Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach

Mingyu Yan[1,2,3], Xing Hu[3], Shuangchen Li[3], Abanti Basak[3], Han Li[1,2], Xin Ma[3], Itir Akgun[3], Yujing Feng[1,2], Peng Gu[3], Lei Deng[3], Xiaochun Ye[1,*], Zhimin Zhang[1], Dongrui Fan[1,2] and Yuan Xie[3]
[1]SKLCA, ICT, CAS, Beijing, China, [2]UCAS, Beijing, China, [3]UC Santa Barbara, CA, USA

ABSTRACT

Graph analytics is an emerging application which extracts insights by processing large volumes of highly connected data, namely graphs. The parallel processing of graphs has been exploited at the algorithm level, which in turn incurs three irregularities onto computing and memory patterns that significantly hinder an efficient architecture design. Certain irregularities can be partially tackled by the prior domain-specific accelerator designs with well-designed scheduling of data access, while others remain unsolved.

Unlike prior efforts, we fully alleviate these irregularities at their origin—the data-dependent program behavior. To achieve this goal, we propose *GraphDynS*, a hardware/software co-design with *decoupled datapath* and *data-aware dynamic scheduling*. Aware of data dependencies extracted from the decoupled datapath, *GraphDynS* can elaborately schedule the program on-the-fly to maximize parallelism. To extract data dependencies at runtime, we propose a new programming model in synergy with a microarchitecture design that supports datapath decoupling. Through data dependency information, we present several data-aware strategies to dynamically schedule workloads, data accesses, and computations. Overall, *GraphDynS* achieves 4.4× speedup and 11.6× less energy on average with half the memory bandwidth compared to a state-of-the-art GPGPU-based solution. Compared to a state-of-the-art graph analytics accelerator, *GraphDynS* also achieves 1.9× speedup and 1.8× less energy on average using the same memory bandwidth.

CCS CONCEPTS

• **Hardware → Application specific integrated circuits**.

KEYWORDS

graph analytics, accelerator, software and hardware co-design.

ACM Reference Format:
Mingyu Yan[1,2,3], Xing Hu[3], Shuangchen Li[3], Abanti Basak[3], Han Li[1,2], Xin Ma[3], Itir Akgun[3], Yujing Feng[1,2], Peng Gu[3], Lei Deng[3], Xiaochun Ye[1,*], Zhimin Zhang[1], Dongrui Fan[1,2] and Yuan Xie[3]. 2019. Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358318

*Corresponding author, Email: yexiaochun@ict.ac.cn.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358318

1 INTRODUCTION

In the big data era, graphs are used as effective representations of data in many scenarios. Graph analytics can mine valuable insights from big data for a variety of applications, such as social network analysis [30, 35, 38], cybersecurity analysis [36], knowledge graphs [23], autonomous vehicles [47], brain science [8], web search [48], and robot navigation [19].

Although graph analytics inherently possesses a high-degree parallelism, it is difficult to achieve practical acceleration due to the irregularity challenges [16, 17, 32]. High-degree parallelism exists since millions of vertices and hundreds of millions of edges can be processed simultaneously via the popular Vertex-Centric Programming Model (VCPM) which is widely used by recent work [1, 21, 24, 34, 42]. However, due to the data-dependent program behavior [16, 29, 32] of graph algorithms, existing architectures face several challenges. Specifically, imbalanced workloads, large amounts of random memory accesses across diverse memory regions, and redundant computations result from the following three types of irregularities. ***(1) Workload irregularity***: The workloads among threads are significantly imbalanced [21, 42]. VCPM partitions the program into threads, where different active vertices are attributed to distinct threads. Consequently, various number of edges are processed by each thread, as each vertex is usually associated with a different number of edges in the graph [10, 40, 42, 46]. ***(2) Traversal irregularity***: Edges are traversed irregularly in each iteration [24, 25, 33, 39, 56]. This is caused by the different active vertices among iterations and the irregular connections between active vertices and their neighbors. The unpredictable traversal of the edges during each iteration leads to two critical difficulties. First, it introduces a large number of random memory accesses due to the poor locality. Second, it incurs long-latency atomic operations when multiple edges update the same vertex [33, 39, 41, 55]. ***(3) Update irregularity***: Both the update of the vertex property and the activation of vertex vary iteration-by-iteration [46]. Although few vertices are actually updated and activated, the program needs to check all of them. This update irregularity leads to a large amount of unnecessary computations and memory accesses.

Table 1: Our work solves all three types of irregularities.

Irregularity	GPU-based solutions	Graphicionado	**Our work**
Workload	Expensive preprocessing	✘ inefficient	✔
Traversal			✔
Update		✘	✔

Due to the aforementioned irregularities, GPU-based solutions suffer from workload imbalance, memory divergence, high synchronization overhead, and branch divergence[10, 26, 33, 40, 46, 54]. Most GPU-based solutions rely on preprocessing to tackle these irregularities [20, 26, 27, 33, 42, 52]. However, the preprocessing is costly. Unless multiple applications run on the same static graph repeatedly, the preprocessing overhead usually offsets its benefits. Recently, domain-specific hardware has been proposed to partially address the irregularity challenges of graph analytics. For example, Graphicionado [24], a state-of-the-art graph analytics accelerator, stores almost all the data, as long as it has a random access pattern, in a large on-chip scratchpad memory to mitigate the traversal irregularity. Therefore, it achieves 1.76-6.54× speedup while consuming 50-100× less energy compared to a state-of-the-art software graph analytics framework. However, as shown in Table 1, workload and update irregularities remain unsolved.

In fact, above irregularities are caused by the data-dependent program behavior. It raises a critical question in graph analytics: *could we schedule the program by considering the data dependency in order to tackle these irregularities?* Inspired by the question, this work introduces *GraphDynS*—a hardware/software co-design with ***decoupled datapath*** and ***data-aware dynamic scheduling***. *GraphDynS* can alleviate all three types of irregularities in graph analytics at its origin. **Decoupled datapath** is used to extract the data dependency in microarchitecture level. A *Dispatching/Processing* programming model is proposed to extract the workload size as well as the exact prefetching indication of the coming accessed data. Moreover, the execution flow is decoupled into two pipelined stages. Along with the proposed programming model, a microarchitecture design is also proposed to facilitate the decoupling of microarchitecture datapath. **Data-aware dynamic scheduling** is used to schedule the program on-the-fly by considering the data dependency. *To address the workload irregularity*, we dynamically dispatch workloads to processing elements in a balanced manner with knowledge of the precalculated workload sizes. *To mitigate the traversal irregularity*, we perform an exact prefetching to prefetch graph data with knowledge of the exact prefetching indication. Furthermore, we propose a store-reduce mechanism with a microarchitectural pipeline, which dynamically schedules the read-after-write data access for eliminating any stalls caused by atomicity. *As for the update irregularity*, we maintain a bitmap to record the ready-to-update vertices with the indication from the proposed pipeline. During the update, only the marked vertices are scheduled to compute.
Our contributions are summarized as follows:

- We propose *Dispatching/Processing* programming model and the accelerator architecture to decouple the microarchitecture datapath for data dependency extraction at runtime.
- We propose data-aware dynamic scheduling considering data dependencies, which elaborately schedules program on the fly, effectively tackling all three types of irregularities.
- We implement *GraphDynS* in RTL and evaluate it using a detailed microarchitectural simulation. Our comprehensive evaluations involve five well-known graph analytics algorithms with six large real-world graphs and five synthetic graphs. Compared to a state-of-the-art GPGPU-based solution, *GraphDynS* achieves 4.4× speedup and 11.6× less energy on average with half the memory bandwidth. Compared to a state-of-the-art graph analytics accelerator, *GraphDynS* achieves 1.9× speedup and 1.8× less energy on average with the same memory bandwidth.

2 BACKGROUND

In this section, we introduce common graph representation and programming models for graph analytics algorithms.

2.1 Graph Representation

Compressed sparse row (CSR) is an efficient and popular format for storing graphs, widely used by various software frameworks [22, 43, 49, 51] and graph accelerators [1, 5, 24, 39] due to its efficient usage of memory storage. As shown in Fig. 1, CSR format represents graphs by three one-dimensional arrays: offset, edge, and vertex property. The offset array stores the offset pointing to the start of the edge list for each vertex in the edge array. The edge array successively stores outgoing edges (i.e., neighbour IDs and weights in the case of weighted graphs) of all vertices. The vertex property array stores the property value of every vertex. The offset and vertex property arrays are indexed by the vertex ID, while the edge array is indexed by the offset.

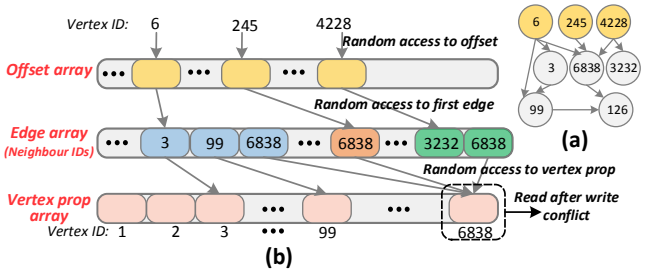

Figure 1: CSR representation. (a) Graph example. (b) CSR format and random data accesses.

2.2 Graph Programming Model

Deriving from the scattered structure of graphs, millions of vertices and ten millions edges are processed simultaneously and iteratively. In order to exploit the parallelism in graph analytics, Vertex-Centric Programming Model (VCPM) was first proposed by the Google Pregel system [34]. With advantages of simplicity, high scalability, and good expressiveness, VCPM has been widely adopted by various software frameworks [20, 21, 27, 31] and graph accelerators [1, 5, 24, 39, 56]. The push-based implementation of VCPM (PB-VCPM) is shown in Algorithm 1, which consists of two alternately running phases *Scatter* and *Apply*. First, in the *Scatter* phase, outgoing edges of each active vertex are traversed to update the temporary vertex property *v.tProp* of its destination vertex with application-defined *Process_Edge* and *Reduce* functions. Second, in the *Apply* phase, *Apply* function is executed with the constant vertex property *v.cProp* and *v.tProp*. Then, if the result of the *Apply* function *applyRes* is not equal to the vertex property *v.prop*, *v.prop* is updated and the vertex is activated. Finally, the above process executes iteratively until no more vertex is activated or a maximum number of iterations is reached. The application-defined functions in PB-VCPM of five well-known graph analytics algorithms, i.e., Breadth-First Search (BFS), Single Source Shortest Path (SSSP), Connected Components

(CC), Single Source Widest Path (SSWP), and PageRank (PR) are shown in Table 2.

