GraphPEG: Accelerating Graph Processing on GPUs

YASHUAI LÜ, Space Engineering University, China
HUI GUO, LIBO HUANG, QI YU, LI SHEN, NONG XIAO, and ZHIYING WANG,
National University of Defense Technology, China

Due to massive thread-level parallelism, GPUs have become an attractive platform for accelerating large-scale data parallel computations, such as graph processing. However, achieving high performance for graph processing with GPUs is non-trivial. Processing graphs on GPUs introduces several problems, such as load imbalance, low utilization of hardware unit, and memory divergence. Although previous work has proposed several software strategies to optimize graph processing on GPUs, there are several issues beyond the capability of software techniques to address.

In this article, we present GraphPEG, a graph processing engine for efficient graph processing on GPUs. Inspired by the observation that many graph algorithms have a common pattern on graph traversal, GraphPEG improves the performance of graph processing by coupling automatic edge gathering with fine-grain work distribution. GraphPEG can also adapt to various input graph datasets and simplify the software design of graph processing with hardware-assisted graph traversal. Simulation results show that, in comparison with two representative highly efficient GPU graph processing software framework Gunrock and SEP-Graph, GraphPEG improves graph processing throughput by 2.8× and 2.5× on average, and up to 7.3× and 7.0× for six graph algorithm benchmarks on six graph datasets, with marginal hardware cost.

CCS Concepts: • **Computer systems organization → Single instruction, multiple data**;

Additional Key Words and Phrases: GPU, graph processing, load balance, hardware acceleration

ACM Reference format:
Yashuai Lü, Hui Guo, Libo Huang, Qi Yu, Li Shen, Nong Xiao, and Zhiying Wang. 2021. GraphPEG: Accelerating Graph Processing on GPUs. *ACM Trans. Archit. Code Optim.* 18, 3, Article 30 (April 2021), 24 pages.
https://doi.org/10.1145/3450440

1 INTRODUCTION

Graph representations are commonly used in many real-world application domains, ranging from social networks, machine learning, and data mining to biology and scientific computing [15, 33]. Graph algorithms, such as **breadth-first search (BFS)**, **single-source shortest paths (SSSP)**, **betweenness centrality (BC)**, **ST-connectivity (STCON)**, and so on, often serve as building blocks to these applications. In the era of "Big-Data," parallelization of graph algorithms is imperative for real-world, large-scale datasets.

This work was supported by the NSF of China (Grants No. 61872374 and No. 61672526).
Authors' addresses: Y. Lü, Space Engineering University, now working at Cambricon Technology Co., Beijing, China; email: lvyashuai@cambricon.com; H. Guo, L. Huang (corresponding author), Q. Yu, L. Shen, N. Xiao, and Z. Wang, School of Computer, National University of Defense Technology, Changsha, Hunan, China; email: {huiguo, libohuang, qiyu, lishen, nongxiao, zywang }@nudt.edu.cn.

1544-3566/2021/04-ART30
https://doi.org/10.1145/3450440

Contemporary processor architecture provides increasing parallelism to deliver high throughput while maintaining energy efficiency. Modern GPUs are at the leading edge of this trend. The high computational power provided by the *single instruction multiple thread***(SIMT)** execution model presents an opportunity for accelerating graph algorithms on GPUs. Thus, numerous studies have been conducted to improve the performance of various fundamental graph algorithms, such as BFS, SSSP, BC, STCON, and to develop high-level GPU graph processing libraries that deliver high-performance [4, 7, 8, 14, 16, 17, 25, 27, 30, 31, 32, 37, 45, 46] and excellent programmability [6, 9, 10, 18, 19, 47, 48, 52].

However, efficient graph processing on GPUs remains an open challenging problem. Although GPUs' excellent peak throughput and energy efficiency have been shown in many application domains, the irregularity property in graphs has prevented GPUs from achieving high performance efficiently, due to the demand of SIMT for repetitive processing patterns on regular data. The inherent irregularity of graph data structures leads to irregularity in data access and control flow. Consequently, accelerating graph processing becomes a difficult task. Specifically, achieving efficient graph processing parallelization poses several obstacles, such as control flow divergence, non-uniform work distribution, and memory divergence.

Prior work mainly focuses on addressing the problems of control flow divergence and workload imbalance. Although considerable software-based strategies have been proposed [6, 7, 8, 16, 18, 19, 25, 27, 32, 37, 47, 48], there are several limitations that are difficult to be addressed by a software approach. First, in some cases, software-based strategies incur high load-balancing overhead. Second, the graph datasets do not admit a one-size-fits-all characterization; that is, no one software strategy performs well for various input datasets. Third, irregular memory accesses introduce memory divergence. Fourth, the complicated codes introduced by software approaches are not friendly to people who are unfamiliar with GPU programming. The hardware-based proposals for graph processing acceleration on GPUs are relatively few. Lakshminarayana et al. proposed spare register-aware prefetching for graph algorithms on GPUs [22], which does not address the load imbalance problem. Kim et al. proposed HWWL [20] to accelerate worklist operations on GPUs. However, HWWL cannot adapt to different graph datasets (see later part for details).

This article presents GraphPEG, a low-cost hardware graph processing engine for GPUs, to address the inefficiencies of graph processing. Our main contributions are as follows.

- We analyze the limitations of software optimization strategies for graph processing on GPUs and identify the common data access pattern of graph traversal algorithms, thereby motivating the design of GraphPEG.
- We propose GraphPEG, combining automatic edge gathering and fine-grain work distribution to accelerate graph processing on GPUs. GraphPEG improves the performance of graph processing by enhancing the utilization of computing resources and memory bandwidth, and low overhead load balancing.
- We provide a detailed evaluation of GraphPEG with performance comparisons to several software and hardware GPU graph processing implementations. With minor architectural modifications, GraphPEG can substantially improve graph processing performance on GPUs.

2 BACKGROUND AND MOTIVATION

This section introduces a brief background on graph processing on GPUs, discusses the limitations of software-based optimization strategies, and motivates the need for a hardware-based approach. For convenience of description, we use NVIDIA's terminology throughout the article.

```
__global__ void BFSDataDriven(int* depth, int iteration, Graph graph,
                              Vertexlist in_vl, Vertexlist out_vl) {
 1   for (int base_id = blockIdx.x*BLOCK_SIZE;
 2           base_id < in_vl.size(); base_id+=GRID_SIZE ) {
 3     int task_id = base_id + threadIdx.x;
 4     int nn = in_vl.pop_id(task_id);
 5     int edge_start = graph.neighbor_start(nn);
 6     int edge_end = graph.neighbor_end(nn);
 7     for (int i = edge_start; i < edge_end; i ++) {
 8       int edge = graph.edge_array[i];
 9       int alt_depth = iteration;
10       int old_depth = atomicMin(&depth[edge], alt_depth);
11       if (alt_depth < old_depth) out_vl.push(edge);
12     }
13   }
14 }
```

Fig. 1. Pseudocode of a simple data-driven implementation of the BFS algorithm.

