An Efficient Graph Accelerator with Parallel Data Conflict Management

Pengcheng Yao[†] Long Zheng[†] Xiaofei Liao[†] Hai Jin[†] Bingsheng He[‡]

[†]Service Computing Technology and System Lab/Cluster and Grid Computing Lab/Big Data Technology and System Lab, Huazhong University of Science and Technology, Wuhan, 430074, P.R. China
[‡]School of Computing, National University of Singapore, Singapore, 117418, Singapore
{pcyao, longzh, xfliao, hjin}@hust.edu.cn, hebs@comp.nus.edu.sg

ABSTRACT

Graph-specific computing with the support of dedicated accelerator has greatly boosted the graph processing in both efficiency and energy. Nevertheless, their data conflict management is still sequential when certain vertex needs a large number of conflicting updates at the same time, leading to prohibitive performance degradation. This is particularly true and serious for processing natural graphs.

In this paper, we have the insight that the atomic operations for the vertex updating of many graph algorithms (e.g., BFS, PageRank, and WCC) are typically incremental and simplex. This hence allows us to parallelize the conflicting vertex updates in an accumulative manner. We architect AccuGraph, a novel graph-specific accelerator that can simultaneously process atomic vertex updates for massive parallelism while ensuring the correctness. A parallel accumulator is designed to remove the serialization in atomic protections for conflicting vertex updates through merging their results in parallel. Our implementation on Xilinx FPGA with a wide variety of typical graph algorithms shows that our accelerator achieves an average throughput by 2.36 GTEPS as well as up to 3.14x performance speedup in comparison with state-of-the-art ForeGraph (with its single-chip version).

1 INTRODUCTION

Graph processing plays an important role in many real-world applications, e.g., ranking the web sites [23], analysing the social networks [15], and discovering 3D motifs in protein structures [27]. Therefore, a large number of research efforts have been made to build the dedicated hardware that can execute graph applications with more efficiency than what the general-purpose processors can provide [6, 9, 17, 20].

Nevertheless, the graph algorithms may still suffer from considerable performance impacts caused by the atomic protections in existing graph accelerators. During the graph iterations, each vertex sends its value to all associated vertices. Therefore, it is common that many vertices may read/write the same vertex simultaneously, needing a significant number of atomic protections for preserving the correctness. This performance overhead arising from the atomic operations can be as much as nearly half of total graph execution, as demonstrated in previous work [16, 31] and also witnessed in our motivating study in Section 2.

Therefore, a lot of efforts have been made to reduce the atomic overheads. Recent researches significantly reduce the data access overheads by offloading the atomic operations to specialized memory [1, 16]. Some studies also attempt to reduce the number of atomic operations by sophisticated preprocessing, e.g., graph partition [6] and dynamic scheduling [20]. Unlike these previous work that concentrates on optimizing the individual atomic overhead, this work focuses on the performance impact of sequentiality between atomic operations, which is under-studied in graph processing.

Interestingly, graph processing for many graph algorithms (e.g., BFS, PageRank, and WCC) shows significant, common features for their atomic operations: 1) *incremental*–the atomic operations follow the commutative and associative law, 2) *simplex*–all atomic operations are similar. Instead of enforcing sequential execution of conflicting operations as traditional designs, this unique observation enables to execute massive conflicting vertex updates in an accumulative manner. Through simultaneously processing multiple atomic operations and merging the results in parallel, these vertex updates can be fully parallelized without changing final results. In this paper, we are addressing how we can design

PACT '18, November 1–4, 2018, Limassol, Cyprus

ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243201

such an efficient accumulator for parallelizing the conflicting data accesses for vertex updates in graph processing.

We propose AccuGraph, a novel accelerator that executes atomic operations in an accumulative manner. AccuGraph simultaneously processes multiple atomic operations for parallelizing the conflicting vertex updates while ensuring the correctness. A specialized accumulator is provided to remove the sequentiality in atomic opeartions through merging their results in parallel. Considering that the real-world graphs generally follow sparse and power-law topology [8, 15], the accumulator is designed to distinguish the process of low-degree and high-degree vertices. Internally, it executes multiple low-degree vertices in parallel for efficient edge-level parallelism, and limits the vertex parallelism for the high-degree vertices to avoid frequent synchronizations. To provide efficient vertex access, AccuGraph is also built with a high-throughput on-chip memory.

The contributions of this work are summarized as follows:

- We study a wide range of graph workloads and perform a detailed analysis on their atomic operations. We demonstrate that their distinct characteristics enable the parallel execution for conflicting vertex updates.
- We propose a graph-specific accelerator which supports parallel execution of atomic operations. A parallel accumulator is designed to guarantee efficient process of vertices with different degrees. A high-throughput on-chip memory is also provided for the efficient use.
- We compare our accelerator with the state-of-the-art ForeGraph. Experimental results with three graph algorithms on six real-world graphs show that our accelerator provides 2.36 GTEPS on average, outperforming ForeGraph by up to 3.14x speedup.

The rest of this paper is organized as follows. In Section 2, we introduce the background and provide our motivations and challenges in detail. Section 3 and Section 4 propose our accumulator designs and optimizations. The evaluation results are presented in Section 5. We survey related work in Section 6 and conclude the paper in Section 7.

2 BACKGROUND AND MOTIVATION

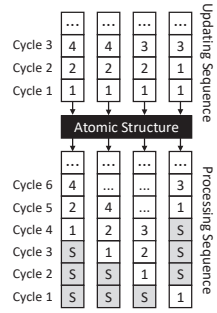
This section first reviews the vertex updating mechanism of existing graph accelerators for the conflicting data accesses. We next discuss its potential deficiency for graph processing through a motivating study, finally presenting our approach.

2.1 Modern Graph Accelerator and Its Data Conflict Management

In graph representation, each entity is defined as *vertex*, and its connection is defined as *edge*. The *degree* of a vertex denotes the number of connections. The operations in graph

```
Input: Graph G = (V, E), root vertex r
Output: Distance dis[V], initialized by ∞
1  dis[r] ← 0 ;
2  finished ← false;
3  while finished = false do
4      finished ← true;
5      for v ∈ V do
6          for u ∈ {k|(k, v) ∈ E} do
7              temp ← dis[u] + 1;
8              if temp < dis[v] then
9                  dis[v] ← temp;
10                 finished ← false;
11             end
12         end
13     end
14 end
```

(a) Pseudocode of BFS



(b) Execution flow of BFS

Figure 1: BFS pseudocode and its execution flow. In (b), the numbers indicate the ID of vertices to be updated, and the 'S' indicates a pipeline stall.

processing could be generally classified into *computing operations* that perform process on the edges, and *updating operations* that reduce their results to update the vertices [8, 26].

In existing graph processing frameworks, the vertices are shared and might be simultaneously accessed by multiple neighbors due to the complex graph connections [8, 15, 34]. As a result, there is a high coverage of data contentions for updating operations. For ensuring the correctness of vertex updating, existing researches often seek to use atomic structures (e.g., content addressable memory [21]), which tend to atomically protect the updates of each vertex if a conflicting data access to this vertex has been detected.