Algorithm 1: ***Vertex-Centric Programming Model***

```
                                ◁ Scatter Phase
1  for each active vertex u do
2      offset ← OffsetArray[u.vid];
3      while EdgeArray[offset].src_vid == u.vid do
4          e(u,v) ← EdgeArray[offset++];
5          edgeProResult ← Process_Edge(u.prop, e.weight);
6          v.tProp ← Reduce(v.tProp, edgeProResult);
7      end
8  end
                                ◁ Apply Phase
9  for each vertex v do
10     applyRes ← Apply(v.prop, v.tProp, v.cProp);
11     if v.prop!=applyRes then
12         v.prop ← applyRes;
13         activate vertex with v.vid and v.prop;
14     end
15 end
```

Table 2: Application-defined functions. Edge *e=(u, v)*, source vertex *u* and destination vertex *v*.

Algorithm	Process_Edge	Reduce	Apply
BFS	$u.prop + 1$	$min(v.tProp, res)$	$min(v.prop, v.tProp)$
SSSP	$u.prop + e.weight$	$min(v.tProp, res)$	$min(v.prop, v.tProp)$
CC	$u.prop$	$min(v.tProp, res)$	$min(v.prop, v.tProp)$
SSWP	$min(u.prop, e.weight)$	$max(v.tProp, res)$	$max(v.prop, v.tProp)$
PR	$u.prop$	$v.tProp + res$	$(\alpha + \beta \cdot v.tProp)/v.deg$

Note: The res and $v.deg$ represent $edgeProResult$ and $v.cProp$ respetively. The α and β are constant.

3 MOTIVATION

In this section, we motivate our approach by identifying irregularities in graph analytics and limitations of prior work.

3.1 Challenges of Graph Analytics

Although VCPM helps exploit the parallelism of graph analytics, it incurs significant irregular computing and memory patterns, both of which raise the difficulty for designing efficient architecture. As mentioned above, the irregularities can be categorized into three classes: 1) Workload irregularity, 2) Traversal irregularity, and 3) Update irregularity.

Workload irregularity. The active vertices in each iteration usually have different degrees, leading to a varying number of edges to process. As shown in Fig. 2, the degree distribution of active vertices within an iteration can vary significantly. For example, in each iteration shown in Fig. 2, the degree of active vertices can vary from 1 to over 64. Since VCPM partitions and distributes workloads to distinct threads based on active vertices, the size of each workload varies significantly among different threads. Such workload irregularity can degrade GPU utilization by up to 25.3% - 39.4% for commonly used graph analytics algorithms [27, 42].

Traversal irregularity. Edges are traversed irregularly in each iteration due to the diverse connections in the graph. Pointer chasing during traversal introduces abundant random memory accesses, which not only raises challenges for efficient prefetching, but also possibly incurs read-after-write (RAW) conflicts. As shown in Fig. 1(b), random accesses include accesses to *offset*, *first edge*, and *vertex property*. Such random memory access results in a limited cache efficiency and higher bandwidth demands [1, 3, 4, 18, 46]. For example, the hit rate of L2 cache is only 10% for graph traversal workloads in CPU [4]. Moreover, since threads may share vertices, long-latency atomic operations are adopted to avoid thread contention when multiple threads modify the same vertex property. Previous study [33] shows that atomic operations cause a slow down of up to 32.15×.

Update irregularity. During runtime, vertices are irregularly updated and activated across iterations [46]. Although the number of vertices which are updated and activated may actually be few, all vertices need to be checked during the program execution. Therefore, unnecessary computation and memory accesses are introduced by such irregularity. As shown in Fig. 2 with the update line, 76% of iterations only update 10% of vertices. Up to 50% of iterations update less than 100 vertices while the graph has up to one million vertices. According to our evaluation, unnecessary memory access takes up to 50% of total memory accesses and results in 20% performance overhead in *Graphicionado* [24].

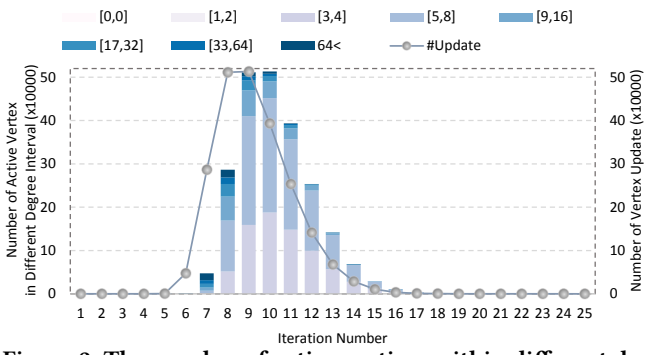

Figure 2: The number of active vertices within different degree intervals and the number of vertex updates on Flickr dataset as iteration goes (SSSP workload).

3.2 Limitation of State-of-the-Art

Graphicionado [24] partially addresses the traversal irregularity via an on-chip buffer to improve performance and energy efficiency. However, the overhead of atomics within the traversal irregularity, workload irregularity, and update irregularity have not yet been mentioned in their work. First, the workload irregularity leads to workload imbalance within pipeline. Because active vertices possess more than ten neighbors on average, the back-end of the pipeline has 10× more workload than the front-end. Moreover, hash-based (i.e., random) workloads allocated to different pipelines lead to only half of the pipelines experiencing workloads most of the time. Second, Graphicionado enforces atomicity by stalling the pipeline when contention is detected. These stalls cause up to 20% additional execution time. Third, update irregularity results in 20% additional execution time and 40% additional energy consumption. In fact, these irregularities result from the data dependent program behavior, which relies on intermediate results within and across iterations. Therefore, we propose a hardware/software co-design

to completely tackle these irregularities by data dependency aware scheduling on the fly.

4 GRAPHDYNS ARCHITECTURE

In light of the overhead introduced by the workload, traversal, and update irregularities, we propose a hardware/software co-design including optimized programming model, hardware accelerator, and data-aware dynamic scheduling strategies to address such issues.

As shown in Fig. 3(a), the optimized programming model allows *GraphDynS* to enable the hardware design datapath decoupling and to endow more visibility for the scheduling. Our hardware design helps to extract runtime data dependencies. The dynamic scheduling fully utilizes the knowledge of data dependency to schedule workloads, data accesses and updates.

4.1 Optimized Programming Model

We optimize the programming model based on VCPM for better practicality, since VCPM has been adopted by many recent graph processing frameworks and accelerator designs. Note that our programming model shares the same programmability with VCPM and our optimizations are transparent to users. The optimizations include: 1) determining the workload size during execution for scheduling a balanced workload, 2) obtaining the prefetching indication during execution for exact prefetching, and 3) decoupling the phases into two pipeline stages to overlap workload scheduling and execution.

4.1.1 Dynamically Determining Workload Size. To obviate the workload imbalance overhead introduced by workload irregularity, we optimize the programming model to export the workload size statistics. The key idea is to determine the number of edges for each active vertex by the offset array in the *Apply* phase, and use it in the *Scatter* phase of the next iteration.

The workload size of a processing element is dependent on the number of active vertices and the number of edges for each active vertex in *Scatter* phase, as well as the number of vertices in *Apply* phase. 1) Active vertex count and vertex count can be directly obtained in the original PB-VCPM, because active vertices are determined before each iteration and the vertex count is fixed after a dataset specified. 2) However, the number of edges for each active vertex (i.e., edge counter *edgeCnt*) is dependent on the vertex ID of the active vertex in each iteration and cannot be obtained in the original PB-VCPM. Therefore, we modify the programming model to acquire the offset array *OffsetArray* in *Apply* phase for *edgeCnt* calculation. As shown in Algorithm 2, we access the *offset* array sequentially in the processing stage of *Apply* phase. The *offset* of current vertex and succeeding vertex are used to calculate the *edgeCnt* of every active vertex. $v.prop$, $offset$ and *edgeCnt* are used to activate a vertex, which constitutes active vertex data. This way, we dynamically acquire the workload size that can be used as hints for scheduling.

4.1.2 Acquiring Exact Edge-Prefetch Indication. After quantitative analysis towards the graph applications, we observe that many of active vertices only possess 4-8 edges, as shown in Fig. 2, which is smaller than one cacheline size (64bytes). Therefore, accesses to edge lists have limited data locality and become the new bottleneck after the random access to vertex property issue has been solved. To address this issue, we propose an exact prefetching technique. Exact prefetching means that only the necessary data will be prefetched given the deterministic prefetching address and amount to prefetch. Exact prefetching helps to maximize the number of in-flight memory requests in order to utilize the memory bandwidth more efficiently and hide memory latency. All the sequentially accessed data such as active vertex data and vertex data can be prefetched exactly, since the data address and the amount to prefetch are available before these data are required.

To prefetch edge data exactly, we need the exact prefetching indication including *offset* and *edgeCnt* for all active vertices. As mentioned in Sec. 4.1.1, we have acquired *offset* and *edgeCnt* for all active vertices from the programming model. More details about exact edge prefetching hardware implementation are shown in Sec. 5.2.1.

