I: Resource utilization

Algorithm	Logic (ALM)	Register	Block RAM
BFS	7.6%	84892	62.6%
SSSP	7.6%	87395	62.6%

We generate synthetic graphs using the Graph 500 graph generator [1], which has been widely used [3], [8], [13], [15], [30]. Table II summarizes the key characteristics of the graph datasets. The pre-processing overhead to generate our hybrid

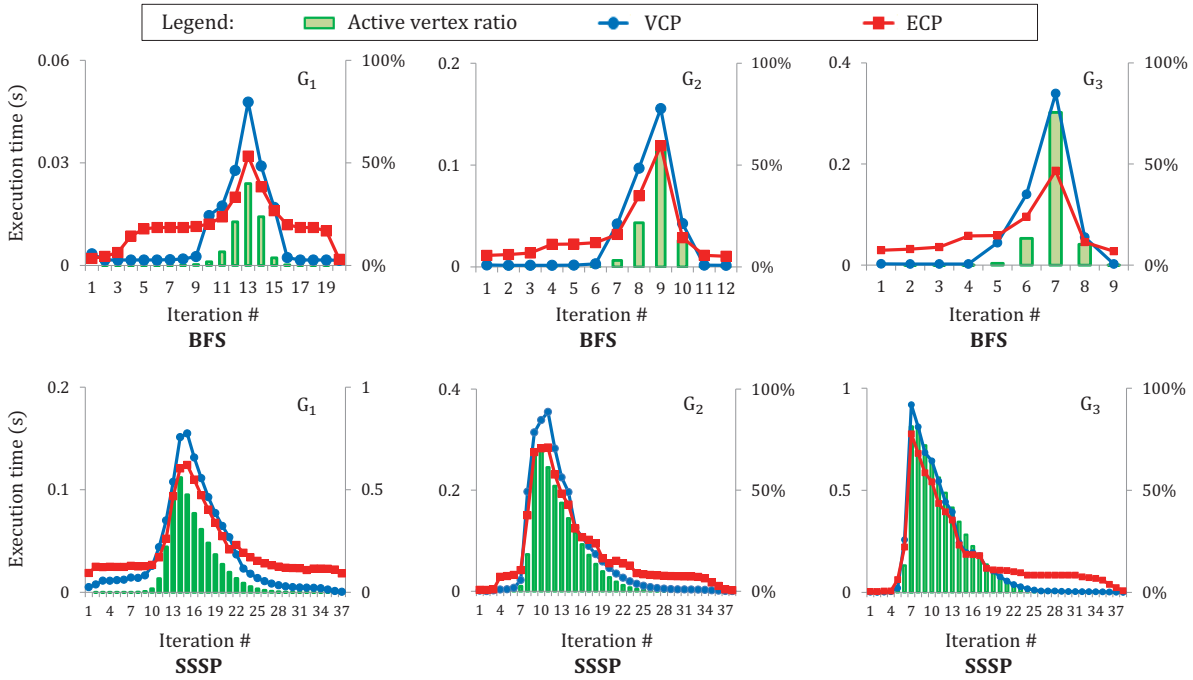

Figure 5: VCP vs. ECP on CPU

data structure (Section III-B) is also included in Table II. The pre-processing is performed by the CPU of the target platform. We assume the input graphs are initially stored in the COO format and do not change during the execution. Results in this work were generated using pre-production hardware and software from Intel, and may not reflect the performance of production or future systems.

Table II: Graph datasets

Notation	# Vertices ($\|V\|$)	# Edges ($\|E\|$)	$T_{pre-processing}$
G_1	10 M	40 M	0.005 s
G_2	10 M	80 M	0.012 s
G_3	10 M	160 M	0.025 s

We use execution time and throughput as the performance metrics for evaluations. The throughput metric is proposed by the Graph 500 community for performance comparison across various architectures and frameworks [1]. It is defined as the number of Traversed Edges Per Second (TEPS) [1].