A typical procedure of data conflict management used in many graph accelerators [9, 20] is as follows. Multiple edges of the given vertices will be fetched and sent to the accelerator in each cycle. When receiving these edges, the accelerator will check the pipeline states at first. If an edge is connected with a vertex which is executing in the pipeline, its process will be stalled until the prior one finishes execution. In this way, the same vertex cannot appear in more than one pipeline stage at the same time, thus ensuring atomicity.

2.2 Inefficiency in Graph Processing

Graph often exhibits the complex connections where any vertex may be shared among different vertices. This is particularly true and serious for nature graphs that follow the power-law degree distribution, where some vertices have extremely large degree [8]. Thus, there involves a high risk that a large number of low-degree vertices simultaneously access the same high-degree vertex, leading to serious data contention. Unfortunately, modern graph accelerators (e.g., ForeGraph [6] and Graphicionado [9]) fall short in handling these highly-frequent data conflicts due to its serial semantics in atomic protection for the vertex updates.

Atomic Protection Analysis: Figure 1(a) illustrates the pseudo-code of *Breadth-First-Search* (BFS). It starts from a root vertex *r* and iteratively traverses the graph to calculate

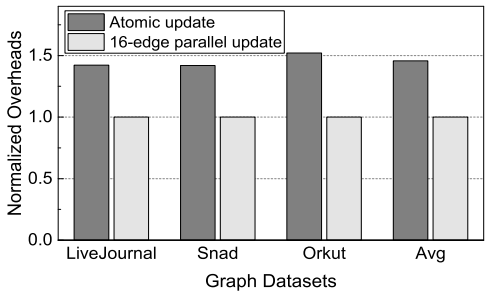
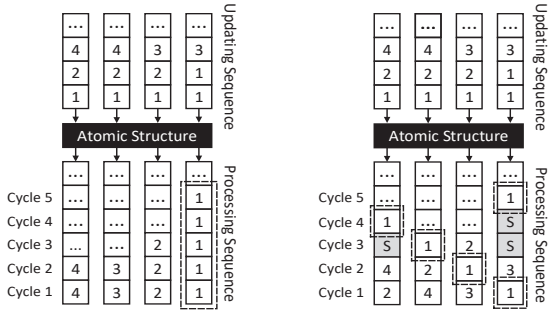


Figure 2: Normalized performance overhead caused by sequential atomic operations



(a) Reordering between all pipelines (b) Reordering inside each pipeline

Figure 3: Two mechanisms that avoid simultaneously scheduling conflicting updates

the shortest distance from the root vertex to other vertices. During the traversal, each vertex v will receive values *temp* from its neighbors (Line 7) and update itself (Line 8 and 9).

Figure 1(b) shows an example of the execution flow of four pipelines with atomic protection. In the first cycle, the accelerator receives four operations that all need to update the vertex one. Because of the atomic protection, these received operations from neighboring vertices have to be processed one-by-one in each cycle for preserving the correctness of final result. As a result, only the operation in the leftmost pipeline is processed while the others are stalled. These operations will not be released before receiving the completion of prior process. In other word, the process inside each vertex is enforced to be sequential for reducing data contention at the cost of performance.

Experimental Demonstration: We further make a set of experiments to investigate how much performance impact may be incurred by the atomic protection in graph processing. We use a cycle-accurate simulator to perform the vertex iteration with a parallel update for a maximal set of 16 edges[1]. Figure 2 depicts the comparative results. It is observed that the pure atomic protection leads to significant performance degradation for all real-world graphs, with 45% extra overheads on average in contrast to 16-edge parallel vertex update. This is particularly true and serious for those graphs that have the greater average degree (e.g., *Orkut*).

Remark: There are also a number of potential solutions that can be used for reducing the performance impact arising from atomic operations. ForeGraph [6] proposes a shuffling mechanism to reorder the edges with potential data conflicts. [20] excessively schedules destination vertices and process part of them based on a credit based mechanism. Similarly, the basic idea of the above solutions is to avoid simultaneously scheduling edges with the same vertex.

Figure 3 depicts two examples of the scheduling mechanisms used by them. Although these mechanisms effectively reduce the pipeline stalls, the updates of each vertex (e.g., the five updates of vertex one) are still enforced to be sequentially processed. As a result, the temporary vertex data is frequently synchronized between pipelines, leading to an increasing number of memory overheads. Moreover, such sequentiality would lead to unbalanced process for the power-law graphs. For example, the accelerator needs two more cycles to wait for the process of vertex one in Figure 3.

Some work [1, 16] uses novel *processing-in-memory* (PIM) technology [7] to offload the atomic operations to specialized memory region, which reduces the execution time of atomic operations. However, it needs to incorporate with specialized memory architecture and also increases the memory requests since all atomic operations need to be sent to the memory.

Table 1: Atomic operation types for the vertex update in different graph algorithms

Algorithm	Operation Type
Breadth-First Search	CAS if less
Weakly Connected Components	CAS if less
Shortest Path	CAS if less
PageRank	Atomic add
Triangle Counting	Atomic add
Degree Centrality	Atomic add
Collaborative Filtering	Atomic add

2.3 Potential of Accumulator

The key insight of this work is that atomic operations for many graph algorithms can be parallelized in an accumulative manner. Table 1 illustrates the typical operations that need an atomic protection for seven popular graph algorithms. We can observe that these atomic operations as a whole have two aspects of significant properties.

Observation 1: *The atomic operations for updating the conflicting vertex follow the commutative and associative law.*

The commutative law means that the execution sequence of the operations has no effect on the result. Associativity ensures the correctness of merging multiple operations. That is, any of the operations can be simultaneously merged without

[1]The simulation is conducted with a pipelined architecture that is similar to ForeGraph [6]. While data width of edges is usually 32-bits in BFS, we set 16-edge parallelism according to the memory access granularity (512-bits). Edge shuffling optimization [6] is not covered in our simulation.

changing the final result. For example, *PageRank* follows the atomic-add operations to update every vertex by following $Rank(v) = \varepsilon + \sum_{u \in neighbor(v)} Rank(u)/|neighbor(u)|$, where ε is a constant. Actually, no matter how we change the sequence or merge successive atomic-add operations, the final result can be still consistent.

When considering the potential computation error caused by floating-point values, the observation is still suitable for graph processing because the algorithms are iterative and inherently tolerate the imprecision [25].

Observation 2: *The atomic operations for updating the conflicting vertex are simple and used repeatedly.*

Taking *PageRank* as the example, we find that all of its atomic operations use the same atomic-add to sum up their values to the final result. This similarity allows to use a unique structure to merge all atomic operations.