2.1 Graph Processing on GPUs

Many graph algorithms have a common traversal pattern that makes multiple iterations over a graph. In each iteration, an operator is applied to vertices in a graph to update the state of the graph and generate substantial work for the next iteration. The data-driven approach is one of the most popular methods to efficiently map graph algorithms to GPUs [35]. It maintains a vertex-list/worklist of active vertices in the graph; in each iteration, only the vertices in the vertex-list are processed, and their neighbors in the graph are visited. A vertex can either *push* the updates to its neighbors, or let its neighbors *pull* the updates [47]. Figure 1 shows an example kernel code of the BFS algorithm using a simple Push data-driven implementation.

The pseudocode in Figure 1 is similar to a sequential BFS implementation on CPU and easy to understand even for people unspecialized in GPU programming. However, this pseudocode may lead to severe control flow divergence and load imbalance. The parallelism exists at two levels of the BFS algorithm; that is, (1) the vertices in the current input vertex-list can be processed in parallel, and (2) the edges of the active vertices can be processed in parallel. The implementation in Figure 1 applies the first level of parallelism by assigning each vertex in the vertex-list to a thread, leaving the second level of parallelism unimplemented. In other words, each thread separately processes the edges of the assigned active vertex, as illustrated in the inner loop of the pseudocode in Figure 1. In this situation, because they have different amounts of work to process, threads may iterate different times over the inner loop and GPU suffers from low utilization.

To address the aforementioned problem, previous works have proposed several strategies [8, 16, 18, 19, 25, 32, 48] to balance workload between threads. For example, the *cooperative blocks* strategy [8] assigns a block of threads to process a set of vertices in parallel. First, all the neighbor list offsets of the vertices, which are assigned to a thread block, are loaded into shared memory. Then, the neighbor list of a vertex is split, and multiple threads can cooperatively process per-edge operations on the neighbor list. This method balances work within a thread block given that threads cooperatively process edges of a set of vertices. However, it may introduce inter-block load imbalance due to unbalanced amount of work processed by different blocks. To address this issue, the *load-balanced partitioning* [8] strategy organizes groups of edges into equal-length chunks and

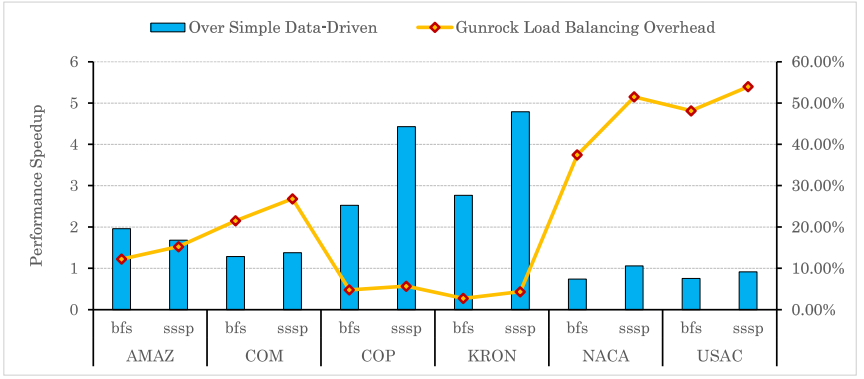


Fig. 2. Performance improvement over simple data-driven implementation and the load-balancing overhead of the Gunrock.

assigns each chunk to a block. This division requires finding the starting and ending indices for all blocks. The sorted search [5] is used to map the indices with the scanned edge offset array. Before processing a neighbor list of a new node, a binary search should be performed to find the originating vertex for the edges to be processed. Although this method ensures a perfect load balance between thread blocks, it introduces more load-balancing overhead than the *cooperative blocks* strategy.

In summary, software-based load-balancing strategies have some drawbacks as follows. (1) The load balance is achieved at the cost of considerable complicated codes that are not friendly to people who are unfamiliar with GPU programming. (2) For graphs that are not highly skewed, an implementation using a load-balancing strategy may sometimes incur worse performance than the simple straightforward implementation. (3) No single strategy is beneficial for all types of graph datasets [47, 48], thereby making it difficult for graph application developers to make an optimal choice between different strategies, especially when the characteristics of an input graph dataset are undetermined beforehand. (4) Moreover, the memory divergence caused by irregular memory accesses is not addressed, thereby limiting the performance improvement of these software strategies.

2.2 Motivation

As described above, to efficiently process graphs on GPUs, two levels of parallelism should be taken care of, i.e., the processing of the vertices in the vertex-list and the processing of the edges in the ***edge-frontier (EF)*** (the subset of edges that are traversed in a given iteration [31, 48]). However, the SIMT execution model makes it difficult to efficiently extract the two levels of parallelism at the same time. The reason is twofold. (1) The out-degree of vertex accessed in the first level determines the number of iterations executed in the second level. (2) Dynamically detecting load imbalance and exchanging work between threads is difficult for software strategies due to the expensive global inter-thread communication on GPUs. Figure 2 shows Gunrock's performance improvement over the simple data-driven implementation and the load-balancing overhead of Gunrock. The experimental setup is described in Section 4. Clearly, datasets have an important impact on the load-balancing overhead of Gunrock. Generally, the load-balancing overheads for datasets with small average out-degrees, such as NACA and USAC, are relatively high. Moreover, the performance improvement decreases with the increase of load-balancing overhead. In some cases, Gunrock even performs worse than the simple data-driven implementation.

A key observation is that the two levels of parallelism stem from a common data access pattern shared by many graph algorithms; that is, a pair of nested loops in which the outer loop iterates over a set of vertices and the inner loop iterates over the edges connected to a given vertex. Notably, it is the edges in the EF rather than the vertices in the active vertex set that are the final items processed by a graph algorithm. Based on this observation, we can design a hardware scheme that follows this data access pattern to gather edges automatically and directly supply threads with the edges of the EF in a load-balanced manner.

This hardware scheme can bring some potential benefits over software strategies. First, since automatic edge gathering conducts graph traversal in algorithms with hardware support, the kernel code only needs to process edges in the EF. Second, this scheme can gather enough edges and distribute them evenly to fully utilize the GPU computing resources and eliminate control flow divergence. Third, automatic edge processing can gather edge data as early as possible to improve the utilization of memory bandwidth, thereby alleviating memory divergence. Fourth, this scheme unburdens graph application developers from making choices between different strategies, because the hardware scheme can adapt to various graph datasets. These benefits motivate us to design GraphPEG to improve graph processing performance on GPUs.