B. VCP vs. ECP on CPU

To explore the tradeoffs between VCP and ECP, we first compare their performance on the CPU of the target platform. Figure 5 shows the execution time comparison of each iteration. It can be observed that the active vertex ratio has a significant impact on the execution time of each iteration for both VCP and ECP: when the active vertex ratio of an iteration increases, the execution time of the iteration increases as well. In addition, we observe that when the active vertex ratio is high, ECP results in lower execution time; whereas when the active vertex ratio is low, VCP results in lower execution time. We also notice that ECP sustains roughly 10× higher memory bandwidth than VCP for reading edges in the scatter phase.

C. Hybrid Algorithm on CPU

Further, we compare our hybrid algorithm (Section III-C) with VCP and ECP on the CPU of the target platform. Figure 6 shows the execution time comparison. For BFS, our hybrid algorithm achieves up to 1.4× (1.5×) speedup compare with VCP (ECP); for SSSP, our hybrid algorithm achieves up to 1.1× (1.3×) speedup compared with VCP (ECP).

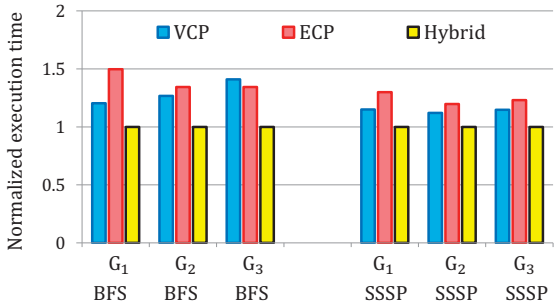

Figure 6: Execution time comparison among VCP, ECP, and hybrid algorithm on CPU

D. FPGA Acceleration for Hybrid Algorithm

We use our FPGA accelerator design (Section IV-B) to accelerate the hybrid algorithm. Figure 7 shows the speedup due to the FPGA acceleration. The CPU-FPGA co-processing design achieves up to 1.5× (1.9×) speedup for BFS (SSSP) compared with the CPU-only design. The achieved speedup is

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

Approach	Algorithm	Platform	Memory BW (GB/s)		Throughput (TEPS)	Generality
			CPU	FPGA		
[8]	BFS	12-core processor + Virtex 5	32	20	550 M	Supports BFS only
[15]		4-core processor + Kintex Ultrascale	17	60	166 M	
This paper		14-core processor + Arria 10	30	12.8	670 M	Supports various algorithms
[9]	SSSP	Virtex 7	–	31.8	18 M	Supports SSSP only
This paper		14-core processor + Arria 10	30	12.8	75 M	Supports various algorithms

mainly constrained by the QPI bandwidth (12.8 GB/s). When FPGA has a higher bandwidth to access the memory, more workloads can be offloaded onto the FPGA and our design will achieve even higher speedup.

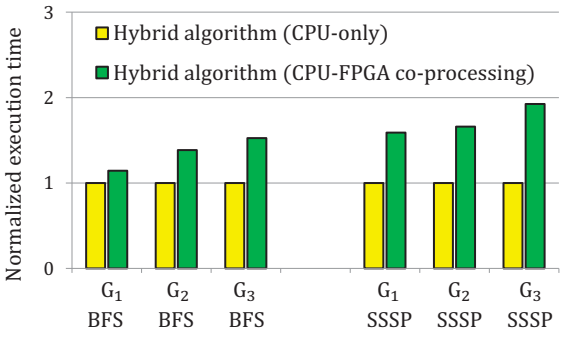

Figure 7: Accelerating hybrid algorithm by CPU-FPGA co-processing on heterogeneous platform

E. Comparison with State-of-the-art Designs

1) Comparison with FPGA-based Design: There are several designs to explore FPGA to accelerate BFS [8], [15] and SSSP [9]. These designs are highly optimized implementations with optimizations only applicable to the specific target algorithm. We compare with the state-of-the-art FPGA-based designs based on the throughput performance. Table III summarizes the comparison results. Our CPU-FPGA co-processing design achieves up to $4.0\times$ ($4.2\times$) throughput improvement for BFS (SSSP) with a even lower memory bandwidth.