These two observations consequently enable us to leverage existing well-developed accumulator to parallelize the conflicting vertex update. Accumulator is a hardware component that merges the inputs into a set of results with specific function. Based on the accumulator, we could simultaneously process the conflicting operations and merge their results in parallel to achieve fully-pipelined and balanced computation. Nevertheless, designing such accumulator for large-scale graph processing remains tremendously challenging.

Challenges: First, the real-world graph topology is often sparse with a low averaged degree. Traditional accumulator designs [3, 10, 12] often establish a fixed mapping relationship between the inputs and results. The reality is that the degrees of vertices are largely different during the iterations. Consequently, the traditional accumulators can only accumulate the atomic operations of a single low-degree vertex at the same time, leading to extremely low parallelism for graph processing. There remains a significant gap in applying the accumulation ideology into graph processing.

Second, natural graphs often follow a power-law distribution. For the high-degree vertices with a large number of edges that can be easily more than millions (e.g., `twitter`), an accumulator with limited width is extremely difficult to handle so many edges simultaneously. As a result, the accumulator will be invoked several times at the cost of increased synchronization overheads. Moreover, if multiple vertices are simultaneously processed in this case, it may lead to massive random edge accesses since their edges are likely to be non-sequentially stored. Therefore, there still lacks an effective technique that can improve the synchronization overheads and random accesses for an efficient accumulation.

Third, it is also difficult to predict the non-sequential neighboring vertices of each vertex in real-world graphs. Although the accumulator can largely reduce the atomic overheads, the random vertex access remains to be a potential bottleneck and significantly limits the throughput.

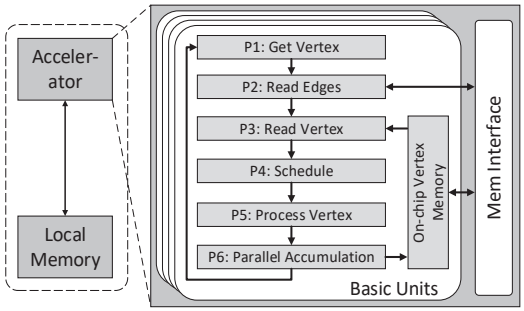


Figure 4: Architecture of AccuGraph. P_i denotes the ith pipeline stage.

2.4 Architectural Overview

Figure 4 shows an overview of our accelerator, which is designed with six pipeline stages in total. These stages basically serve as two major objectives as follows:

Designing an Efficient Accumulator (Section 3): As explained in the challenge discussions, the accumulator generally suffers from the sparse topology and power-law degree distribution in real-world graphs. To achieve desirable performance, the accumulator is expected to efficiently process both of the low-degree and high-degree vertices.

For the low-degree vertex, the accumulator (P6) is designed to simultaneously process multiple vertices for efficient parallelism. Since the vertex degrees are mutable during the process, it establishes a dynamic relationship between the input vertices and the final results to ensure the correctness.

For the high-degree vertex, the accumulator (P6) is designed with an specialized synchronization mechanism to reduce synchronization overhead. Moreover, for avoiding random edge accesses, a scheduling mechanism is proposed in P2 to be dynamically aware of the changes in degree and distinguish the schedule of different vertices.

Using Accumulator Efficiently (Section 4): While the accumulator could provide high execution efficiency, the on-chip memory is likely to be a potential performance bottleneck. To keep with the throughput of accumulator, the on-chip memory is partitioned into independent parts to process multiple accesses. Furthermore, considering the randomness and unbalance in vertex accesses, the on-chip memory would reorder them to ensure a high throughput.

3 PARALLEL ACCUMULATOR DESIGN

This section discusses the design guideline for a parallel accumulator as well as its core components for the efficiency.

3.1 Design Philosophy

Since accumulator is bounded with fixed width, it generally needs to consider two situations where skewed graph vertices with different degrees that can be greater or less than accumulator width, involving different parallel designs.

3.1.1 **Efficient Accumulation for Low-Degree Vertex.** Most of vertices for a natural graph have very low degrees which are always no more than the fixed number of ports for a typical accumulator. It is clear of a necessity to simultaneously process both updates of the same and different vertices for desirable parallelism.

Problem Definition: Assuming N updates, belonging to M vertices, need to be processed at once. This problem can be described by $p_j = \sum_{1 \le i \le N} a_i \cdot b_{ij}, 1 \le j \le M$, where p_j denotes the accumulated result of vertex j. a_i denotes the update i, and b_{ij} denotes whether a_i belongs to vertex j. The objective is to get all p_j with minimal latency.

Considering the locality of graph traversal, this problem can be further simplified. During traversal, edges of the same destination vertex are sequentially accessed in common graph representations, e.g., CSR/CSC [23]. It ensures that update values of the same destination vertex are sequentially received by the accumulator. Therefore, assuming that $C_j = [c_j^1, c_j^2]$ denotes the interval of vertex j's update values in all a_i, the problem could be simplified by $p_j = f(c_j^2)$, where

$$f(i) = \begin{cases} f(i-1) + a_i, & i \notin \{c_1^1, c_2^1, \ldots, c_M^1\} \\ a_i, & i \in \{c_1^1, c_2^1, \ldots, c_M^1\} \end{cases} \quad (1)$$

Solution: In Equation (1), we find that $f(i) = f(i-1) + a_i$ is a typical prefix-sum problem, which has been extensively studied [4, 10–12, 24]. Beyond the prefix-sum problem, a significant problem is that we still need to consider solving the otherwise case. This needs to 1): dynamically recognize the breakpoints that *break* the sequential computation and cancel the related operations, and 2): select the appropriate results since not all outputs are required. These are what we have additionally contributed to cope with.

3.1.2 **Efficient Accumulation for High-degree Vertex.** There are also many high-degree vertices that over-fit the width of an accumulator. Invoking the accumulator multiple times can be considered a useful approach by dividing these edges into multiple parts and processing one of them at the same time, but this costs more overhead.

Problem Definition: First, iteratively reading and writing back the temporary vertex data can lead to extra synchronizations. The accumulator is desired to reduce the number of synchronizing operations for high-degree vertex to improve the memory performance.

Second, the graph edges are sequentially stored with common data structure (e.g., *CSR/CSC*), which means that the edges of a high-degree vertex may be distributed to many continuous addresses. When multiple vertices are simultaneously processed in this case, their edges are located in non-adjacent addresses, leading to performance degradation. The accumulator should dynamically change the number of vertex scheduled to avoid random edge accesses.

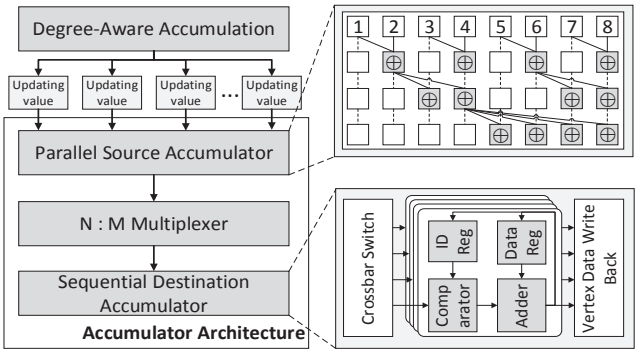


Figure 5: Architecture of parallel accumulator

Solution: Considering the locality of graph traversal that updates of the same destination vertex come in sequence, the first problem can be effectively solved. Such locality ensures that the results of multiple accumulations for the same high-degree vertex are also continuously generated. Therefore, the write back of the vertex data can be delayed before the accumulator sending a different vertex.