Graph processing is quite complicated in terms of algorithms, application behaviors, and data sets. For graph processing on GPUs, there are many challenges to be addressed. The GraphPEG proposed in this article focuses on improving graph processing efficiency for data-driven graph traversal scheme on a single GPU. We leave other challenges such as graph processing on multiple GPUs and dynamic mutation of the graph structure to our future work.

3 GRAPHPEG

GraphPEG, a hardware engine that addresses the limitations of graph processing on GPUs, is presented in this section. The key idea of GraphPEG is twofold. First, it takes advantage of the common data access pattern in graph traversal algorithms and automatically gathers edge data on the basis of the neighbor-list indices of active vertices. Second, the gathered edges is evenly distributed to threads to achieve dynamic load balancing.

In the following subsections, we initially provide an architectural overview of GraphPEG and then describe the manner in which GraphPEG is used by introducing the primitives and a pseudocode example. Finally, we describe the detailed workflow of GraphPEG.

3.1 Overview

GraphPEG is attached to each **Streaming Multiprocessor (SM)** to improve graph processing throughput on GPUs. GraphPEG also aims at minimizing its hardware cost to make it highly practical for implementation, therefore it reuses maximally the original hardware resources of a GPU.

Figure 3 illustrates the architectural overview of GraphPEG and integration with an SM. GraphPEG utilizes shared memory as an on-chip scratchpad buffer for temporary graph data (active vertices and edge data), which is similar to many software graph processing implementations [8, 16, 18, 19, 25, 31, 32, 48, 51]. A kernel that utilizes GraphPEG needs to configure the maximal shared memory size that GraphPEG can use. The GraphPEG is mainly composed of three functional units and a central data structure. The three functional units are as follows. (1) The *edge gathering unit* is responsible for gathering edge data from the global memory into shared memory on the basis of the indices gathered from adjacency lists. (2) The *vertex coordination unit* controls refilling/spilling of the vertices in a vertex-list into/off shared memory. (3) The *edge-frontier (EF) exchange unit* is responsible for balancing the edges of EFs between different SMs. (4) The central data structure is called *adjacency list working table*, which tracks the index and edge

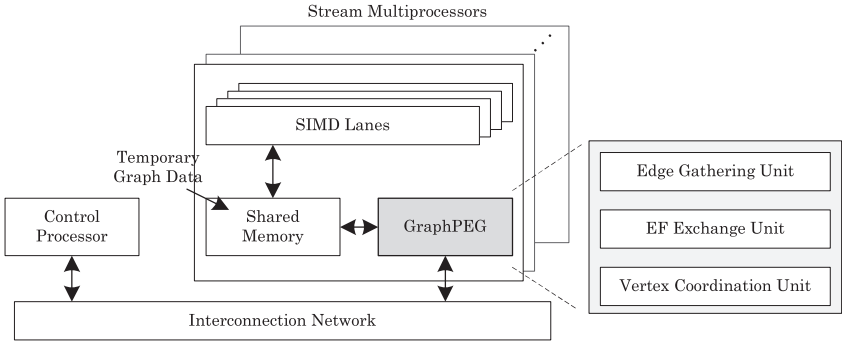


Fig. 3. Overview of GraphPEG.

Table 1. Intrinsic Functions of GraphPEG

Intrinsic Functions	Description
void __pop_edge_src(int vl, int& src)	Pop the source vertex.
void __pop_edge_dst(int vl, int& dst)	Pop the destination vertex.
void __pop_edge_wt(int vl, T& wt)	Pop the weight of an edge. T denotes any basic data types.
void __pop_edge_idx(int vl, int& idx)	Pop the index of an edge.
void __pop_vert(int vl, int& v)	Pop a vertex from an input vertex-list.
void __push_vert(int vl, int v)	Push a vertex into an output vertex-list.

gathering states. (5) Finally, GraphPEG leverages GPU's control processor [40] for global workload balancing.

3.2 Programming Interface

Before describing the details of GraphPEG's workflow, we first describe its programming interface and how it interacts with GraphPEG.

GraphPEG automatically gathers edges and balances them between GPU threads. Accordingly, a graph application can simply replace the graph traverse code with the intrinsic functions of GraphPEG to process edges. Different from some speciaized graph analytic hardware accelerators [13, 41, 49] that provide support for general graph computations, GraphPEG is designed as a functional unit with minimal hardware cost, so that it can be integrated with an SM of a GPU to accelerate graph traversal operations. As a result, the general graph computations, such as updating the values bound to vertices or edges, are left to the GPU. Table 1 outlines the intrinsic functions introduced to interact with GraphPEG.

Fundamentally, the GraphPEG works on vertex-lists and edge frontiers. GraphPEG can be configured to one of two working modes: *edge gathering* mode and *vertex-list* mode. The edge gathering mode is the primary and most frequently used mode. In this mode, GraphPEG automatically gathers edge data (including the source vertex, destination vertex, edge weight, and edge index), which can be accessed by a kernel via the *__pop_edge* intrinsic functions. The vertex-list mode serves as a complementary to the edge gathering mode and is usually used to filter out certain vertices of a vertex-list on the basis of an application-specific criterion. For example, in SSSP, the vertex-list mode is used to split a vertex-list into *near* and *far* sets [8].

The *__pop_vert* intrinsic function is used in the vertex-list mode to get vertices from the input vertex-list. Note that, this function cannot be used in the edge gathering mode, because the input

```
__global__ void BFSPush(int* level, int iteration,Graph graph,
                        int in_vl, int out_vl) {
 1   int dst=-1;
 2   do {
 3     __pop_edge_dst(in_vl, &dst);
 4     if (dst >= 0) {
 5       int alt_level = iteration;
 6       int old_level = atomicMin(&level[dst], alt_level);
 7       if (alt_level < old_level) {
 8         __push_vert(out_vl, dst);
 9       }
10     }
11   } while (dst >= 0);
12 }
```

(a) BFS kernel using Push for vertex updates.

```
__global__ void BFSPull(int* level, int iteration, Graph graph,
                        int in_vl, int out_vl, bool* active) {
 1   int src=-1, dst = -1;
 2   do {
 3     __pop_edge_src(in_vl, &src);
 4     __pop_edge_dst(in_vl, &dst);
 5     if (src >= 0 && active[src]) {
 6       if (level[dst] > iteration) {
 7         level[dst] = iteration;
 8         __push_vert(out_vl, dst);
 9       }
10     }
11   } while (src >= 0);
12 }
```

(b) BFS kernel using Pull for vertex updates.

Fig. 4. Pseudocodes of BFS kernels using GraphPEG.

vertex-list is automatically processed by GraphPEG. The *__push_vert* intrinsic function is used to push newly generated vertices into the output vertex-list and can be used in both working modes. In addition to working mode configuration, an application should configure GraphPEG the edge data needed for kernel execution in the edge gathering mode. For example, if edge weights are not needed, an application should configure GraphPEG not to load edge weights during kernel execution.