Algorithm 2: ***Optimized Programming Model***

```
◃ Dispatching Stage of Scatter Phase
for each active vertex u do
    dispatch(u.prop, u.offset, u.edgeCnt) to PE;
end
◃ Processing Stage of Scatter Phase
for e(u,v) ←EdgeArray[u.offset : u.offset + u.edgeCnt] do
    edgeProResult ← Process_Edge(u.prop, e.weight);
    v.tProp ← Reduce(v.tProp, edgeProResult);
end
◃ Dispathching Stage of Apply Phase
for each vertex list do
    dispatch(start id vListStartID and size vListSize of vertex
     list) to PE;
end
◃ Processing Stage of Apply Phase
for vid ← vListStartID : vListStartID + vListSize do
    edgeCnt ← OffsetArray[vid+1] - OffsetArray[vid];
    applyRes ← Apply(v_vid.prop, v_vid.tProp, v_vid.cProp);
    if v_vid.prop != applyRes then
        v_vid.prop ← applyRes;
        activate v_vid with v_vid.prop, OffsetArray[vid],
         edgeCnt;
    end
end
```

4.1.3 Decoupling Execution Flow. To overlap the latency of work scheduling and execution, we decouple *Scatter* and *Apply* phases into *Dispatching* and *Processing* stages as shown in Algorithm 2.

Dispatching Stage: Workloads are dispatched to processing elements (*PEs*). In the *Scatter* phase, edge workload of each active vertex is dispatched to *PEs*. In the *Apply* phase, vertex workload is dispatched to every *PE*.

Processing Stage: *PEs* process the workload. In the *Scatter* phase, *PEs* process edge workload by executing *Process_Edge* and *Reduce*

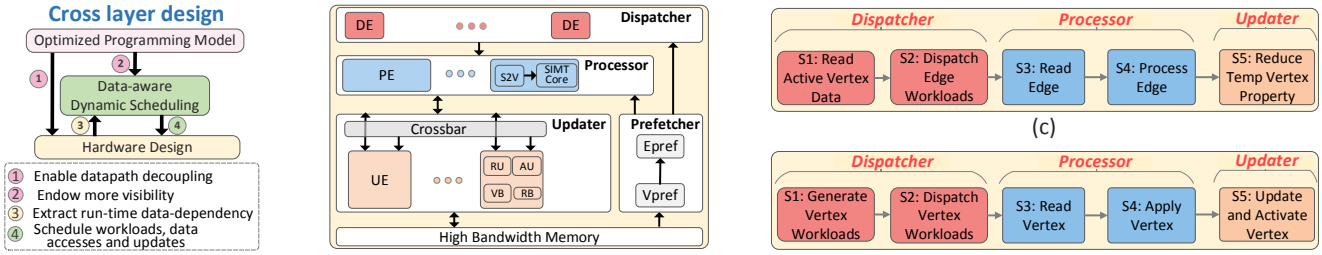

Figure 3: *GraphDynS* architecture overview. (a) Cross-layer design, (b) hardware design, hardware-platform stages in (c) *Scatter* phase and (d) *Apply* phase.

functions. In the *Apply* phase, *PEs* process vertex workload by executing *Apply* function and activating vertex for next iteration.

4.2 Hardware Design

In conjunction with the proposed programming model, we propose *GraphDynS* hardware with several components to decouple the datapath in microarchitecture.

4.2.1 Hardware Components. As shown in Fig. 3(b), *GraphDynS* consists of following 4 components:

Dispatcher: *Dispatcher* dispatches workload to *Processor* and consists of 16 *Dispatching Elements (DEs)*, which are simple cores.

Processor: *Processor* receives the workload from *Dispatcher* and processes the workloads. *Processor* consists of 16 *PEs*. Each *PE* consists of a *Scalar to Vector Unit (S2V)* and a SIMT (8 lanes) core. *S2V* unit is used to transform workload from a scalar to a *SIMT* vector. We construct the *PE* using single-precision floating point adders, multipliers, and comparators.

Prefetcher: *Prefetcher* prefetches graph data from High Bandwidth Memory (HBM), and consists of *Vertex Prefetcher (Vpref)* and *Edge Prefetcher (Epref)*. *Vpref* prefetches active vertex and vertex data to vertex prefetching buffer (*VPB*). *Epref* prefetches edge data to edge prefetching buffer (*EPB*).

Updater: *Updater* receives the stored result from *Processor* and updates the vertex property or activates vertex, which is composed of a 128-radix crossbar switch and 128 *Updating Elements (UE)*. *UE* consists of a *Vertex Buffer* (*VB*, 256 KB dual-ported on-chip eDRAM), a Ready-to-Update Bitmap (*RB*, 256 entries), a *Reducing Unit (RU)*, and an *Activating Unit (AU)*. *VB* is used to cache all temporary vertex property data. To process larger graphs (i.e., *VB* cannot hold all temporary vertex property), the graph is sliced into several slices and a single slice is processed at a time with the slicing technique proposed in *Graphicionado* [24]. *RB* is used to indicate the ready-to-update vertex. *RU* is used to execute *Reduce* function, including a microarchitecture pipeline called *Reduce Pipeline*. *AU* is used to activate vertices, and consists of four 16-entry buffer queues to store active vertices.

4.2.2 Hardware-Platform Stage. According to the functionality of aforementioned components, the hardware-platform stage of *Scatter phase* and *Apply phase* are described as follows:

Scatter phase (Fig. 3(c)): First, *DE* reads active vertex data from *VPB* in step S1 and then dispatches edge workloads (i.e., edge list) to *PEs* in step S2. Next, *PE* reads edges from *EPB* in step S3 and then executes *Process_Edge* function to process edges in step S4. Finally, *UE* reads temporary vertex property from *VB*, executes *Reduce* and writes result to *VB* in step S5.

Apply phase (Fig. 3(d)): First, *DE* generates a vertex index list as vertex workloads in step S1, and then dispatches vertex workloads to *PE* in step S2. Next, *PE* reads vertex data from *VPB* in step S3 and then executes *Apply* function to process vertex in step S4. Finally, *UE* updates vertex property and activates vertex for next iteration in step S5.

4.2.3 Decoupling Datapath. To acquire the information at runtime, we decouple the datapath into the following three segments based on the optimized programming model and aforementioned components.

Workload Management Sub-Datapath: dispatching and processing workloads. As shown in Fig. 3(b), *DE* dispatches workloads to *PE* and the *S2V* unit transforms workloads to vector workloads with a loop unrolling fashion. To alleviate workload irregularity, we dispatch workloads to every *PE* in a balanced way and extend workload processing with SIMT execution model to improve workload processing throughput using information of workload size, which is discussed in Sec. 5.1 in detail.

Data Access Sub-Datapath: off-chip memory access and on-chip memory access. As shown in Fig. 3(b), *Prefetcher* exactly prefetches graph data from off-chip memory, *Processor* and *Dispatcher* coalesce access data from on-chip memory (i.e., *VPB*, *EPB*, and *VB*), and *RU* accesses *VB* atomically. To alleviate traversal irregularity, we schedule data access using data type information, access size and access address to reduce random accesses, improve data access throughput and remove atomic stalls, which is discussed in Sec. 5.2 in detail.

Data Update Sub-Datapath: data update includes vertex property update and vertex activation. As shown in Fig. 3(b), in the *Scatter* phase, *RU* writes *RB* to mark vertex as ready-to-update if its temporary vertex property is modified. In the *Apply* phase, *Prefetcher* prefetches vertex data and informs the *Dispatcher* to dispatch vertex workloads to *PE* for update computation. Next, *AU* updates vertex property and activates vertex. To alleviate update irregularity, we dynamically select ready-to-update vertices using their modification status, and then prefetch their data and process them. Moreover, we coalesce stores to off-chip memory of active vertices given updated computation results, which is discussed in Sec. 5.3 in detail.

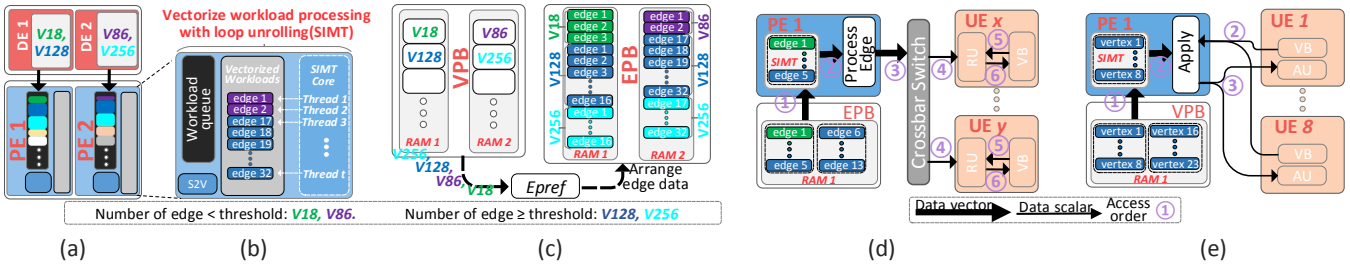

Figure 4: Data-aware Dynamic Scheduling. (a) *DE*s dispatch workloads to *PE*s with a threshold. (b) *S2V* unit vectorizes workload processing with loop unrolling. (c) Data organization of *VPB* and *EPB*. On-chip vectorized access procedure of *PE* in (d) *Scatter* phase and (e) *Apply* phase.

5 DATA-AWARE DYNAMIC SCHEDULING

Leveraging the run-time information from the optimized programming model and decoupled datapath, we present a data-aware dynamic scheduling scheme to schedule workloads, data accesses, and update computations based on the proposed architecture. In this section, we show how the proposed techniques tackle the workload, traversal, and update irregularities.

5.1 Workload Scheduling

We first balance the workload given the workload size, then explain how to improve workload processing throughput based on the *SIMT* execution model.