For the second problem, the inefficiency mainly comes from fixed granularity for vertex scheduling. Without considering the differences in the vertex degree, it schedules fixed number of vertices and simultaneously accesses their edges in each cycle. Therefore, the viable method is to sequentially access all edges and dynamically schedule the vertices based on the accessed edges, instead of accessing the edges based on the scheduled vertices.

3.2 Parallel Accumulator Architecture

Figure 5 shows the overview of our parallel accumulator, consisting of four parts. The *parallel source accumulator* and *multiplexer* provide efficient accumulation for the low-degree vertices, while the *sequential destination accumulator* and *degree-aware accumulation* reduce the synchronization and access overheads for the high-degree vertices.

Parallel Source Accumulator: To efficient accumulate the low-degree vertices, the parallel source accumulator dynamically recognizes the breakpoints and cancels the related operations based on prefix-sum adders [4, 11, 12, 24]. In this work, we choose Ladner-Fischer Adder [12] as the basis of our accumulator among a large number of previous efficient accumulators for three reasons as follow.

First, our main objective is to get the accumulated results in minimal latency, which filters the networks with depth larger than $\log(N)$. Second, among all networks with minimal latency, it has relatively fewer adders, which means that we could use fewer extra resources for breakpoint recognition and result selection. Finally, although its fanouts are relatively larger than others, it does not increase the length of critical path since its delay and route time are much smaller comparing to that of on-chip memory access.

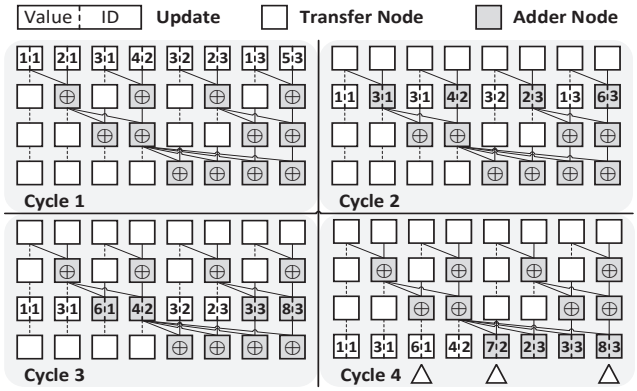


Figure 6: Execution flow of performing PageRank (with atomic-add operations) in the accumulator

Ladner-Fischer Adder opens a great opportunity for our graph-specific accumulator. In Ladner-Fischer Adder's original design, it establishes a fixed mapping relationship between the inputs and outputs, which leads to incorrect results when multiple vertices with mutable degrees are processed. Therefore, we complement a breakpoint recognizing mechanism. We add a new vector $V = (v_1, v_2, \ldots, v_N)$ where v_i denotes the destination vertex that a_i belongs to. With the vector V, the recognition conditions could be easily implemented by comparing the destination vertices of two inputs:

$$f(i) = \begin{cases} f(i-1) + a_i, & v_i = v_{i-1} \\ a_i, & v_i \neq v_{i-1} \end{cases} \quad (2)$$

We then attach each update value with the ID of its destination vertex in our design. To further reduce resource usage, we compress the destination vertex ID by only using its last $\log(m)$ bits, where m denotes the width of the accumulator. Based on Formula (2), the adder nodes (refer to the gray nodes) are modified to selectively aggregate the updates. Moreover, they could be easily adapted to different algorithms through replacing the accumulating logic.

Figure 6 shows the process to perform PageRank in a eight-width parallel source accumulator. Assuming that eight updates $A = \{1, 2, 3, 4, 3, 2, 1, 5\}$ belongs to three vertices $B = \{1, 1, 1, 2, 2, 3, 3, 3\}$. In the first cycle, the accumulator receives the updates and attaches them with vertex IDs in the first row as described above. In the left cycles, these updates are pushed to the next rows and processed by the transfer or adder nodes. The transfer nodes directly send the input updates to the output ports, while the adder nodes are implemented with the logics based on Formula (2). More specifically, each adder node aggregates the two input updates if their IDs are the same. Otherwise, it directly transfers the second input update to the output. In this way, both updates of the same and different vertices are processed in parallel, and ensured to be aggregated to separate locations.

Multiplexer: Once the updates are accumulated, the next is to dynamically select the results for each destination vertex from the output ports of parallel source accumulator. We use a $N : M$ multiplexer to implement such logics. Instead of directly comparing the destination vertex IDs, the multiplexer selects the data based on *edge offsets* to simplify the conditional logic. When the edges in pipeline stage P2 are accessed, each scheduled vertex is attached with its edge offset, indicating the last edge connected to it. Based on this information, the multiplexer is thus able to naturally select the data for each scheduled destination vertex in the ports related to its last edge. For example in Figure 6, the multiplexer selects the data from the ports pointed by triangles.

Sequential Destination Accumulator: To efficiently accumulate the high-degree vertices, we design a sequential destination accumulator for reducing the synchronization overheads. In light of the sequential arrival of accumulated values, it opens an opportunity to avoid synchronization on the temporary vertex data by delaying the write back of the destination vertex data until the accumulated value of a different vertex is received.

Therefore, the accumulator holds the destination vertex ID and the accumulated value in private registers. In each cycle, if the IDs in the input and the register are found to be the same, the accumulator will accumulate the vertex data in the input and register. Otherwise, the vertex data in the register would be written back and replaced by the input data. Furthermore, since the accumulator may simultaneously process multiple destination vertices, we replicate the destination vertex accumulators and use a crossbar switch to connect them with multiplexer. The crossbar switch routes the vertex data based on the destination vertex. That is, the last $\log(m)$ bits in its ID are used for m replications.

Degree Aware Accumulation: To reduce the random edge accesses for high-degree vertices, we propose a degree aware accumulation, as shown in Figure 7. The basic idea is to sequentially access all edges and dynamically schedule vertices based on the runtime information of their edge offsets (e.g., edge ID table in CSR/CSC [9] which denotes the location for the edges of each vertex).