Figure 4 illustrates a pseudocode example that utilizes GraphPEG to accelerate BFS. GraphPEG supports both Push and Pull [47] for vertex updates. The pseudocode in Figure 4(a) uses Push for vertex updates, whereas the pseudocode in Figure 4(b) uses Pull for vertex updates. Evidently, both pseudocode kernels are highly concise, and even simpler than the pseudocode kernel in Figure 1. Compared with Figure 1, only one loop is present in the pesudocode kernel of Figure 4(b), because the outermost loop of is replaced by GraphPEG's intrinsic function *__pop_edge_dst* (Line 3 of Figure 4(a)). This intrinsic function returns the neighbors of the active vertices in an input vertex-list. A negative value indicates that all neighbors of vertices in the input vertex-list have been consumed. The GraphPEG guarantees that the gathered edges are balanced between

different threads. The intrinsic function *__push_vert* is used to push a vertex into the output vertex-list. For Pull updates (pseudocode in Figure 4(b)), a kernel needs to use the *__pop_edge_src* intrinsic function to obtain the source vertex of an edge. If a Pull update succeeds, then the destination vertex instead of source vertex is pushed into the output vertex-list.

Since the underlying hardware of GraphPEG is responsible for the execution of these intrinsic functions, GPU executes the kernel code in a persistent thread model, that is only a few thread blocks running on an SM at a time. Therefore, programmers unnecessarily take care of the mapping between software thread and the underlying hardware, and only need to focus on implementing the function of graph applications.

3.3 Workflow

In this subsection, we first describe the automatic edge gathering and work distribution workflow of GraphPEG, and then describe a lightweight scheme for EF load balancing.

3.3.1 Automatic Edge Gathering and Work Distribution. To efficiently store large graphs in memory, a graph's vertices and edges are usually represented with two one-dimensional arrays: the vertex array and the edge array (e.g., the most commonly used compressed sparse row (CSR) or **compressed sparse column (CSC)** format). Although other encoding schemes have been proposed for graph processing [18, 19], the GPU graph programming frameworks, such as Gunrock and SEP-Graph [47, 48], that are based on the CSR/CSC format show performance superiority.

The *i*th entry of the vertex array contains an index to the edge array. Specifically, such an index points to the start of an adjacency list that contains the neighbors of vertex *i*. The entries in the edge array store vertex identifiers that can index the vertex array. For weighted graphs, an extra array stores the weight of each edge. When using Push mode for vertex updates, an application should use the CSR graph format where the vertex array contains source vertices. Whereas for Pull mode, an application should use the CSC format where the vertex array contains destination vertices.

Each GraphPEG in an SM maintains its local input and output vertex-lists to prevent memory contention. However, these local vertex-lists are software interface transparent; in other words, the application developers can treat the input and output vertex-list as one single global vertex-list.

Figure 5 shows the workflow of edge processing in GraphPEG. The central data structure named **adjacency list working table (ALWT)** is used to gather edge data (❶). Each entry of the ALWT stores the status data of an active vertex. The entry has four fields: the base vertex (Vertex in the figure), the start and end indices of the adjacency list in the edge array (SIndex and EIndex, respectively), and the current working status (Status). There are four statuses in total: **fetching index (FI)**, **ready (RD)**, **fetching edge (FE)**, and **invalid (IVD)**. Note that for the CSR graph format, the base vertex represents the source vertex of an edge, whereas for the CSC graph format, the base vertex represents the destination vertex.

To fetch edge data from the global memory, the first step is to fetch adjacency list indices from the vertex array. In each clock cycle, if an IVD entry exists in the ALWT, and the vertex-list is not empty, the edge gathering unit will change the state of an IVD entry into FI (❷), and send a memory request to the LD/ST unit of the SM to fetch the start and end indices from the vertex array (❸). The address (virtual) of this memory request is calculated by adding the base address of the vertex array with an offset indicated by the first item in the vertex-list. This memory request will go through L2 cache by default and the virtual address of the memory request will be translated into the corresponding physical address in the TLB just like other memory requests. When the two indices are fetched into the SM from the global memory, they will be filled into the allocated entry in the ALWT, and the entry state will be changed into RD.

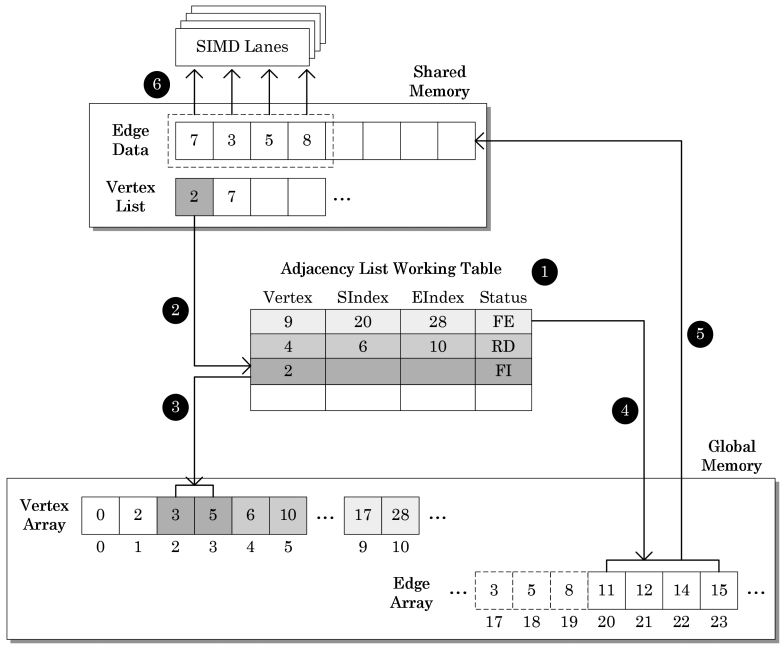


Fig. 5. Workflow of edge gathering in GraphPEG.

In each clock cycle, if a RD entry exists in the ALWT, and no FE entry exists, the edge gathering unit will change a RD entry into FE state and issue memory requests to fetch edge data from the edge array and edge weight array (if necessary) on the basis of the start and end indices (❹). The start index of the entry increases by the number of the edge data being fetched each time. The fetched edge data is filled into shared memory, given that it is not full (❺).

When the number of fetched edges in shared memory is equal to or larger than the warp size, or all edges have been gathered, the instructions that correspond to the *__pop_edge* intrinsic function can be issued into the pipeline to retrieve edges from shared memory (❻). Otherwise, these instructions will be stalled until the GraphPEG gathers enough edges from the global memory. Note that the edges supplied to a warp can belong to different vertices, as illustrated in Figure 5. As the threads of a warp simultaneously retrieve edge data from shared memory, warp divergence or load imbalance within a warp is avoided. The workload is also balanced between warps and thread blocks in an SM as long as warps and thread blocks are scheduled with equal priority.