5.1.1 Workload-Balanced Dispatch. To achieve workload balance, we dynamically dispatch workloads to each *PE* according to the workload size. Different from *Graphicionado*, our design processes the edge list of low-degree vertices in the *PE* in batches, and distributes the edge list of active vertices with high degrees to each *PE* evenly. In this way, the number of scheduling operations are reduced significantly, and the workloads are balanced in each *PE*. Next, we describe the scheduling flow:

Scatter phase: *DE* reads active vertex data from *VPB* and then dispatches the edge list of this active vertex to the *PE* according to *edgeCnt*. *DE* dispatches the entire processing workload of the edge list to the *PE* with the same number (i.e., DE_i sends to PE_i) if *edgeCnt* is smaller than a predefined threshold *eThreshold*. Otherwise, *DE* partitions the edge list into several even sub edge lists (the size of each sub edge list is *eThreshold*) and then dispatches them to every *PE*. For example, as shown in Fig. 4(a), the *edgeCnt* of active vertices ***V18***(3 edges) and ***V86***(2 edges) are smaller than *eThreshold* (i.e., 16) while ***V128***(32 edges) and ***V256***(32 edges) are larger than *eThreshold*. Thus, DE_1 dispatches the entire workload of ***V18*** to PE_1 while splitting the edge list of ***V128*** into two even sub edge lists and then dispatches them to every *PE*. *PE* saves these lists in a workload queue before processing them.

Apply phase: *DE* generates a vertex list (size of each vertex list is *vListSize*) and dispatches processing workload of the vertex list to *PE* with the same number (i.e., DE_i sends to PE_i). The vertex list generated by DE_i starts from $i \times vListSize$ with stride *number of DE* $\times$ *vListSize*.

5.1.2 Workload-Processing Vectorization. To improve workload processing throughput, we extend the execution unit of *PE* with the SIMT execution model. We use *S2V* unit to execute loop unrolling on the workload list (i.e., edge or vertex list) so that the workload is transformed from a scalar to vector pattern. As shown in Fig. 4(b), *S2V* unit in PE_2 gets edge lists of ***V86*** and ***V128*** from the workload queue and assigns an edge to each *SIMT* thread simultaneously. Next, *SIMT* core executes these workloads together. In addition, we also combine the small workload list which is smaller than the number of threads of SIMT core together to improve *SIMT* efficiency. Not only does this optimization improve processing throughput, but it also improves the computation efficiency.

5.1.3 Configuration for Dispatch and Vectorization. To improve the workload dispatching and processing efficiency, we need to appropriately set the value of the number of SIMT threads (*nSIMT*), the number of sub edge list size(*eListSize*), *eThreshold*, and *vListSize*. As shown in Fig. 2, more than 60% of active vertices have more than 5 neighbors over iterations and up to 99% of active vertices have more than 3 neighbors over iterations. Moreover, active vertices have more than 10 neighbors on average. So, considering the efficiency of *SIMT* and degree distribution, we set *nSIMT* as 8. To reduce the complexity of *Dispatcher* and workload imbalance due to high-degree active vertices, we set *eThreshold* as 128. Considering the data access granularity of *PE* and latency of *EPB*, we set *eListSize* as 16. To simplify the access to *VB*, we set *vListSize* as 8, which is discussed in Sec. 5.2.2.

5.2 Data Access Scheduling

In this section, we first propose exact edge prefetching hardware implementation, and then vectorize the data access to on-chip memory. Finally, we propose a zero-stall atomic maintenance mechanism with a customized microarchitecture pipeline.

5.2.1 Exact Edge Prefetching Implementation. After random accesses to vertex property data have been solved by the on-chip memory in *Graphicionado*, accesses to poor-locality edge list become the new bottleneck as mentioned in Sec. 4.1.2. With the exact edge prefetching indication from our programming model, we design an exact *Prefetcher*, which prefetches other sequentially accessed data too.

Exact prefetching the edge list of all active vertices is executed as following. Step **1**, *Vpref* uses the base address of *active vertex array* and the *number of active vertices* to send active vertex access requests to HBM. Step **2**, *Vpref* receives *offset* and *edgeCnt*. Step **3**, *Vpref* sends *offset* and *edgeCnt* to *Epref*. Step **4**, *Epref* uses *offset* and *edgeCnt* to send edge access request to HBM. Step **5**, *Epref* receives edge data into *EPB*.

Previous work introduced long latency to start edge prefetching or a large on-chip memory caching *offset* array to reduce the latency, since it needs a random access to start location *offset* of edge list with active vertex ID [1, 24]. In addition, it costs additional memory storage as well as abundant unnecessary memory accesses for source vertex ID, src_vid. In previous design, every edge data includes src_vid, which is compared against the active vertex ID $u.vid$ in order to determine the end of the edge list traversal for an active vertex. As a result, each edge traversal requires access to extra elements, thus wasting up to half of the bandwidth.

On the contrary, our design begins prefetching as soon as possible, since the accesses to offset in our work do not rely on active vertex ID and the offset can be prefetched. The *Prefetcher* gets the exact number of edges (i.e, *edgeCnt*) to prefetch, so that it can efficiently prefetch edges without causing unnecessary data accesses and additional memory storage. Moreover, since an off-chip memory request acquires a significant amount of active vertex data, our work can use these offsets and edge counters to coalesce memory accesses to edge data and maximize the number of in-flight memory requests. Consequently, our optimization enables better utilization of the bandwidth.

5.2.2 Vectorizing On-Chip Data Access. To improve on-chip memory access efficiency and throughput, we vectorize the on-chip memory access. We first organize the data in *Prefetching Buffer* with *SIMT* granularity and then arrange the data access to *Vertex Buffer*.

Prefetching buffer organization: Both *VPB* and *EPB* have 16 RAMs. The vertex data placement are based on a hash algorithm (i.e., RAM id = *accessing order % RAM number*). Each RAM of *VPB* and *EPB* only can be accessed by specific *DE* or *PE* (i.e., DE_i and PE_i only access RAM_i).

Since the size of edge list for each active vertex varies, it is neccessary to maintain the processing edge vector in *PE* corresponding to edge data vector in *EPB*. Therefore, *Epref* adopts the same workload-balance strategy of *DE* to arrange the edge data in *EPB*. We illustrate an example of on-chip data organization in *VPB* and *EPB*, as shown in Fig. 4(c). The RAM_1 of *VPB* and RAM_1 of *EPB* are connected to PE_1, while the RAM_2 of *VPB* and RAM_2 of *EPB* are connected to PE_2. Four active vertices (***V18***, ***V86***, ***V128***, ***V256***), are placed into RAM of *VPB* dependent on their access order. Then, the edge workload of these four active vertices will be dispatched to the corresponding PE according to the dispatch strategies in Section 5.1.1. As shown in Fig. 4(a), the edge workload of ***V18*** and partial workload of ***V128*** are assigned to PE_1. In the meantime, *Epref* sequentially reads active vertex data of ***V18***, ***V86***, ***V128***, ***V256*** in *VPB* for edge data prefetching. After receiving the prefetched edge data, *Epref* places the edge data of active vertices to the corresponding RAM according to *edgeCnt* and *eThreshold*. For example, as same as workload dispatching flow, ***V18***'s edge data are stored in RAM_1 of *EPB* while ***V128***'s edge data are stored in RAM_1 and RAM_2 evenly. In this way, PE_1 read edge data of ***V18*** and partial of ***V128*** in order with their corresponding workload. Note that PE_i can access *nSIMT* data from RAM_i each time.

VB data access arrangement (Scatter phase): To improve *VB* access throughput in *Scatter* phase, we split the vertex temporary property data evenly into 128 partitions according to a hash algorithm and place them in 128 *VBs* respectively (i.e., *VB* ID = Vertex ID % *number of UE*). Moreover, we use a crossbar switch to route the access of each thread to different *VBs* by access address. Fig. 4(d) illustrates the on-chip memory access procedure of PE_1 in *Scatter* phase. Step **1**, PE_1 reads an edge vector from RAM_1 of *EPB* each time. Step **2**, PE_1 processes the edge vector by *Processing_Edge* function. Step **3**, PE_1 sends the edge processing result vector to *Updater*. Step **4**, *Updater* uses a crossbar switch to route the edge processing result from each thread to different *UEs* by the destination vertex temporary property store address. Step **5**, *RU* reads the destination vertex temporary property from *VB* and executes *Reduce* function. Step **6**, *RU* writes the reduced result to *VB*. The details of data accesses inside the *Updater* are explained in Sec. 5.2.3. By arranging accesses to *VB* as explained, complex logic design is not needed to move data from *VB* to *PE* and to transform the scalar data to the vector data.

VB data access arrangement (Apply phase) To simplify the accesses to the *VB* and improve *VB* access throughput in *Apply* phase, each *PE* only accesses *nSIMT* with consecutive *VBs*. Since the vertex IDs within the vertex list are consecutive and generated by *DE*, each vector access from *PE* can directly accesses *nSIMT* consecutive *VBs*. Moreover, the vertex lists are generated by the same *DE* with a stride (i.e., $number\ of\ DE \times vListSize$ as mentioned before). Thus, the access of *PEs* can be distributed to *nSIMT* with specific *VBs*. Hence, conflict arising from accessing *VBs* can be avoided. *PE* just receives the data from *nSIMT* registers, since both access and data return of the same *UE* occur at the same time and the access latency is constant. Fig. 4(e) illustrates the on-chip memory access procedure of PE_1 in *Apply* phase. Step **1**, PE_1 reads *nSIMT* vertex vector from RAM_1 of *VPB* each time. Step **2**, PE_1 reads the temporary vertex property vector from *VB* of $UE_1 - UE_8$, and PE_1 uses it and the vertex vector to execute *Apply* function. Step **3**, PE_1 sends the result vector to *Updater* and then *Updater* sends result of each thread to *AU* of $UE_1 - UE_8$, respectively. The details of data access inside the *Updater* are explained in Sec. 5.3.2. By arranging the access of *VB* as explained, *PE* can use simple logic to transform scalar data to vector data.

5.2.3 Maintaining Atomicity with Zero Stall. Previous work[55] shows that atomic operation is another major bottleneck due to the parallel data conflict on scattered feature of graph processing, especially for graphs with power-law degree distribution. In addition, the atomic bottleneck in complete edge access algorithms (e.g. PageRank) becomes more significant when other issues are addressed. It introduces many stalls during access to the temporary vertex property, causing heavy performance degradation.