More specifically, we use a specialized generator to automatically generate memory address for sequentially accessing all edges. In each cycle, every vertex pipeline stores received edge offsets of each vertex, and compares the top data in the FIFO with the generated memory address. If the memory address is within the range of two edge offsets, the top vertex would be scheduled and sent to the next stage. Moreover, if the memory address is equal to the right edge offset, which means all edges of the vertex have been read, the top vertex in the FIFO would be removed. In this way, the number of scheduled vertex is ensured to be the same with that of vertex contained in requested edges. Furthermore, the

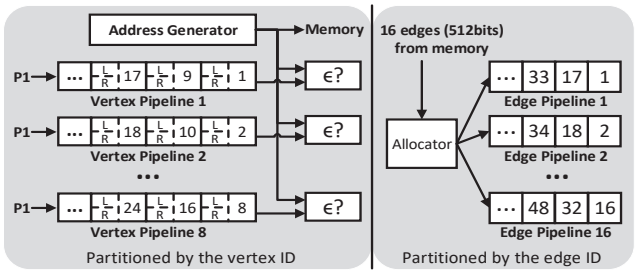


Figure 7: Degree aware accumulation with 8 vertex pipelines and 16 edge pipelines. 'L' and 'R' represent the memory address of the first and last edge for each vertex, respectively.

edge pipelines could be shared among all vertex pipelines to improve resource utilization.

Algorithm 1: Pseudocode of the rearranging mechanism

Input: Graph $G = (V, E)$, partition number P
Output: Rearranged edge list $NewEdge$

```
for v ∈ G do
    for u ∈ {k|(k, v) ∈ E} do
        Edge(v, u MOD P).push(u);
    end
    N(v) ← |{k|(k, v) ∈ E}|;
end
for v ∈ G do
    i ← 0;
    while N(v) > 0 do
        NewEdge(v).push(Edge(v, i).pop());
        i ← (i + 1) MOD P;
    end
end
```

4 OPTIMIZATIONS FOR EFFICIENT USE

In this section, we present several optimizations that are the key for using the proposed parallel accumulator efficiently.

4.1 Source Vertex Access Parallelization

While the above accumulator can provide reasonable execution efficiency, the memory access is likely to be a potential performance bottleneck. In practice, the neighbors of every vertex are discontinuous, leading to significant randomness in vertex access. Consequently, the vertex data is typically stored in on-chip memory (e.g., BRAM in FPGA) [6, 17, 20].

Despite that it could efficiently reduce the access latency, the throughput of on-chip memory is hard to keep with that of accumulator. Considering the limitation of capacity and frequency for on-chip memory in typical FPGA chips, memory partitioning [5, 30] is the most practical method to make the memory simultaneously process multiple requests. Typical memory partitioning mechanisms divide the memory into n independent parts and shuffle the requests to achieve a maximal throughput of n. Nevertheless, due to the randomness in vertex access, we find a significant number of requests are shuffled to the same memory partition in each cycle, leading to extra cycles to process these requests.

To balance the distribution of vertex access, we rearrange the edges of every vertex during preprocessing. Algorithm 1 represents the pseudocode of our mechanism. The basic idea is to rearrange the edges of each vertex to ensure that the address values are relatively balanced in cacheline-width granularity before processing the graph. Assuming that the memory is partitioned to 16 dependent parts, we would also maintain 16 queues for each vertex to store the edges based on the connected vertex's ID. During rearranging, we would iteratively select edges from each queue in sequence for every vertex. The overhead of rearrangement is about O(|E|), which is the same as that of compressing algorithms commonly used in graph processing (e.g., CSR/CSC). With the mechanism, the address values could be evenly rearranged, thus improving the memory performance.

Moreover, considering that some vertices might have more accesses on a specific memory partition (edge values themselves are unbalanced), we try to change processing granularity to deal with such imbalance. More specifically, we allow the on-chip memory to process the requests in an unblocking (out-of-order) manner. Through unblocked process, the idle memory ports could be utilized by the latter requests, thus improving memory efficiency.

Figure 8 shows the workflow of our mechanism. In each cycle, stage P3 receives N edges from memory, and shuffles them to different request FIFOs based on their values. The FIFOs cache these edges and send the requests generated by the top ones to the on-chip memory. To avoid the unblocked requests breaking sequentiality of edge access and further leading to incorrect results, a reorder stage is involved after accessing the source vertex data. The reorder stage caches the accessed vertex data, reorders them to match the sequence of original requests, and sends reordered data to stage P4. To implement such reordering logic, each memory request would be attached with a token based on the last $\log(m)$ of original edge memory address, where m denotes the size of buffer in reorder stage. All accessed data with the same token would be stored in the same location in reorder stage. Once the top data finishes reordering, i.e., all data of the first request has been received, it would be sent to the next stage.

4.2 Source-Based Graph Partition

While storing vertex data in on-chip memory could avoid costly random access in main memory, it might require a large number of resources that may exceed the capacity of the chip. Assuming the 4-byte width of vertex data and 8

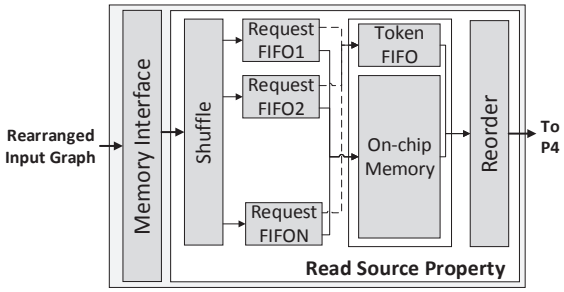


Figure 8: Workflow of accessing source vertex data

M vertices, the on-chip memory is desired to be larger than 32 MB, which is unpractical for most of FPGAs. To enable process of large-scale graphs without losing the benefit of on-chip memory usage, we partition the graph into several parts and process a single part at a time.

To ensure that all vertex data in each graph part could be held in on-chip memory, we use a source-based partition mechanism [8]. The partition mechanism works as follows. First, the vertices of the input graph are divided into K parts based on their vertex IDs. The value of K depends on the number of vertex and the capacity of on-chip memory. For each part, the out-edges of each vertex are also included. After the input graph is partitioned, our accelerator sequentially processes each graph part in each iteration. Since every edge would be partitioned to the graph part which includes its destination vertex, no edges need to be processed twice. The graph partition does incur some extra memory overheads, since the same destination vertex data might be read and written more than once. More specific impacts would be discussed in Section 5.5.

5 EVALUATION

This section evaluates efficiency of AccuGraph on typical graph algorithms with real-world graph datasets.

5.1 Experimental Settings

Evaluation Tools: We implement our accelerator on Xilinx Virtex Ultrascale+ XCVU9P-FLGA2104 FPGA with -2L speed grade. The target FPGA chip provides 1.18 M LUTs, 2.36 M registers, and 9.49 MB on-chip BRAM resources. We verify the correctness and get the clock rate as well as resource utilization using Xilinx Vivado 2017.1. All these results have passed post-place-and-route simulations. The off-chip memory requests are processed by one DRAM, which is Micron 4GB DDR4 SDRAM (MT40A256M16GE-083E) in our evaluation. We use DRAMSim2 [22] to simulate the cycle-accurate behavior of the off-chip access. The memory has a running frequency of 1.2 GHz and a peak bandwidth of 19.2 GB/s.