The *vertex coordination unit* monitors the sizes of the vertex-lists in shared memory. It spills extra vertices into the global memory directly, once the output vertex-list in shared memory is full. This functional unit is also responsible for refilling vertices from the global memory to the input vertex-list in shared memory. As previously described, the input and output vertex-lists for each GraphPEG are localized to avoid memory contention; hence, each GraphPEG has its own global memory space for refilling and spilling vertices. For clarity, the workflow of the coordination unit is not shown in Figure 5.

For the *vertex-list* working mode, the edge gathering unit and ALWT are disabled, and threads can retrieve vertices from shared memory using the *__pop_vert* intrinsic function. In this mode, the GraphPEG works similarly to a **hardware worklist (HWWL)** [3, 20, 23, 43, 44].

3.3.2 Lightweight EF Load Balancing. The automatic edge gathering and work distribution scheme described above guarantees workload is balanced in a thread block. However, thread blocks scheduled onto different SMs may still suffer from load imbalance, since the GraphPEGs on some SMs can hog a majority of edges. We present a lightweight scheme to alleviate this issue by allowing work to be redistributed between SMs.

Fundamentally, two types of load balancing should be considered by GraphPEG, that is, the vertex-list and EF load balancing. However, based on experimental observation (see experimental evaluation section for more details), we found that the vertex-list load balancing makes little contribution to load balancing between SMs. The reasons are as follows. First, the vertex-list load balancing is expensive. It involves moving a large number of vertices between local vertex-lists that reside in different global memory locations. Second, for graph processing, edges are the final items to be processed. However, the vertex-list load balancing cannot guarantee edges are balanced between different SMs, especially in the case of power-law graphs. Thus, we only focus on the EF load balancing in this study.

Intuitively, the EF load balancing faces the same issue as the vertex-list load balancing, that is, moving a large number of edges between different GraphPEGs, which may lead to high load-balancing overhead. To reduce this overhead, we propose a simple and lightweight scheme based on two key observations. One is that we can exchange the entries of the ALWTs instead of transferring a large number of edges to achieve EF load balancing. Another is that the load imbalance between different SMs occurs much less frequently than the load imbalance in an SM. The reason is twofold. First, load balancing in an SM involves balancing workload between hundreds of threads that are scheduled on to the same SM, while load balancing between SMs only needs to balance workload between tens of SMs. Second, serious load imbalance usually occurs in the presence of power-law graphs with highly skewed degree distributions. In power-law graphs, a few vertices holds a huge number of edges, while the out-degrees of the rest vertices are not highly skewed. Consequently, the role of load balancing between different SMs is to detect the vertices with extremely large out-degrees and balance their edges between SMs. However, only a few number of vertices have such a large out-degrees, therefore the load balancing does not need to be performed frequently. On this basis, we propose an ALWT entry exchange scheme with simple load-balancing policies for GraphPEG to balance edges between SMs.

To minimize the hardware cost of GraphPEG and the modifications to the baseline GPU architecture, GraphPEG does not use a dedicated task routing network [20, 21, 23, 44] for load balancing. Instead, it leverages the GPU's control processor [40]. The control processor on a contemporary GPU assigns kernels to SMs and manages hardware resources. To manage load balancing for graph processing tasks, several control registers are added to each SM, and the control processor is extended to monitor and manage these registers. The added control registers are EF size register (R_{EFS}), balancing control register (R_{BC}), balancing state register (R_{BS}), and two balance working registers (R_{BW0} and R_{BW1}). These registers act as the interaction interface between the EF exchange unit of GraphPEG and the control processor.

Figure 6 is a walkthrough example to show how the control processor manages EF load balancing with an interval-based policy. Register R_{EFS} records the total number of edges to be processed in an SM, which is the sum of the number of edges that have been previously loaded into shared memory and the total number of edges in the ALWT. The control processor periodically checks the value of register R_{EFS} in each SM. If the difference between the maximum and minimum edges to be processed exceeds a pre-defined threshold (time **T0** in Figure 6), then the control processor initiates the EF load balancing between the two SMs. Based on experimental observation and hardware cost consideration, 512 is selected as the threshold in this article. For practical implementations, both load-balancing interval and threshold can be configured with runtime/driver APIs.

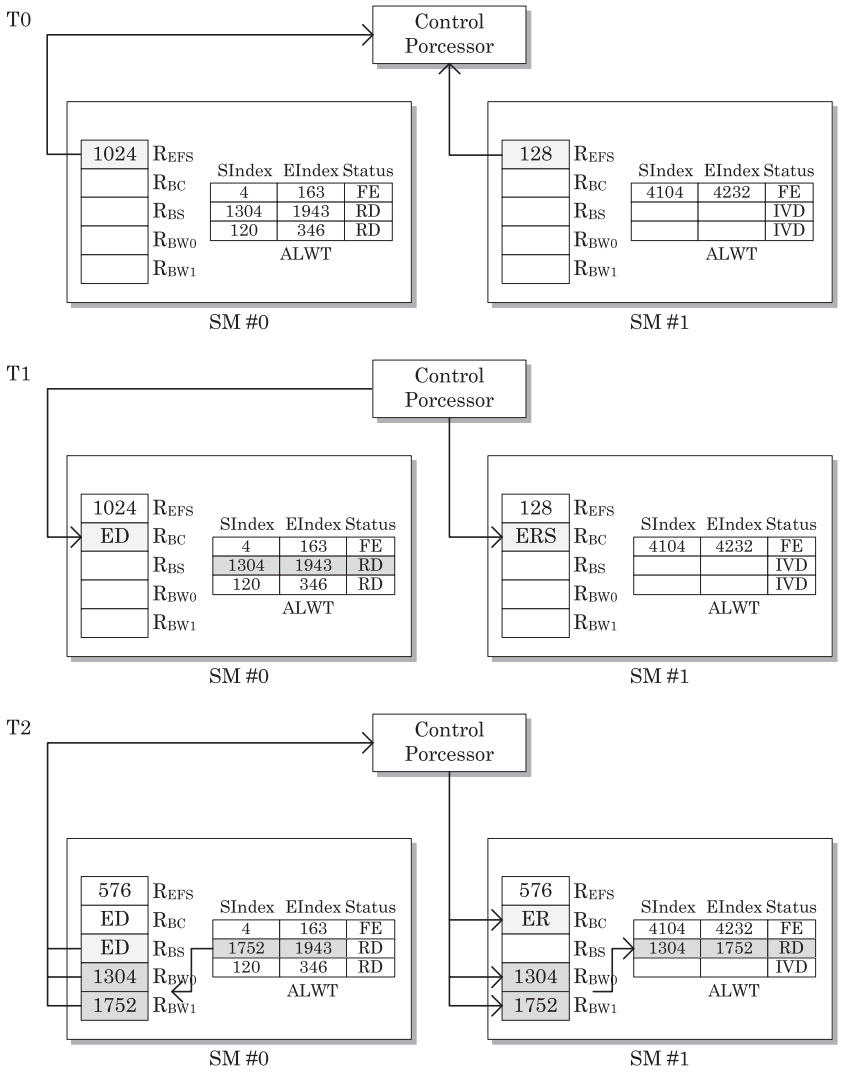


Fig. 6. Walkthrough example for EF load balancing.