Opportunity: Fortunately, we find an opportunity to eliminate stalls from atomic operations based on the following observations: 1) Read-after-write (RAW) only occurs in *Reduce* function, where *v.tProp* is modified, 2) Each algorithm only has one specified *Reduce* function, and 3) The *Reduce* function of most graph workloads is a simple compare or accumulation operation (i.e, only one instruction and few instruction operation types)[21, 39].

Key idea: Based on these observations, we propose a store-reduce mechanism with a microarchitecture pipeline. The key idea

is to shorten the distance between RAW conflict data and the arithmetic unit. Next, up-to-date data is selected for executing in the arithmetic unit.

Specially, we first shorten the distance between the compute pipeline and data to five cycles by the store-reduce mechanism. Next, we customize the compute pipeline to further shorten the distance to one cycle. Finally, we select up-to-date data by comparing the access address between a source operand and the destination operand, and send the data to the arithmetic unit inside the microarchitecture pipeline.

Implementation: We first transform read, reduce and write operations of *v.tProp* to a store operation to *VB* and then execute *Reduce* function in *RU* (namely store-reduce). Each *PE* executes the store operation to send the store address of *v.tProp* and the edge processing result *edgeProResult* to *Updater*. The crossbar switch inside the *Updater* routes the data in each thread to corresponding *UE* based on *v.tProp* store address. Next, *RU* reads the *v.tProp* from *VB*, executes *Reduce* function, and writes the *v.tProp* to *VB*.

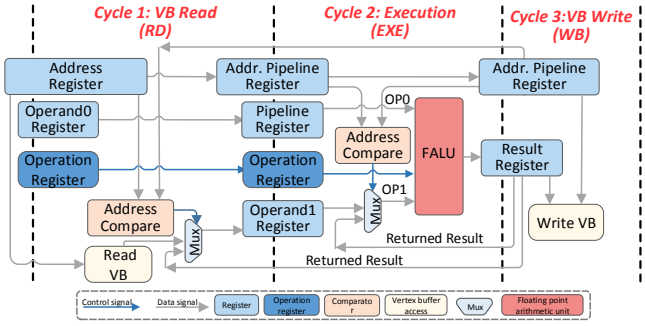

Figure 5: *Reduce Pipeline.*

To avoid pipeline stalls inside *RU*, we present a three-stage pipeline *Reduce Pipeline* to execute the *Reduce* function. As shown in Fig. 5, we remove the instruction fetching and instruction decode stages of the classic five-stage pipeline since operation is decided after graph algorithm is specified. ***1) RD Stage,*** the old *v.tProp* is read from *VB* with the address of *v.tProp*. Whether the read *v.tProp* or the result returned from *WB Stage* will be set as *Operand 1 Register* is decided by an address comparison of addresses from the *RD Stage* and *WB Stage*. The returned result is set as *Operand 1 Register* if these two addresses are equal. Otherwise, the read *v.tProp* is set as *Operand 1 Register*. The address is pipelined next to *EXE Stage*. ***2) EXE Stage,*** the floating point arithmetic logical unit (*FALU*) calculates the result. Similar to *RD Stage*, the Operand 1 of *FALU* is also determined by address comparison. The returned result from *WB Stage* is sent to *FALU* if these two address are equal. Otherwise, the value of *Operand 1 Register* is sent to *FALU*. The address is forwarded next to *WB Stage*. The execution stage only consumes one cycle due to the simplicity of *Reduce Operation*. ***3) WB Stage,*** the new *v.tProp* is written into *VB*. The result and address are returned to *RD Stage* and *EXE Stage*. We integrate *Reduce Pipeline* into every *RU* so that the stalls are removed while atomicity is maintained.

5.3 Data Update Scheduling

During *Apply* phase, it is not necessary to update all the vertices. To avoid unnecessary updates, we select the vertices that require an update with the indication from *RU*. And then, to reduce irregular (intermittent) memory accesses due to random updates, we coalesce these intermittent accesses to effectively use the bandwidth.

5.3.1 Selecting Ready-to-Update Data. We next show the opportunity and implementation that selects ready-to-update data.

Opportunity: The result of *Reduce* function can indicate whether a vertex is ready to update for most algorithms. Therefore, it poses an opportunity to select out the vertices that require an update via the result of *RU*. As shown in Algorithm 2, *Apply* function uses the result of *Reduce* function whose modification makes *applyRes* change in most algorithms. It indicates that the vertex is ready to update. With the indication from *RU*, we maintain a bitmap (namely, Ready-to-Update Bitmap, i.e, *RB*) to record vertices that are ready to update in *Scatter* phase. Next, we only update these ready-to-update vertices in *Apply* phase. This way, the computation and memory access for unnecessary updates are eliminated.

Implementation: We integrate *RB* into each *UE* to record *v.tProp* whose vertex has been modified in *Scatter* phase of current iteration. In the *Apply* phase, *Vpref* only prefetches the ready-to-update vertex data and *DE* only generates ready-to-update vertex workload based on the bit value in *RB*. To determine the bitmap size, the trade-off between unnecessary computation and the bitmap implementation overhead needs to be considered. To simplify the hardware implementation and to leverage the *SIMT* advantages, we use 1 bit to represent the ready status of 256 consecutive vertices. Experimental results show that it is effective to reduce the unnecessary computations.

5.3.2 Coalescing Intermittent Accesses. To alleviate the memory bandwidth pressure resulting from random updates, we coalesce the intermittent off-chip memory accesses to the active vertex array.

Opportunity: Whether or not the active vertex array needs to be updated is determined by a data-dependent condition shown in Algorithm 2 lines 14-17, which introduces intermittent accesses to off-chip memory leading to inefficient bandwidth utilization. Since there is only a single path in this branch, we can transform this path into a conditional store operation. Next, each *UE* uses two sets of buffer queues working in a double buffer fashion to buffer active vertex data and write them to off-chip memory when the buffer queue is full or at the end of *Apply* phase. Thus the stall possibility incurred by the out of buffer queue is reduced and the off-chip memory access efficiency is improved.

Implementation: The store operation executed in *PE* sends the result of conditional branch (i.e., condition flag), *applyRes*, *offset* and *edgeCnt* to *UE*. The *AU* in the *UE* activates vertex if *condition flag* is true, and then writes the active vertex data to the buffer queue. Vertex properties are stored in off-chip memory together whether *condition flag* is true or not to reduce random accesses.

5.4 Summary

To summarize, in this work we first extract the data dependencies at runtime with decoupling the datapath with our optimized programming model and hardware design. Next, we dynamically schedule balanced workloads and data accesses elaborately, and eliminate unnecessary updates through our data dependency aware scheme.

In particular, to mitigate *Workload Irregularity*, *Dispatcher* leverages the workload size information to balance workload across

multiple *PEs* and *Processor* utilizes SIMT execution model to process workloads in vector. To address *Traversal Irregularity*, *Prefetcher* leverages the offset and length of various data structure to exactly prefetch graph data and to vectorize on-chip memory access. Moreover, *Updater* leverages the access address of vertex property to maintain atomicity with zero stall based on the observation of the simplicity of reduce operations. To address *Update Irregularity*, *Updater* utilizes the modification status of temporary vertex property to implement selective updates and coalesces the intermittent off-chip access of active vertex data.

6 EXPERIMENTAL METHODOLOGY

Methodology. The performance and energy of *GraphDynS* and *Graphicionado* are measured by the following tools.

Accelerator simulator. For the performance evaluation, a customized cycle-accurate simulator was designed and implemented to measure execution time in number of cycles. This simulator models the microarchitectural behavior of each hardware module of our design. In addition, the performance model implements a detailed cycle-accurate scratchpad memory model and a crossbar switch model. This is also integrated with Ramulator [28], a memory simulator, to simulate the cycle-accurate behavior of the memory accesses to HBM.

CAD tools. For area, power and critical path delay (in cycles) measurements, we implemented a Verilog version of each hardware module, then synthesized it. We used the Synopsys Design Compiler with the TSMC 16nm standard VT library for the synthesis, and estimated the power consumption using Synopsys PrimeTime PX.

eDRAM, crossbar switch and HBM measurements. The area, power and access latency for the on-chip scratchpad memory were estimated using Cacti 6.5 [53]. Since Cacti only supports down to 32nm technology, we apply four different scaling factors to convert them to 16nm technology as in [46]. The area, power and throughput for the crossbar switch were estimated using the wire length, routing resources and physical parameters of the metal layers with the model in [9]. Similarly, we also apply the four different scaling factors mentioned above to convert them to 16nm technology. The energy for HBM 1.0 was estimated using 7 pJ/bit as in [44].

Baselines. To compare the performance and energy efficiency of *GraphDynS* with state-of-the-art work, we evaluate *Graphicionado* and a GPU-based solution *Gunrock* [52] in a Linux workstation equipped with an Intel Xeon E5-2698 v4 CPU, 256GB memory, and an NVIDIA V100 GPU. Table 3 shows the system configurations for above implementations.

Table 3: System used for *GraphDynS* and baselines.

	GraphDynS	Graphicionado	Gunrock (V100)
Compute Unit	1Ghz 16×SIMT8	1Ghz 128×Streams	1.25Ghz 5120×cores
On-chip memory	32MB eDRAM	64MB eDRAM	34MB
Off-chip memory	512GB/s HBM 1.0	512GB/s HBM 1.0	900GB/s HBM 2.0

Note: GPU's on-chip memory consists of register file, shared memory and L2 cache.

Graph datasets and algorithms. Table 4 describes the graph datasets used for our evaluation. Both real-world graphs and synthetic graphs are used for the evaluation. We use SSSP, CC, BFS, SSWP, and PR algorithms to evaluate *GraphDynS*. For the evaluation on unweighted real-world graphs, random integer weights between 0 and 255 were assigned.