Graph Algorithms: We implement three well-known graph algorithms on our accelerator, covering both CAS-if and atomic-add operation types in Table 1.

Table 2: Graph datasets

Names	# Vertices	# Edges	Description
Slashdot	0.08 M	0.95 M	Link Graph
DBLP	0.32 M	1.05 M	Collaboration Graph
Youtube	1.13 M	2.99 M	Social Network
Wiki	2.39 M	5.02 M	Website Graph
LiveJournal	4.85 M	69.0 M	Follower Graph
Orkut	3.07 M	117 M	Social Network

- *Breadth First Search (BFS)* is a basic traversal algorithm utilized by many graph algorithms. It iteratively traverses the input graph and calculates the distance of shortest path from root to every vertex.
- *PageRank (PR)* is an important graph algorithm used to rank web pages according to their importance. It updates every vertex based on the formula $Rank(v) = \varepsilon + \sum_{u \in in-neighbor(v)} Rank(u)/|out - neighbor(u)|$ in each iteration, where ε is a constant.
- *Weakly Connected Components (WCC)* is an algorithm that checks the connectivity between two vertices in a graph. During the traverse, every vertex would receive the labels from all neighbors and update itself with the minimal one.

Graph Datasets: The graph datasets for the experiments are summarized in Table 2. All these graphs are real graph data sets collected from SNAP [13] and TAMU [28]. In our implementation, each undirected edge is treated as two directed edges between source destination vertex and processed twice. Therefore, the number of edges for undirected graphs (*DBLP*, *Youtube*, and *Orkut*) is considered double in our evaluation.

Table 3: Resource utilization and clock rate

	BFS	PR	WCC
LUT	7.39%	10.1%	8.26%
registers	2.53%	4.47%	3.02%
BRAM	57.9%	69.9%	69.9%
Maximal clock rate	256 MHz	211 MHz	251 MHz
Simulation clock rate	250 MHz	200 MHz	250 MHz

5.2 Overall Performance

Resource utilization: Table 3 shows the resource utilization and clock rate of the FPGA design with 8 vertex pipelines and 16 edge pipelines, which maximizes throughput given the peak DRAM bandwidth. First of all, because of the shared edge pipeline design described in Section 3.2, the number of resources required is reduced. Therefore, the logic resource (LUT and register) consumption of our accelerator is relatively low. Second, we implement the on-chip memory with BRAM resources to maintain vertex data. Similar to prior work [6], we use 1 byte integer to represent the depth value in BFS, single-precision floating point (4 bytes) in PR, and 4 bytes integer in WCC. In this way, the maximal memory requirement is $1 \times 4.85 = 4.85$ MB for 1 byte data and $4 \times$

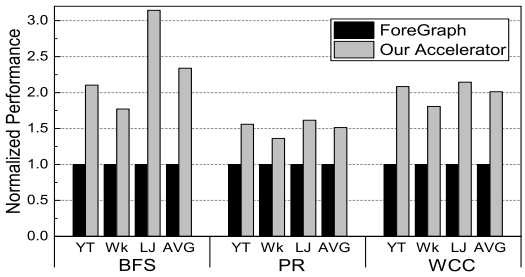


Figure 9: Our accelerator normalized to the ForeGraph performance. YT denotes graph *Youtube*, Wk denotes graph *Wiki*, and LJ denotes graph *LiveJournal*. AVG presents the average speedup of all tested graphs

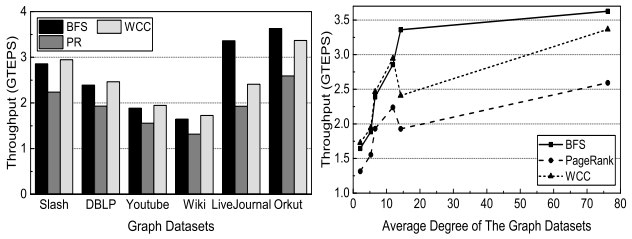
4.85 = 19.4 MB for 4 bytes data. Therefore, we hold all vertex data when running BFS and about 1.7 M vertex data for other algorithms, which consumes 57.9% and 69.9% of available BRAM resources, respectively. The UltraRAM resources are not used in our implementation.

Throughput: Figure 9 shows the normalized performance comparing to ForeGraph, which is one of the fastest graph processing accelerator implemented on FPGA, with respect to throughput. By throughput, we refer to the number of *traversed edges per second* (TEPS) [19], which is a performance metric frequently used in graph processing.

Since ForeGraph has not been open-sourced, we execute the same graph algorithms (BFS, PR, and WCC) and datasets (*youtube*, *wiki-talk*, and *LiveJournal*) used by its evaluation on our accelerator, and compare the results with the performance reported in its work (just as previous work has also done [6, 33]). When running PR and WCC on *Wiki*, the BRAM resources available in the FPGA chip used in ForeGraph is large enough (up to 16.6 MB) to hold all vertex data on-chip, which is unreliable for that of our FPGA chip (9.49 MB). Therefore, we compress the vertex data to 2 bytes when running PR and WCC on *Wiki* for fair comparison.

As shown in Figure 9, AccuGraph achieves 1.36x ∼ 3.14x speedup compared to the ForeGraph. As analyzed in Section 2.2, the speedup comes from the reduced synchronization overheads by simultaneously processing atomic operations. Moreover, our accelerator could achieve better load-balance using degree-aware accumulation by dynamically deciding the number of vertices scheduled.

For the results of different algorithms, we find that the speedup of PR is smaller. This is because of the lower clock rate caused by complex floating units. Since the number of edge pipelines is fixed in our implementation, the clock rate directly influences the overall performance. Moreover, the floating point units significantly increase the length of pipelines, thus would need more cycles when recovering from pipeline stalls. Therefore, the algorithms that use integer values could achieve slightly higher performance.



(a) Performance for different graphs (b) Different average degrees

Figure 10: Sensitive study on throughput with different graphs and average degrees

5.3 Sensitivity Study

To get a more comprehensive performance result, we execute all graphs described in Table 2 on our accelerator. The structures of these graphs significantly differ from each other (e.g., number of vertices and edges, average degree), thus providing an in-depth overview on the performance. As shown in Figure 10(a), AccuGraph achieves 1.4 GTEPS ∼ 3.5 GTEPS over all graph algorithms and datasets.

Among all graph datasets, *Wiki*'s throughput is particularly low when executing on AccuGraph. This is because *Wiki* is extremely sparse and makes the accelerator exhibits unbalance between the vertex and edge pipelines. With low average, the edges accessed from *Wiki* in each cycle prefer to belong to multiple vertices (more than 8). Therefore, the vertex pipelines might need more than one cycle to process these edges, leading to lower performance.

As shown in Figure 10(b), the performance is almost linearly increased when the average degree is less than 16. This is because that the percentage of low-degree vertex (≤ 2) decreases. Moreover, the performance improves slightly when increasing the average degree from 16 to 76. This is because that the memory bandwidth becomes the potential bottleneck in these cases, since it could only send a cacheline-width edges in each cycle. In summary, the performance improves as the average degree increases before reaching the limitation of maximal memory bandwidth.