In Figure 6, at time **T0**, SM #0 (has the maximum R_{EFS} value) is designated as the work donator, whereas SM #1 (has the minimum R_{EFS} value) is considered the work receiver. The EF balance size is determined on the basis of the value difference between the donator's and the receiver's R_{EFS}. As shown in Figure 6, the balance size is 448. With the balance size, the control processor issues a balancing request to the donator. At time **T1**, the control processor sets the register R_{BC} of SM #0 to the *edge donating* (ED) state and sets the register R_{BC} of SM #1 to the *edge receiving standby* (ERS) state to initiate load balancing. For EF load balancing, only one ALWT entry is transferred for each load-balancing interval to simplify the hardware design. To respond to the balancing request, the donator SM #0 checks the entry with the maximum neighbor size in its ALWT, which is the second entry in this example. On this basis, the second entry of the ALWT in SM #0 is modified to move 448 edges to SM #1 at time **T2**. As a result, the start index of this entry is increased to 1752. The original start index (1304) and the new start index (1752) are copied into registers R_{BW0} and R_{BW1},

Table 2. Baseline GPU Microarchitectural Parameters

Parameter	Value
SM	980 MHz; 32 lanes; 15 SMs/GPU
Warp Scheduler	Greedy-Then-Oldest [43]; 4 Warp Schedulers/SMX; 8 Inst. Dispatch Units/SMX;
On-chip Storage	65536 Registers/SM; 48 KB Shared Memory; 16 KB L1D Cache; 1536KB L2 Cache;

respectively. Then, the control processor copies the two values into registers R_{BW0} and R_{BW1} of SM #1, and changes the state of register R_{BC} of SM #1 into *edge receiving* (ER). The EF exchange unit on SM #1 inserts a new entry in its ALWT with the start and end indices being the values of registers R_{BW0} and R_{BW1}, respectively. Accordingly, 448 new edges are offloaded to SM #1 without actually moving these edges.

For modern NVIDIA GPUs like Turing and Ampere GPUs, SMs are organized in hierarchy. That is, SMs are first organized into **Texture Processing Clusters (TPCs)** and then organized into **Graphics Processing Clusters (GPCs)**. For these GPUs, EF load balancing should be applied to GPCs not individual SMs. GraphPEG in each GPC is then responsible for distributing edge workload evenly between SMs within a GPC just like the raster engine of each GPC.

4 EXPERIMENTS AND EVALUATION

In this section, we present an experimental evaluation of GraphPEG. We initially describe the experimental setup, followed by studying the manner in which the architectural parameters of GraphPEG can be tuned for edge gathering and EF load balancing. Finally, we compare the performance of GraphPEG to other implementations and discuss the scalability and hardware cost of GraphPEG.

4.1 Experimental Setup

We used GPGPU-Sim 3.2.1 [3] to evaluate the GraphPEG. The simulator was modified and configured as shown in Table 2, which is our baseline architecture. The load imbalance monitoring interval is set to 512 cycles, and the communications between the control processor and SMs are modeled through the on-chip network.

To evaluate the GraphPEG, we selected various graph algorithms as benchmarks, namely, BFS, SSSP, BC, STCON, **graph coloring (GC)**, and **k-core decomposition (KC)**. For SSSP, we used the near-far method [8]. As previously discussed, these graph processing benchmarks make multiple iterations over the input graph. One difference between them is that the active vertex sets begin with one or a few vertices in the first four benchmarks (i.e., BFS, SSSP, BC, and STCON); whereas the active vertex sets in the last two benchmarks (i.e., GC and KC) begin with all vertices in the graph.

We selected graphs from the DIMACS implementation challenges [2] and the Stanford Large Data Collection [24]. Particularly, we considered datasets in different application domains: a product co-purchasing network (AMAZ), a community network (COM), a paper co-citation network (COP), a synthetic graph (KRON), a graph that has been used in numerical simulations (NACA), and a road network (USAC). Among the six datasets, USAC is the only graph with edge weights. To test the SSSP benchmark across all datasets, we generated random weights for the edges in the other five datasets. Table 3 lists the characterization of the graph datasets. Notably, the graph datasets show high variance in terms of graph size and out-degree.

Table 3. Dataset Characterization

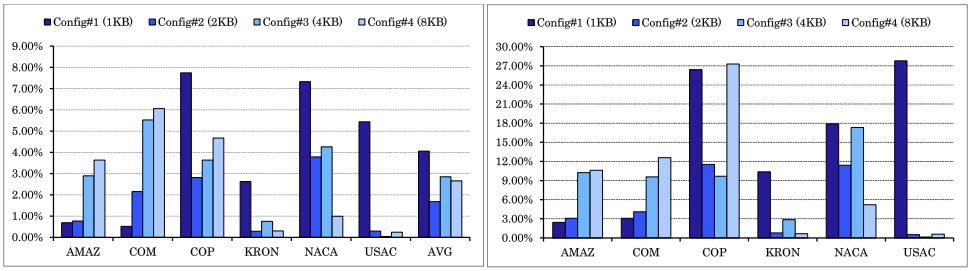
Dataset	Abbr.	Nodes	Edges	Max Degree	Average Degree
Amazon0601	AMAZ	0.4M	4.8M	2752	12
com-amazon	COM	0.3M	1.8M	549	6
coPapersCiteseer	COP	0.4M	32.0M	1188	80
kron_g500-logn17	KRON	0.1M	10.2M	29935	102
NACA0015	NACA	1M	5.2M	9	5.2
USA-road-d.CAL	USAC	1.8M	4.6M	7	2.6

4.2 Edge Gathering

We initially study the architectural parameters on edge gathering. Due to the excessive tests required to explore architectural parameters and long simulation time, for the parameter sensitivity tests, we ran simulations from the start of the graph processing for 4 million edges or until completion. We mainly focused on two architectural parameters, that is, the manner in which shared memory is split and the number of ALWT entries. To explore the best performance that GraphPEG can reach, we made GraphPEG utilize the whole shared memory. In the final overall performance evaluation subsection, we will evaluate the performance influence of different shared memory size configurations.