Table 4: Graph datasets used in evaluation.

Graph	#Vertices	#Edges	Brief Explanation
Flickr (FR)[15]	0.82M	9.84M	Flickr Crawl
Pokec (PK)[15]	1.63M	30.62M	Pokec Social Network
LiveJournal (LJ)[15]	4.84M	68.99M	LiveJournal Follower
Hollywood (HO)[15]	1.14M	113.90M	Movie Actors Social
Indochina-04 (IN)[15]	7.41M	194.11M	Crawl of Indochina
Orkut(OR)[15]	3.07M	234.37M	Orkut Social Network
RMAT scale 22 (RM22)[37]	4.19M	67.11M	Synthetic Graph
RMAT scale 23 (RM23)[37]	8.38M	134.22M	Synthetic Graph
RMAT scale 24 (RM24)[37]	16.76M	268.44 M	Synthetic Graph
RMAT scale 25 (RM25)[37]	33.52M	536.88M	Synthetic Graph
RMAT scale 26 (RM26)[37]	67.04M	1073.76M	Synthetic Graph

7 EXPERIMENTAL RESULTS

In this section, we compare GraphDynS to baselines and provide analysis.

• ***Speedup:*** We first compare the performance speedup normalized to *Gunrock*, as shown in Fig. 6. The last set of bars, labeled as GM, indicate the geometric mean across all algorithms. Overall, *GraphDynS* achieves 4.4× speedup over *Gunrock* and only possesses half the off-chip memory bandwidth. Meanwhile, compared to *Graphicionado*, *GraphDynS* achieves 1.9× speedup and only consumes 50% of on-chip memory. The better performance of GraphDynS over others comes from highly effective memory bandwidth utilization, elimination of atomic stalls, and the reduction of update operations.

In the further step, we analyze the performance improvement on different applications. Compared to *Gunrock*, the speedup of CC algorithm is smaller than other algorithms since the online preprocessing of *Gunrock* efficiently reduces unnecessary workloads by filtering unnecessary active vertices [52]. On the other hand, PR algorithm achieves higher speedup on *GraphDynS* and *Graphicionado* due to fewer random accesses to off-chip memory.

Compared to *Graphicionado*, *GraphDynS* achieves higher speedup on PR algorithm than other algorithms. The reasons are 1) exact prefetching that eliminates the redundant data access that harvests more bandwidth to improve throughput, and 2) zero-stall atomic optimization that eliminates the stall caused by frequent RAW conflicts introduced by high-throughput graph computing. For the other algorithms, the performance speedup is lower than PR. This is due to the long latency introduced by non-sequential accesses to the first edge of active vertices. Furthermore, most of the active vertices have an edge list smaller than off-chip memory access granularity, leading to underutilized bandwidth. We also analyze the influence of dataset and observe that speedup on HO dataset is higher than other datasets. This is because HO dataset has high edge-to-vertex ratio. Edge data possesses high spatial locality and thus access latency can be hidden.

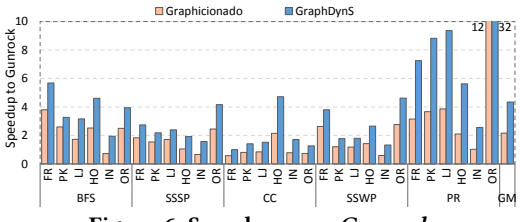

Figure 6: Speedup over *Gunrock*.

• ***Throughput:*** Additionally, we compare the throughput of these three techniques, as shown in Fig. 7. Throughput is defined as the number of edges processed per second (GTEPS, i.e., giga-traversed

edges per second). The ideal peak throughput is 128 GTEPS. *GraphDynS* achieves 43 GTEPS on average, while *Graphicionado* and *Gunrock* achieve 21 GTEPS and 8 GTEPS respectively. Although *GraphDynS* achieves 5× throughput over *Gunrock* on average, its average speedup over *Gunrock* is 4.35×, since online preprocessing dominates execution time of *Gunrock* and fewer edges are processed compared to *GraphDynS*.

Due to atomic operation stalls and low data access throughput of prefetch buffer, PR algorithm in *Graphicionado* only achieves 37.5 GTEPS throughput on average. With the optimization of zero-stall atomics and prefetch buffer organization, PR algorithm in *GraphDynS* achieves throughput of 87.5 GTEPS on average. Although all the edges on PR algorithm are processed iteratively, PR algorithm does not achieve peak throughput since DRAM refresh and memory accesses to vertex data consumes bandwidth. For the other algorithms, CC achieves higher throughput than SSSP, SSWP and BFS on average since it starts from all vertices (i.e., all vertices in first iteration are active vertices) and converges quickly.

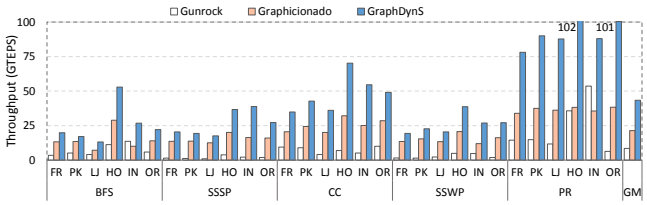

Figure 7: Throughput.

• ***Power/Area:*** Fig. 8 shows the power and area breakdown of *GraphDynS* except for HBM. The total power and area of *GraphDynS* are 3.38 W and 12.08 mm^2 respectively. *Dispatcher* and *Prefetcher* only cost 5% power as well as 2% area. Meanwhile *Processor* costs 59% power as well as 8% area due to its 128 processing elements. *Updater* costs 36% power as well as 90% area due to its 32MB eDRAM and crossbar switch. However, *Graphicionado* requires 64MB eDRAM to cache *temporary vertex property* array and *offset* array, which is 50% larger than our design. Hence, although *GraphDynS* is composed of 128 reducing pipelines and 128 active units, its power and area are still less than *Graphicionado*, only 68% and 57% of *Graphicionado*.

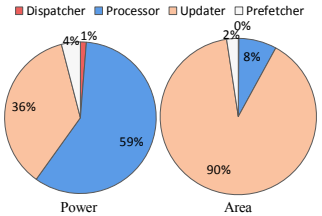

Figure 8: Power and area breakdown.

• ***Energy:*** Fig. 9 shows the normalized energy consumption of *GraphDynS* and the baselines (including energy consumption of HBM), normalized to *Gunrock*. *GraphDynS* reduces energy consumption by 91.4% compared to *Gunrock* on average. It also reduces energy consumption by 45% on average. The main reason is the reduction of a large number of memory accesses due to exact prefetching and unnecessary updates. Compared to *Gunrock*, CC algorithm running on FR dataset and PR algorithm on IN dataset are less energy efficient than other cases on *GraphDynS* since these two cases introduce many off-chip memory accesses.

As shown in Fig. 10, 7.8% of total energy is consumed on average by the components of *GraphDynS* while 92.2% of energy is consumed by off-chip memory access (i.e., HBM). The reason is that these graph algorithms have extremely low computation-to-communication ratio. The *Processor* consumes 4.0% of energy while the *Updater* consumes 3.0%. The rest of the components consume less than 0.8%.

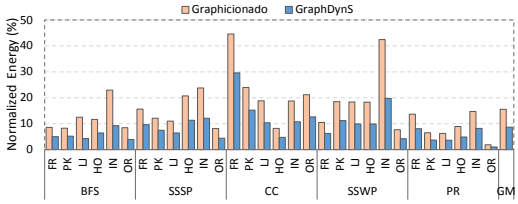

Figure 9: Energy consumption normalized to *Gunrock* (including HBM).

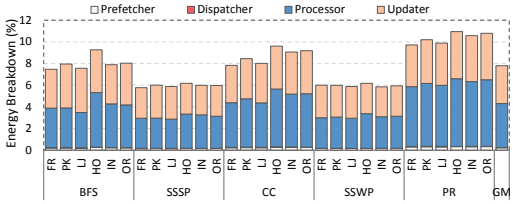

Figure 10: Energy breakdown.

• ***Memory Storage Usage:*** Fig. 11 shows the maximum off-chip memory storage usage of *GraphDynS* and the baselines at runtime, normalized to *Gunrock*. *GraphDynS* only uses 35% and 63% memory storage with respect to *Gunrock* and *Graphicionado* on average. The main reason for the reduction of memory storage is that *GraphDynS* does not require storing extra information such as preprocessing metadata and additional data in edges. *Gunrock* uses more than 2× storage than original graph data for storing preprocessing metadata. In the *Graphicionado* design, the edge data includes the source vertex ID *src_vid* and the active vertex data includes the active vertex ID *u.vid*. Contrarily, *src_vid* and *u.vid* are not required by *GraphDynS* since it has an edge counter and offset for each active vertex.

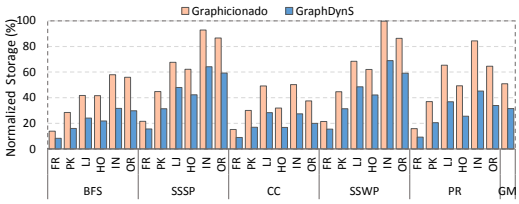

Figure 11: Off-chip memory storage usage normalized to *Gunrock*.

• ***Memory Access:*** Fig. 12 shows total data accessed from off-chip memory of *GraphDynS* and the baselines during runtime, normalized to *Gunrock*. Although *GraphDynS* accesses *offset* array additionally in each iteration, *GraphDynS* still reduces data accesses by 64% and 47% on average compared to *Gunrock* and *Graphicionado*. *Gunrock* produces many memory requests resulting from the random access for edge traversal. However, online preprocessing helps *Gunrock* significantly reduce data access on CC algorithm. Note that the data access of *Gunrock* in the cases of CC algorithm running on FR dataset and PR algorithm on IN dataset are smaller than

GraphDynS since there is good data locality of vertex property in these two cases. *GraphDynS* and *Graphicionado* use the on-chip memory to reduce a large amounts of random accesses. Nevertheless, as mentioned above, each access to an edge in *Graphicionado* includes *src_vid*, leading to extra accesses which are proportional to the number of edges.