2) Comparison with Multi-core Design: We further compare our design with a state-of-the-art multi-core design, GraphMat [4]. GraphMat is a highly optimized open-source graph processing framework and has demonstrated the best performance among existing software graph-processing frameworks. We execute GraphMat on our target platform. Table IV shows the execution time comparison with GraphMat. Our design achieves up to $1.5\times$ and $1.8\times$ speedup for BFS and SSSP, respectively.

VI. Related Work

A. Graph Processing Frameworks

Several software-based graph processing frameworks [3], [4] and hardware-based graph processing frameworks [7], [16], [17], [18], [27], [28] have been developed. 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 parallelism. However, these frameworks target homogeneous platforms and are designed based on either VCP or ECP.

Table IV: Comparison with state-of-the-art multi-core design

Algorithm	Dataset	Approach	Exec. time	Speedup
BFS	G_1	[4]	0.17 s	1.42×
		This paper	0.12 s	
	G_2	[4]	0.25 s	1.47×
		This paper	0.17 s	
	G_3	[4]	0.36 s	1.50×
		This paper	0.24 s	
SSSP	G_1	[4]	0.97 s	1.54×
		This paper	0.63 s	
	G_2	[4]	1.81 s	1.65×
		This paper	1.10 s	
	G_3	[4]	3.87 s	1.81×
		This paper	2.13 s	

B. Graph Analytics on Heterogeneous Platforms

Accelerating graph analytics on CPU-accelerator heterogeneous platforms has been studied in [8], [10], [13], [16], [30]. However, most of the existing designs mainly use the CPU of the heterogeneous platform for scheduling and pre-processing; when the accelerator is processing, CPU becomes idle. In [10], [13], the computations of BFS is dynamically mapped onto the CPU or the accelerator during the execution. However, each iteration is executed entirely either on the CPU or the accelerator, making the other idle. The design in [30] tries to concurrently execute the computations of each BFS iteration on both the CPU and GPU. However, the CPU-GPU co-processing design severely degrades the performance of each device. Compared with the existing designs, our design methodology enables efficient CPU-accelerator co-processing to fully utilize the computing resources of heterogeneous platforms. We also address the load balancing issue between the CPU and the accelerator.

C. FPGA-based Graph Analytics Accelerators

Using FPGA to accelerate graph analytics has sparked great research interest. However, many existing FPGA-based accelerators [6], [8], [9], [11], [12], [13], [15] are algorithm-specific and can not be easily extended to accelerate other graph algorithms. GraphGen [7] is an FPGA framework targeting general graph applications. It partitions the graph into sub-graphs and schedules the execution of each sub-graph. However, GraphGen requires the vertex data and edge data of a sub-graph to fit in the on-chip memory of FPGA. For large graphs, this can lead to a large number of sub-graphs and thus significantly increase the scheduling complexity. In [17], FPGA-based accelerators for several graph algorithms are proposed based on ECP. However, the work mainly focuses on optimizing memory and energy-efficiency performance, but does not address the redundant edge traversal issue of ECP. ForeGraph [19] is a multi-FPGA-based graph processing framework. It partitions the graph and uses multiple FPGAs to concurrently process distinct partitions. However, the performance is constrained by the communication overhead among the FPGAs.

VII. Conclusion

In this paper, we proposed a novel hybrid algorithm based on a CPU-FPGA heterogeneous platform to accelerate graph analytics. Our algorithm dynamically selected between vertex-centric and edge-centric paradigms to traverse edges. We developed an efficient paradigm selection approach based on the notion of active vertex ratio. We proposed a hybrid data structure and graph partitioning scheme to enable efficient concurrent execution on heterogeneous platforms. We implemented our design on a state-of-the-art heterogeneous platform to accelerate BFS and SSSP. Experimental results showed that with efficient CPU-FPGA co-processing, our design achieved up to 1.9× speedup compared with various highly optimized baseline designs. Compared with state-of-the-art FPGA-based designs, our design achieved up to 4.0× and 4.2× throughput improvement for BFS and SSSP, respectively.