As described in previous section, shared memory is used by GraphPEG for gathered edges and vertices. The gathered edges will not be used again after being consumed. However, the vertices pushed into the output vertex-list are needed in the next iteration. Therefore, the increased shared memory space should be allocated to vertex-lists. To isolate the influences from other architectural parameters, we set the load-balancing overhead between different SMs to zero and set the number of ALWT entries to extremely large values, such that the ALWT would not become full. We considered four shared memory reservation configurations for gathered edges, that is, 1, 2, 4, and 8 KB. For each configuration, the remaining shared memory space was allocated to vertex-lists to explore the best performance of GraphPEG as described previously. We tested these configurations on the six graph algorithms across the six datasets. We found that no single configuration substantially outperformed other configurations (result is not shown due to space limit). To select a relatively better configuration, we define a relative performance metric, R_{gdn}, to evaluate the performance of configuration n with algorithm g across dataset d. The definition of R_{gdn} is $\frac{E_{gdb}-E_{gdn}}{E_{gdb}}$, where E_{gdb} denotes the best performance among the four configurations with algorithm g across dataset d, and E_{gdn} denotes the performance of configuration n with algorithm g across dataset d. R_{gdn} is used to measure the performance gap between configuration n and the best configuration (the smaller the better).

Figure 7(a) shows the average R_{gdn} on six graph algorithms across different datasets. We can see that no single configuration is superior than others for all the six graph algorithms. For example, although Configuration #1 (1 KB) outperforms others for AMAZ and COM datasets, it performs the worst for the other four datasets. On the one hand, if the reserved shared memory space for gathered edges is too small, then the edge processing procedure will be occasionally stalled due to running out of loaded edges. On the other hand, if reserving too large shared memory space for gathered edges, then additional memory requests for vertex spilling and refilling are required due to small shared memory space reserved for vertex-lists. Although allocating the shared memory dynamically may achieve better performance, for the purposes of simplifying the hardware design, we select a fixed configuration (Configuration #2) based on the average result in the last column of Figure 7(a). We leave the study on dynamical allocation in our future work.



(a) Average R_{gdn} (b) Max R_{gdn}

Fig. 7. Evaluation of different shared memory split configurations. R_{gdn} indicates the relative performance gap between configuration n and the best configuration b.

Table 4. Performance Results of an Eight-entry ALWT in Relation to an ALWT with Infinite Entries

	AMAZ	COM	COP	KRON	NACA	USAC
BFS	1.08	**1.10**	1.07	0.99	**1.34**	**1.91**
SSSP	1.04	1.05	1.01	0.99	1.04	**1.28**
BC	1.01	0.96	1.01	0.99	0.98	**1.16**
STCON	0.99	0.94	0.99	0.94	0.87	**1.11**
GC	0.96	1.01	1.02	1.01	0.82	**1.19**
KC	1.00	1.02	**1.22**	1.02	1.03	**1.48**

Figure 7(b) shows the maximum R_{gdn} on the six graph algorithms. We can see a similar trend from Figure 7(b). On average, Configurations #2, #3, and #4 outperform Configuration #1. However, Configuration #2 performs more stable, because its R_{gdn} does not exceed 12% in all cases. On this basis, for the following experiments, we reserved 2 and 46 KB of shared memory space for gathered edges and vertex-lists, respectively.

Next, we studied GraphPEG's sensitivity to the number of ALWT entries. The ALWT entries are consumed by the edge gathering unit when (1) loading the indices of adjacency lists into allocated entries in the ALWT and (2) loading edge data from the global memory based on ready entries of the ALWT. There is a trade-off between performance and hardware overhead. On the one hand, if the ALWT entries are consumed too fast and considerably very few entries exist in the ALWT, then the edge gathering unit may be frequently stalled. On the other hand, a large ALWT with excessive entries introduces a heavy hardware cost. Thus, we attempted to explore the number of ALWT entries required to achieve acceptable performance. We started with a small ALWT with eight entries. Table 4 lists the performance results of the eight-entry ALWT in relation to an ALWT with infinite entries (i.e., $\frac{Result_{ALWT\_inf}}{Result_{ALWT\_8}}$).

The data in Table 4 shows that an eight-entry ALWT is sufficient in most cases. Among the 36 tests, 27 tests have a relative performance differential below 10%. Some even show a better result than that of the ALWT with infinite entries. This is because for an ALWT with infinite entries, the edge gathering unit continuously issues index fetching requests, which may delay the processing of other memory requests. However, nine tests (indicated by the bold font) show a relative performance differential above 10%. The BFS-USAC test even shows a relative performance differential up to 91%. Further investigation reveals that these tests have abundant small outgoing degrees. Consequently, the ALWT entries usually have a small number of edges, which makes

Table 5. Performance Results of Nine Algorithm-dataset Pairs with Different ALWT Configurations in Relation to an ALWT with Infinite Entries

	12-entry	16-entry	20-entry	24-entry
BFS-COM	0.96	0.97	0.97	0.97
BFS-NACA	1.03	0.98	0.98	1.03
BFS-USAC	**1.35**	**1.12**	1.02	1.00
SSSP-USAC	**1.11**	1.05	1.02	1.01
BC-USAC	1.01	1.02	1.02	1.02
STCON-USAC	0.92	0.95	0.97	0.98
GC-USAC	0.95	0.91	0.90	0.92
KC-COP	**1.15**	**1.14**	**1.12**	1.09
KC-USAC	**1.14**	0.98	0.90	0.89

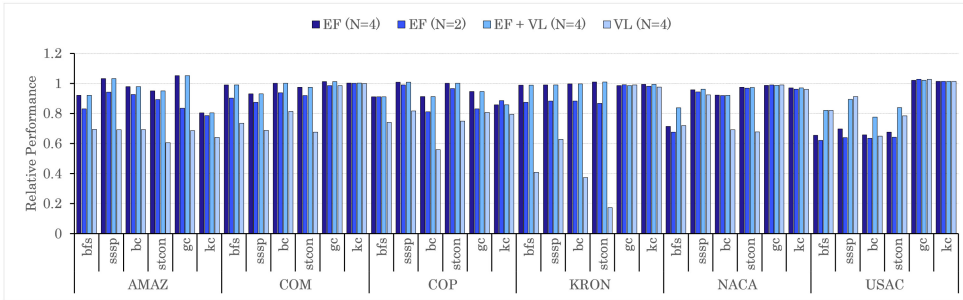


Fig. 8. Performance with different load-balancing configurations in relation to the ideal load balancing.

the edge gathering unit always hungry for ready ALWT entries. Hence, these algorithm-dataset pairs require a larger ALWT. We aimed to keep all the relative performance differentials below 10%. To achieve this, we tested four ALWT configurations (12-, 16-, 20-, and 24-entry) for the nine algorithm-dataset pairs that show a relative performance differential above 10% with the eight-entry ALWT configuration. Table 5 shows the experimental results.