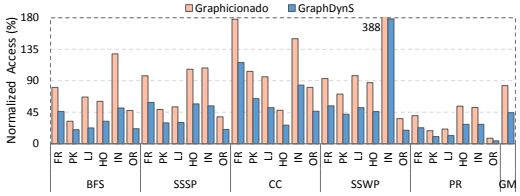

Figure 12: Memory accesses normalized to *Gunrock*.

• ***Utilization of Memory Bandwidth:*** Fig. 13 shows the average memory bandwidth utilization of *GraphDynS* and others. Benefiting from our prefetcher design, *GraphDynS* achieves 56% memory bandwidth utilization on average. A large amount of random accesses cause low bandwidth utilization of only 31% for *Gunrock*. *Graphicionado* has similar bandwidth utilization to *GraphDynS* since *Graphicionado* has more sequential accesses to edge data (i.e., extra access to *src_vid*, 1.65 × over *GraphDynS* on average) and thus the row buffer miss rate is lower than *GraphDynS*. However, *GraphDynS* performs better than *Graphicionado* due to more efficient utilization of memory bandwidth.

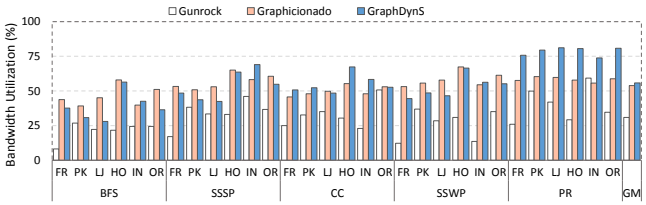

Figure 13: Memory bandwidth utilization.

7.1 Effects of Scheduling Optimization

A detailed evaluation on LJ dataset is presented to provide more insight into the scheduling optimization. The optimizations include Workload Balancing (**WB**), Exact Prefetching(**EP**), Atomic Optimization (**AO**) and Update Scheduling (**US**). We compared the following combinations to understand the effectiveness of the proposed techniques: 1) combining the workload balancing and exact prefeching techniques (**WE**); 2) combining **WE** with atomic optimization techniques (**WEA**); 3) combining **WEA** with update scheduling techniques (**WEAU**).

• ***Effects of Workload Scheduling.*** We first evaluate the load-balance scheduling method. Fig. 14(a) shows the number of scheduling operation reduction of *GraphDynS* with respect to *GraphDynS* without workload scheduling optimization. With **WB** optimization, *GraphDynS* reduces scheduling operations by up to 94% on average. Meanwhile, reducing 128 *DEs* to 16 *DEs* reduces 80% of *Dispatcher*'s area. Although 87.5% *DEs* are reduced, performance does not degrade. The key reason for scheduling complexity reduction is because of the proposed coarse granularity (a batch of edges or vertices) scheduling mechanism, enabling 16 *DEs* to achieve peak throughput of *Processor*.

As shown in Fig. 14(b), normalized workload among *PEs* in several heaviest workload iterations on SSWP algorithm are near to ideal value 1.0 (normalized to average workload in corresponding iteration) in *GraphDynS*. **WB** optimization improves performance by 7% performance on average, as shown in Fig. 14(c). Hence, **WB** optimization not only reduces scheduling overhead, but also balances workload efficiently.

• ***Effects of Data Access Scheduling.*** Next, we analyze the effectiveness of data access scheduling. As shown in Fig. 14(c), **WE** achieves 1.39× speedup on average, compared to *Graphicionado*. The data access to HBM has been reduced by 30% with exact prefetching optimization (**EP**) on average, As shown in Fig. 14(d). With **EP** optimization, *Prefetcher* prefetches edge list as quickly as it can after access to the active vertex data. Therefore, in Scatter phase, memory level parallelism is improved and performance stall due to data misses is reduced. Because most of the edge property have less than 2 elements, our exact edge prefetching will significantly improve memory access efficiency and traversal irregularity. Compared to other algorithms, BFS algorithm exhibits less speedup by **WE** because it spends most of time on *Apply* phase instead of *Scatter* phase.

In the next step, we analyze performance with **WEA** that shows 1.57× speedup compared to *Graphicionado*. Specifically, PR and CC algorithms benefit from **AO** optimization, achieving 20% and 5% of performance improvement respectively. This is because they have more RAW conflicts per cycle than other algorithms due to their higher throughput. For PR and CC running HO dataset, the performance improvement achieves 30% and 15% respectively. The reason is that HO dataset has larger edge-to-vertex ratio, which deteriorates the RAW conflicts with even higher throughput.

In summary, not only does data access scheduling reduce a large amount of unnecessary data access and execution time, but it also efficiently overcomes the traversal irregularity.

• ***Effects of Update Scheduling.*** Lastly, we evaluate the effectiveness of update scheduling. As shown in Fig. 14(c), **WEAU** achieves 1.8× speedup compared to *Graphicionado*. **US** technique alone reduces the execution time by 10%.

In the next step, we show the breakdown of the improvement brought by update operation and data access reduction. As shown in Fig. 14(d), the data access to off-chip memory has been reduced by 18% compared to *Graphicionado*. 60% of the update operations are eliminated with **US** technique. Specifically, for BFS algorithm, **US** reduces up to 88% update operations, 55% of data accesses and 28% of execution time, more than other algorithms. The reason is that *Apply* phase occupies up to 50% execution time while in other algorithms it occupies only 30% or even less. For PR algorithm, **US** cannot provide improvement because PR requires updating all vertices in each iteration. For SSSP algorithm, **US** reduces 63% of update operations and 8% of data accesses. However, it only reduces 4% execution time as *Apply* phase only occupies 20% of execution time for SSSP.

In summary, update scheduling greatly reduces unnecessary update operations and data accesses, which efficiently addresses the update irregularity and reduces execution time.

7.2 Scalability Analysis

• ***Performance over number of UEs:*** Fig. 14(e) shows the performance over the number of *UEs* on LJ dataset, normalized to the

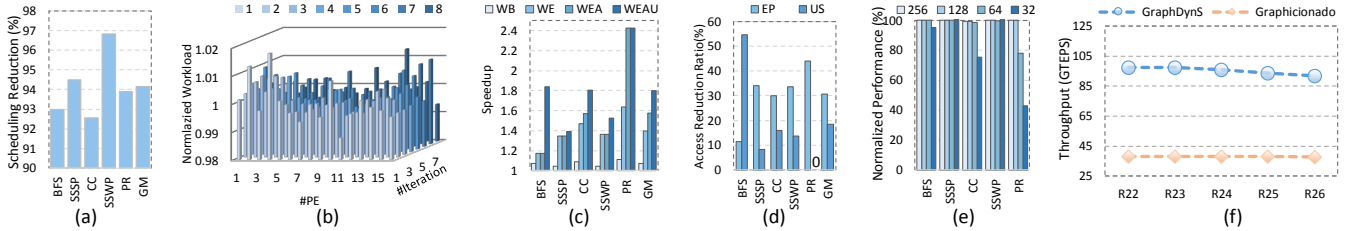

Figure 14: Efficiency of dynamic scheduling and scalability results. (a) Reduction ratio of scheduling of *DEs*, (b) Normalized workload among *PEs* in several iterations, (c) Breakdown of Speedup, (d) Normalized off-chip data access to *Graphicionado*, (e) Performance over number of *UEs*, (f) Throughput over the five synthetic graphs.

performance of 128 *UEs*. Results of PR and CC show that these two algorithms slow down 53% and 20% respectively from 128 *UEs* to 32 *UEs*. They are affected heavily by the number of *UEs*. This is because the reduce operations from different SIMT lanes of different *PEs* may access the same *UEs*, a situation which causes contention on *UEs*. These two algorithms can achieve up to 43 GTEPS throughput with 128 *UEs* while others achieve less than 17 GTEPS. Therefore, they have more contention compared to other algorithms, leading to performance degradation. In summary, the algorithm that achieves high throughput is more sensitive to the number of *UEs*.

• ***Throughput over large-scale synthetic graphs:*** Fig. 14(f) shows the throughput over the five large-scale synthetic graphs with the same edge-to-vertex ratio on PR algorithm. We use throughput of PR algorithm to evaluate the scalability of system performance since the throughput of PR algorithm heavily depends on the edge-to-vertex ratio of graph. The throughput of *GraphDynS* slows down slightly in R24, R25 and R26. This is because graphs are sliced and a single slice is processed at a time, a process of which causes repetitive accesses to active vertices. However, since *Graphicionado* can cache 2× *temporary vertex property* than *GraphDynS*, the throughput of *Graphicionado* slows down more gradually compared to *GraphDynS*. These results show that both *GraphDynS* and *Graphicionado* scale well to larger graphs.

8 RELATED WORK

• ***Graph analytics accelerators:*** Recent work [2, 6, 7, 11, 12, 45] use FPGA to accelerate graph workloads. These works need specific data partitioning to reduce the random memory accesses and RAW conflicts. Furthermore, limited by hardware resources, FPGA-based solutions cannot efficiently accelerate large scale graph processing at high performance. To overcome these limitations, Ozdal et al. [46] presents an architecture template based on asynchronous execution model to exploit memory-level parallelism. Although it achieves high throughput, it needs a complex design to ensure sequential consistency, and also the traversal irregularity causes inefficient bandwidth utilization since the random memory access still exits. Graphicionado [24] uses an on-chip memory to reduce large amount random memory access, and thus it achieves high performance and high efficiency. Whereas, it cannot tackle workload and update irregularities. Our work addresses these irregularities efficiently and achieves up to 1.9× speedup while consuming 1.8× less energy compared to Graphicionado.