Currently, our CPU-FPGA co-processing design selects the interval size based on the available on-chip memory resources of FPGA, without considering the size of on-chip cache of CPU. In the future, we plan to explore the impact of interval size on the performance by varying the interval size. We also plan to vary the threshold ratio to explore its impact on the performance. Future work will also involve extending our design methodology to accelerate other graph problems, such as weakly connected components and community detection.

References

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

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in Proc. of SIGMOD, pp. 135-146, 2010.

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

[4] 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.

[5] 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, vol. 6, 2011.

[6] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk, "A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration," in Proc. of ASAP, pp. 8-15, 2012.

[7] 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 FCCM, pp. 25-28, 2014.

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

[9] G. Lei, Y. Dou, R. Li, and F. Xia, "An FPGA Implementation for Solving the Large Single-Source- Shortest-Path Problem," IEEE Transactions on Circuits and Systems II, vol. 63, iss. 5, pp. 473-477, 2016.

[10] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-core CPU and GPU," in Proc. of PACT, pp. 78-88, 2011.

[11] S. Zhou, C. Chelmis, and V. K. Prasanna, "Accelerating Large-scale Single-source Shortest Path on FPGA," in Proc. of IPDPSW, 2015.

[12] S. Zhou, C. Chelmis, and V. K. Prasanna, "Optimizing Memory Performance for FPGA Implementation of PageRank," in Proc. of ReConFig, 2015.

[13] Y. Umuroglu, D. Morrison, and M. Jahre, "Hybrid Breadth-First Search on a Single-Chip FPGA-CPU Heterogeneous Platform," in Proc. of FPL, 2015.

[14] H. Giefers, P. Staar, R. Polig, "Energy-Efficient Stochastic Matrix Function Estimator for Graph Analytics on FPGA," in Proc. of FPL, 2016.

[15] 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 FPGA, 2017.

[16] T. Oguntebi and K. Olukotun, "GraphOps: A Dataflow Library for Graph Analytics Acceleration," in Proc. of FPGA, pp. 111-117, 2016.

[17] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and Energy-efficient Graph Processing on FPGA," in Proc. of FCCM, pp. 103-110, 2016.

[18] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A High-performance and Energy-efficient Accelerator for Graph Analytics," in Proc. of MICRO, 2016.

[19] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, H. Yang, "ForeGraph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture," in Proc. of FPGA, pp. 217-226, 2017.

[20] S. Zhou, W. Jiang, and V. K. Prasanna, "A Flexible and Scalable High-performance OpenFlow Switch on Heterogeneous SoC Platforms," in Proc. of IPCCC, 2014.

[21] Z. Li, L. Liu, Y. Deng, S. Yin, Y. Wang, and S. Wei, "Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware," in Proc. of ISCA, pp. 575-586, 2017.

[22] "Zynq UltraScale+ MPSoC," https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[23] "Stratix 10 SoC," https://www.altera.com/products/soc/portfolio/stratix-10-soc/overview.html

[24] "Xeon+FPGA Platform for the Data Center," https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[25] "POWER8 Coherent Accelerator Processor Interface (CAPI)," http://www-304.ibm.com/support/customercare/sas/f/capi/home.html

[26] J. Park, H. Chao, H. R. Arabnia, and N. Y. Yen, "Advanced Multimedia and Ubiquitous Engineering," in Future Information Technology, vol. 2, Springer, 2015.

[27] "nvGRAPH," https://developer.nvidia.com/nvgraph

[28] 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. of PPoPP, 2016.

[29] S. Singapura, A. Srivastava, R. Kannan, and V. K. Prasanna, "OSCAR: Optimizing SCrAtchpad Reuse for Graph Processing," in Proc. of HPEC, 2017.

[30] L. Remis, M. J. Garzaran, R. Asenjo, and A. Navarro, "Breadth-First Search on Heterogeneous Platforms: A Case of Study on Social Networks," in Proc. of SBAC-PAD, pp. 118-125, 2016.