Lastly, we find obvious performance degradation for PR and WCC when average degree is about 14 (*LiveJournal*). Moreover, the performance of PR and WCC is significantly lower than that of BFS when average degree is larger than 14 (*LiveJournal* and *Orkut*). This is because that the vertices data is too large to be all held in on-chip memory in these cases. Therefore, the graph partition mechanism is used when executing PR and WCC on these graphs, which involves in more vertex access. More detailed analysis of degree distribution and graph partition is presented in Section 5.4 and 5.5.

5.4 Benefit Breakdown

We next break down the respective benefits of our different graph accelerator designs as follow:

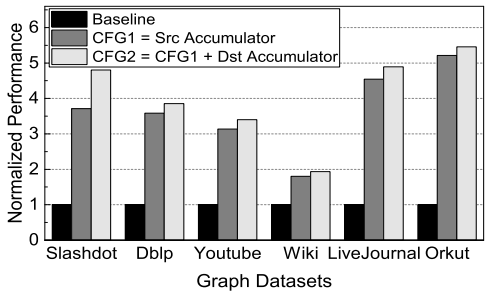
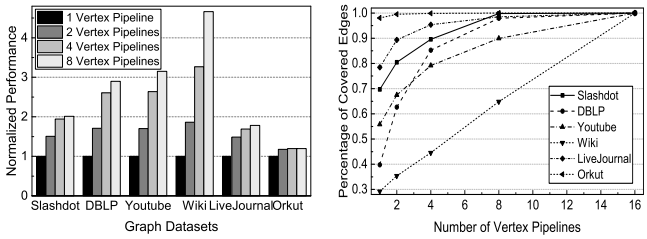


Figure 11: Benefit of parallel accumulation



(a) Performance (b) Percentage of covered edges

Figure 12: Benefit of degree-aware accumulation

Benefits from Parallel Accumulation: Figure 11 shows the normalized performance results. The baseline represents the basic design without any optimizations described in Section 3 and 4. It sequentially processes each edge, and accumulates its values to the final result in each cycle. CFG 1 represents source vertex accumulation. CFG 2 further uses destination vertex accumulation based on CFG1.

It is shown that CFG1 achieves 1.9x∼ 5.2x speedup compared to the baseline. Note that *Wiki* is lowest performance among all graph workloads. This is because that the number of vertex pipelines is set to one, leading to the fact that only one vertex can be scheduled in each cycle for CFG 1. Therefore, the number of edges sent to the accumulator in each cycle is directly depended on the average degree. In a word, the graphs with higher degree could experience higher speedup when using source vertex accumulator.

For CFG 2, destination vertex accumulator achieves about 1.3x speedup in most of graphs, except for *Slashdot* (2.0x speedup). This is because that *Slashdot* has self-loops, which means that some edges connect a vertex to itself. When processing these self-loops, the memory requests of source and destination vertex would be assigned to the same on-chip memory partition, leading to increased memory cycles. With the source vertex accumulator, the request of destination vertex could be avoid, thus improving the overall performance.

Benefits from Degree-aware Accumulation: Second, we explore the impact of degree aware accumulation on above accumulators. Figure 12(a) presents the results which assume that on-chip memory could process any 16 memory requests in each cycle. For the performance, we analyze the speedup brought by different number of vertex pipelines, which denotes the maximal parallelism of the accumulation[2].

We make the observation that the performance improves sub-linearly as the number of vertex pipelines increases. This is because of the power-law degree distribution of graphs. Assuming that the number of vertex pipelines is N, our degree aware mechanism could cover the vertices with degree $\geq 16/N$ with 16 edge pipelines. As depicted in Figure 12(b), the percentage of the covered edges for most graphs increases sub-linearly because high-degree vertices have most of the edges. While for *Wiki*, the skewness of its degree distribution is low, thus leading to an almost linear increment.

Benefits from Vertex Access Parallelization: Figure 13 explores the impact of different optimization for parallel accumulations, without ignoring the influence of the on-chip memory's throughput. The left most bar in Figure 13 represents the baseline case where only parallel accumulation is applied. CFG 3 represents the degree aware accumulation with 8 vertex pipelines based on CFG2. CFG 4 shows the effects of rearranging mechanism and CFG 5 shows the effects of reordering with buffer size of 64 discussed in Section 4.1.

The first observation is that the speedup of degree aware accumulation is decreased to about 1.3x when considering the influence of on-chip memory's throughput. Since the accumulator has already removed potential bottleneck in atomic operations and made the computations fully pipelined, the throughput of on-chip memory becomes the main bottleneck. Without any optimizations, there would be a significant amount of increased memory requests caused by the unbalanced edge values, thus decreasing its impact.

Another observation is that our rearranging mechanism could achieve 1.3x speedup and reordering mechanism could achieve another 1.5x ∼ 2.8x speedup. When increasing the reorder buffer size from 64 to 256, we only get 1.07x speedup for all tested graphs. This is because that the vertex requests do not put much pressure on the reordering buffer. Considering the randomness in vertex accesses, it is uncommon that more than two requests are simultaneously sent to the same memory partition after preprocessing. With these mechanisms, the increased memory requests could be reduced to $\leq 10\%$, which significantly improves the memory efficiency.

5.5 Scalability

When the vertex data can not be held in the on-chip memory, we would partition the graph to enable the process. Figure 14 explores the impact of graph partition described in Section 4.2. The leftmost bar represents the case where the on-chip memory size is large enough to hold all vertex data, denoted as partition number = 1. The other bars represent cases where on-chip memory size is only enough to hold $1/N$ of the total vertex data where N represents the number of partitions.

[2]When the number of vertex pipelines is set to N, the mechanism dynamically schedules $1 \sim N$ vertices based on the degree.

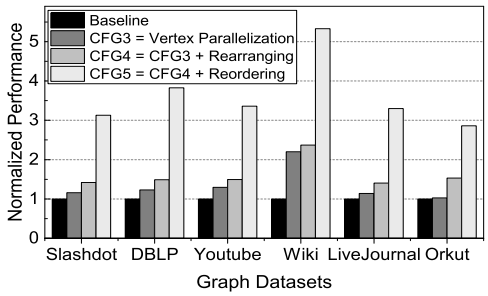


Figure 13: Effect of different optimizations in memory subsystem discussed in Section 4

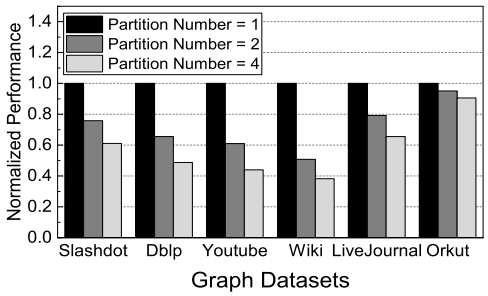


Figure 14: Effect of graph partition mechanism

In general, partitioning the graphs into 4 parts would result in around 40% performance degradation. Among all workloads, the *Wiki* experiences the largest performance degradation which reaches about 61%. This is because we would traverse all vertex in each sub-iteration when processing each graph partition. As the average degree decreases, the increased vertex access overheads would account for a significant percentage of total overheads. Therefore, the performance of graphs with lower average degree would be more sensitive to the partition number.