Besides, recent work [1, 13, 56] modify the logic layer of 3D memory technology Hybrid Memory Cube (HMC) to offload graph workloads into HMC so that they can utilize the internal high bandwidth of memory-centric architecture. Nevertheless, logic layer has limited area and thermal budget. Other work [14, 50, 57] implement ReRAM-based graph processing accelerators to leverage the efficient compute capacity of ReRAM. Above work cannot be adopted as a near term solution since this memory technology is still immature. Different from above methods, our work presents a practical solution by leveraging the off-the-shelf HBM.

• ***GPU-based solutions:*** Integrated with high bandwidth memory, GPU-based solutions [20, 26, 27, 33, 42, 52] achieve orders of magnitude performance improvement over CPU solutions. However, most of these need expensive offline/online preprocessing to transform irregularity to be regular, which dominates the whole execution period, up to 2× of the processing time in *Gunrock* [52]. However, our work alleviates irregularity without preprocessing and achieves 4.4× speedup while reducing energy consumption by 91.7% compared to *Gunrock*.

9 CONCLUSION

In this paper, we propose a hardware/software co-design for graph analytics, called *GraphDynS*. Based on the decoupled datapath, *GraphDynS* can schedule the workload, data access, and update computation by considering the data dependency at runtime to alleviate the irregularity in graph analytics. *GraphDynS* achieves 1.9× speedup while consuming 1.8× less energy compared to the state-of-the-art graph analytics accelerator. Compared to the state-of-the-art GPU-based solution running on a NVIDIA V100 GPU, *GraphDynS* achieves 4.4× speedup with half the memory bandwidth while consuming 11.6× less energy.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program (Grant No. 2018YFB1003501), the National Natural Science Foundation of China (Grant No. 61732018, 61872335, and 61802367), the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDA18000000), the Innovation Project Program of the State Key Laboratory of Computer Architecture (Grant No. CARCH4408, CARCH4412, and CARCH4502), and the National Science Foundation (Grant No. 1730309).

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 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[2] O. G. Attia, T. Johnson, K. Townsend, P. Jones, and J. Zambreno. 2014. CyGraph: A Reconfigurable Architecture for Parallel Breadth-First Search. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*. 228–235. https://doi.org/10.1109/IPDPSW.2014.30

[3] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. 2018. Memory Hierarchy for Web Search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 643–656. https://doi.org/10.1109/HPCA.2018.00061

[4] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–386. https://doi.org/10.1109/HPCA.2019.00051

[5] S. Beamer, K. Asanovic, and D. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization*. 56–65. https://doi.org/10.1109/IISWC.2015.12

[6] B. Betkaoui, Y. Wang, D. B. Thomas, and W. Luk. 2012. A Reconfigurable Computing Approach for Efficient and Scalable Parallel Graph Exploration. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. 8–15. https://doi.org/10.1109/ASAP.2012.30

[7] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. 2006. Parallel FPGA-based all-pairs shortest-paths in a directed graph. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*. 10 pp.–. https://doi.org/10.1109/IPDPS.2006.1639347

[8] Ed Bullmore and Olaf Sporns. 2009. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience* 10 (04 02 2009), 186 EP –. https://doi.org/10.1038/nrn2575

[9] Cagla Cakir, Ron Ho, Jon Lexau, and Ken Mai. 2015. Modeling and Design of High-Radix On-Chip Crossbar Switches. In *Proceedings of the 9th International Symposium on Networks-on-Chip (NOCS '15)*. ACM, New York, NY, USA, Article 20, 8 pages. https://doi.org/10.1145/2786572.2786579

[10] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[11] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 105–110. https://doi.org/10.1145/2847263.2847339

[12] 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 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 217–226. https://doi.org/10.1145/3020078.3021739

[13] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (April 2019), 640–653. https://doi.org/10.1109/TCAD.2018.2821565

[14] Guohao Dai, Tianhao Huang, Yu Wang, Huazhong Yang, and John Wawrzynek. 2019. GraphSAR: A Sparsity-aware Processing-in-memory Architecture for Large-scale Graph Processing on ReRAMs. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference* (2019) *(ASPDAC '19)*. ACM, 120–126. https://doi.org/10.1145/3287624.3287637

[15] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[16] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16* (2016). ACM Press, 85–90. https://doi.org/10.1145/2851553.2851572

[17] A. Eisenman, L. Cherkasova, G. Magalhaes, Q. Cai, and S. Katti. 2016. Parallel Graph Processing on Modern Multi-core Servers: New Findings and Remaining Challenges. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2016-09). 49–58. https://doi.org/10.1109/MASCOTS.2016.66

[18] Stijn Eyerman, Wim Heirman, Kristof Du Bois, Joshua B. Fryman, and Ibrahim Hur. 2018. Many-Core Graph Workload Analysis. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018-11). IEEE, 282–292. https://doi.org/10.1109/SC.2018.00025

[19] J. A. Fernandez and J. Gonzalez. 1998. Hierarchical graph search for mobile robot path planning. In *Proceedings. 1998 IEEE International Conference on Robotics and Automation (Cat. No.98CH36146)*, Vol. 1. 656–661 vol.1. https://doi.org/10.1109/ROBOT.1998.677047

[20] Zhisong Fu, Michael Personick, and Bryan Thompson. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems* (2014) *(GRADES'14)*. ACM, 2:1–2:6. https://doi.org/10.1145/2621934.2621936

[21] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[22] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[23] Google. 2018. Knowledge âĂŞ Inside Search âĂŞ Google. https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html

[24] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783759

[25] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFboost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 411–424. https://doi.org/10.1109/ISCA.2018.00042

[26] F. Khorasani, R. Gupta, and L. N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 39–50. https://doi.org/10.1109/PACT.2015.15

[27] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (2014) *(HPDC '14)*. ACM, 239–252. https://doi.org/10.1145/2600212.2600227

[28] Y. Kim, W. Yang, and O. Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (Jan 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[29] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. 2009. How Much Parallelism is There in Irregular Applications?. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1504176.1504181

[30] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web* (2010) *(WWW '10)*. ACM, 591–600. https://doi.org/10.1145/1772690.1772751

[31] Yucheng Low. 2013. *Graphlab: A distributed abstraction for large scale machine learning*. Ph.D. Dissertation. University of California, Berkeley.

[32] Andrew Lumsdaine, Douglas P. Gregor, Bruce Hendrickson, and Jonathan W. Berry. 2007. Challenges in Parallel Graph Processing. 17 (2007), 5–20. https://doi.org/10.1142/S0129626407002843

[33] 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 *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 195–207. https://www.usenix.org/conference/atc17/technical-sessions/presentation/ma

[34] 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 the 2010 ACM SIGMOD International Conference on Management of Data* (2010) *(SIGMOD '10)*. ACM, 135–146. https://doi.org/10.1145/1807167.1807184

[35] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and Analysis of Online Social Networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC '07)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/1298306.1298311

[36] MITRE. 2018. CyGraph. (2018). https://www.mitre.org/research/technology-transfer/technology-licensing/cygraph

[37] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[38] Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. 2014. Information Network or Social Network?: The Structure of the Twitter Follow Graph. In *Proceedings of the 23rd International Conference on World Wide Web* (2014) *(WWW '14 Companion)*. ACM, 493–498. https://doi.org/10.1145/2567948.2576939

[39] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017-02). 457–468. https://doi.org/10.1109/HPCA.2017.54

[40] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C. Y. Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC15: International Conference for High Performance Computing, Networking, Storage and Analysis* (2015-11). 1–12. https://doi.org/10.1145/2807591.2807626

[41] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units - GPGPU-6* (2013). ACM Press, 96–107. https://doi.org/10.1145/2458523.2458533

[42] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (2018) *(ASPLOS '18)*. ACM, 622–636. https://doi.org/10.1145/3173162.3173180

[43] NVIDIA. 2016. nvGRAPH. https://developer.nvidia.com/nvgraph

[44] Mike O'Connor. 2014. Highlights of the high-bandwidth memory (hbm) standard. In *Memory Forum Workshop*.

[45] Tayo Oguntebi and Kunle Olukotun. 2016. GraphOps: A Dataflow Library for Graph Analytics Acceleration. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (2016) *(FPGA '16)*. ACM, 111–117. https://doi.org/10.1145/2847263.2847337

[46] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. 2016. Energy Efficient Architecture for Graph Analytics Accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016-06). 166–177. https://doi.org/10.1109/ISCA.2016.24

[47] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli. 2016. A Survey of Motion Planning and Control Techniques for Self-Driving Urban Vehicles. *IEEE Transactions on Intelligent Vehicles* 1, 1 (March 2016), 33–55. https://doi.org/10.1109/TIV.2016.2578706

[48] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[49] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013) *(PPoPP '13)*. ACM, 135–146. https://doi.org/10.1145/2442516.2442530

[50] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2018-02). 531–543. https://doi.org/10.1109/HPCA.2018.00052

[51] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983

[52] 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 the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2016) *(PPoPP '16)*. ACM, 11:1–11:12. https://doi.org/10.1145/2851141.2851145

[53] S. J. E. Wilton and N. P. Jouppi. 1996. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (May 1996), 677–688. https://doi.org/10.1109/4.509850

[54] Q. Xu, H. Jeon, and M. Annavaram. 2014-10. Graph processing on GPUs: Where are the bottlenecks?. In *2014 IEEE International Symposium on Workload Characterization (IISWC)* (2014-10). 140–149. https://doi.org/10.1109/IISWC.2014.6983053

[55] Pengcheng Yao, Long Zheng, Xiaofei Liao, Hai Jin, and Bingsheng He. 2018. An efficient graph accelerator with parallel data conflict management. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques - PACT '18* (2018). ACM Press, 1–12. https://doi.org/10.1145/3243176.3243201

[56] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing communication for PIM-based graph processing with efficient data partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 544–557.

[57] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. GRAM: Graph Processing in a ReRAM-based Computational Memory. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference* (2019) *(ASPDAC '19)*. ACM, 591–596. https://doi.org/10.1145/3287624.3287711