6 RELATED WORK

To improve the execution efficiency, a vast body of research efforts have been therefore put into making the graph-specific architectural innovations. Graphicionado [9] proposes a graph accelerator which efficiently utilizes large on-chip scratchpad memory. GraphGen and Graphops [17, 18] automatically compile graph algorithms to specialized graph processors. Compared with these prior researches with strict atomic protection, we argue that the heavy reliance on atomic operations leads to significant performance degradation and propose a novel accelerator to reduce atomic overheads.

There are also a large number of attempts that aim at reducing atomic overheads of graph processing. ForeGraph [6] uses a shuffling mechanism to avoid data contentions. [20] proposes a specialized synchronizing mechanism to avoid scheduling conflicting edges. Generally, their basic idea is to avoid scheduling the edges with conflict vertices. However, these accelerators still enforce to sequentially process the updates of the same vertex, which increases the synchronization overheads. Comparing to these work, we concentrate on improving such sequentiality and propose to process these updates in an accumulative manner. Moreover, comparing to the researches that improve the sequentiality based on general-purpose processors [14, 32], our accelerator needs less computation and avoids global synchronizations.

Some recent work concentrates on improving the performance of graph processing through utilizing GPU. Typical GPU-based graph processing frameworks simultaneously process thousands of vertices to achieve desirable performance and focus on balanced mapping. For example, Gunrock [29] proposes a hybrid mapping mechanism to achieve load-balance. Instead of using massive parallelism to cover synchronization overheads, our accelerator tries to explore the potential parallelism in limited number of scheduled vertices. Through accumulating the atomic operations, the parallelism of these vertices could be fully utilized.

Many other efforts have been put into improving the execution time of atomic operations. Tesseract [1] offloads all graph operations to memory-based accelerator to ensure atomicity without requiring software synchronization primitives. Some researches [2, 16] also enables offloading operations at instruction-level. Compared to these PIM-enabled graph architecture, our accelerator can achieve efficient management on shared data conflicts without introducing special memory components. Moreover, our parallel data conflict management can be also integrated into these accelerators and help to reduce the memory requests.

7 CONCLUSION

In this paper, we present a pipelined graph processing accelerator to enable massive parallelism of vertex updates. Our accelerator provides a parallel accumulator to simultaneously schedule and process multiple destination vertices without losing edge-level parallelism. Moreover, the accumulator is designed to be degree-aware and can adaptively adjust the vertex parallelism to different kinds of graphs. We also present vertex access parallelization and source-based graph partition for better supporting the efficient use of graph accelerator. Our evaluation on a variety of graph algorithms shows that our accelerator can achieve the throughput by 2.36 GTEPS on average, and up to 3.14x speedup compared to the stat-of-the-art FPGA-based graph accelerator ForeGraph.

ACKNOWLEDGMENT

The work is supported by National Key Research and Development Program of China under grant No. 2018YFB1003502. This paper is also supported jointly by NSFC under grant No. 61702201, 61732010, 61628204. To whom the correspondence should be addressed to Long Zheng.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 105–117.

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 336–348.

[3] Guy E. Blelloch. 1989. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*. 38, 11 (1989), 1526–1538.

[4] Richard P. Brent and Hsiang T. Kung. 1982. A Regular Layout for Parallel Adders. *IEEE Transactions on Computers*. 3 (1982), 260–264.

[5] Jason Cong, Wei Jiang, Bin Liu, and Yi Zou. 2011. Automatic Memory Partitioning and Scheduling for Throughput and Power Optimization. *ACM Transactions on Design Automation of Electronic Systems*. 16, 2 (2011), 15:1–15:25.

[6] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. Foregraph: Exploring Large-Scale Graph Processing on Multi-FPGA Architecture. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*. 217–226.

[7] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: The Terasys Massively Oarallel PIM Array. *IEEE Computer*. 28, 4 (1995), 23–31.

[8] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30.

[9] Tae J. Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. 1–13.

[10] Simon Knowles. 2001. A Family of Adders. In *Proceedings of IEEE Symposium on Computer Arithmetic*. 277–281.

[11] Peter M. Kogge and Harold S. Stone. 1973. A Parallel Algorithm for the Efficient Solution of A General Class of Recurrence Equations. *IEEE Transactions on Computers*. 100, 8 (1973), 786–793.

[12] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *Journal of the ACM*. 27, 4 (1980), 831–838.

[13] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[14] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-Accelerated Graph Processing on A Single Machine with Balanced Replication. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 195–207.

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 135–146.

[16] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468.

[17] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. 2014. Graphgen: An FPGA Framework for Vertex-Centric Graph Computation. In *Proceedings of IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 25–28.

[18] Tayo Oguntebi and Kunle Olukotun. 2016. Graphops: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*. 111–117.

[19] Graph 500 Origanization. 2018. Graph 500 Benchmark. http://graph500.org/.

[20] Muhammet M. Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*. 166–177.

[21] Kostas Pagiamtzis and Ali Sheikholeslami. 2006. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *IEEE Journal of Solid-State Circuits*. 41, 3 (2006), 712–727.

[22] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters*. 10, 1 (2011), 16–19.

[23] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 135–146.

[24] Jack Sklansky. 1960. Conditional-Sum Addition Logic. *IRE Transactions on Electronic Computers*. 2 (1960), 226–231.

[25] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 531–543.

[26] Narayanan Sundaram, Nadathur Satish, Md M. A. Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya G. Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proceedings of the VLDB Endowment*. 8, 11 (2015), 1214–1225.

[27] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 425–440.

[28] Davis Tim. 2018. The University of Florida Sparse Matrix Collection. https://sparse.tamu.edu/.

[29] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 11:1–11:12.

[30] Yuxin Wang, Peng Li, Peng Zhang, Chen Zhang, and Jason Cong. 2013. Memory Partitioning for Multidimensional Arrays in High-Level Synthesis. In *Proceedings of the Annual Design Automation Conference (DAC)*. 12:1–12:8.

[31] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*. 408–421.

[32] Long Zheng, Xiaofei Liao, and Hai Jin. 2018. Efficient and Scalable Graph Parallel Processing With Symbolic Execution. *ACM Transactions on Architecture and Code Optimization*. 15, 1 (2018), 3:1–3:25.

[33] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2016. High-Throughput and Energy-Efficient Graph Processing on FPGA. In *Proceedings of IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 103–110.

[34] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on A Single Machine Using 2-Level Hierarchical Partitioning. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 375–386.