We can see that the number of algorithm-dataset pairs that show a relative performance differential above 10% decreases with the increase of ALWT entries in Table 5. When the entries are increased to 24, all algorithm-dataset pairs have a relative performance differential below 10%. On this basis, we set the number of ALWT entries to 24.

4.3 EF Load Balancing

In this subsection, we studied the scheme of balancing workload between different SMs. Recall that GraphPEG considers two types of load balancing, that is, the vertex-list load balancing and EF load balancing. We focused on studying two load-balancing factors that may influence performance, that is, threshold and type. We used $R_{max} - R_{min} \geq \frac{R_{max}}{2^N}$ to define the load-balancing threshold, where R_{max} and R_{min} donate the maximum and minimum value of EF/vertex-list size in an SM, respectively, and N is the threshold configuration. $\frac{R_{max}}{2^N}$ is calculated by shifting R_{max} right by N bits.

We evaluated four load-balancing configurations. Figure 8 illustrates the performance results normalized to the ideal load balancing (no load-balancing overhead and the load-balance status is

checked every cycle). The first two are the EF load balancing with different balancing threshold configurations [$N = 4$ and $N = 2$]. The third configuration (EF+VL) is the combination of EF and vertex-list load balancing, and the fourth configuration (VL) is vertex-list load balancing alone. The balancing thresholds of the last two configurations are set to $N = 4$ (we saw minimal performance difference with balancing threshold configuration of $N = 2$). The vertex-list load balancing was implemented similarly as the EF load balancing, but with some differences. The first difference is that the vertex-list load balancing is performed on the output vertex-list. This approach is based on the fact that the output vertex-list becomes the input vertex-list in the following iteration. The second difference is that when the vertex-list load balancing occurs between two SMs, the number of offloaded vertices is exchanged between two SMs and these vertices are moved from the tail of one SM's local vertex-list to the tail of another SM's local vertex-list.

The results in Figure 8 show that in comparison with the ideal load balancing, the overhead of the realistic load balancing between different SMs is not very high, because the load balancing occurs infrequently, and the overhead is usually covered by edge processing. In some cases (e.g., the BFS benchmark and the USAC dataset), the overhead is relatively higher (affects overall performance), because fewer edges are processed per iteration.

Another observation of Figure 8 is that EF load balancing plays a major role in load balancing. In most cases, the performance declines with vertex-list load balancing instead of EF load balancing. For example, the relative performance drops to 17.33% for the STCON benchmark on the KRON dataset. In comparison, with EF load balancing and threshold configuration of $N = 4$, the relative performance is 101.03%. In some cases (e.g., some benchmarks on the USAC dataset and the BFS benchmark on the NACA dataset), the vertex-list load balancing shows better performance than EF load balancing. This is because the out-degree of each vertex in USAC and NACA is relatively small, which makes the edge gathering unit in each SM consume ALWT entries relatively fast. In this situation, GraphPEG may not be able to find free ALWT entries for offloading. When comparing EF with EF+VL, we found that the vertex-list load balancing does not contribute to overall performance in most cases. In-depth investigation reveals that the number of edges to be processed in each iteration is sufficiently large, and EF load balancing works well. On the basis of these observations and considering the hardware cost, the vertex-list load balancing is not applied in GraphPEG.

The performance of benchmarks GC and KC appears less affected by EF load balancing for some datasets. For the two benchmarks, only the first iteration was simulated, because the number of processed edges in the first iteration had already exceeded 4 million edges, and hence, the simulation was terminated. For these datasets, when the vertices are evenly distributed among the SMs, the edges are evenly distributed as well. In most cases, a lower threshold ($N = 4$) slightly outperforms a higher threshold ($N = 2$). On this basis, we used $N = 4$ as the default EF load-balancing threshold configuration for the following experiments.

4.4 Overall Performance Evaluation

In this subsection, we evaluated overall performance and compared GraphPEG with Gunrock [48], SEP-Graph [47], and the HWWL-accelerated [20] simple data-driven implementation. Gunrock and SEP-Graph were selected for comparisons, because both are representative high-performance graph processing frameworks on GPUs. Many optimizations for graph processing on GPUs have already been incorporated into the two graph processing frameworks. Therefore, graph applications written within these frameworks have comparable performance to that of hand-optimized implementations, sometimes even better. Specifically, Gunrock adopts Push for vertex updates and *load-balanced partitioning* as the workload-balancing strategy. SEP-Graph adopts a hybrid execution implementation. For scale-free graphs, it uses *cooperative blocks* as the load-balancing strategy

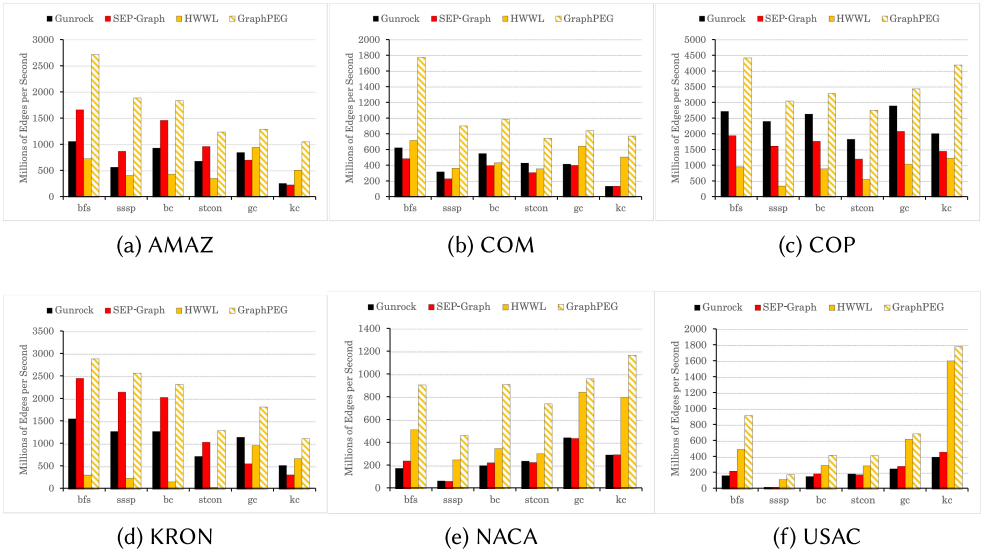


Fig. 9. Overall performance of different graph processing implementations.

and can switch between Push and Pull for vertex updates during execution. For sparse graphs, SEP-Graph always use the Push mode and adopts the simple data-driven implementation without using any load-balancing strategy. For the purpose of a fair comparison, GraphPEG adopts the same hybrid execution strategy as SEP-Graph (i.e., it switches between Push and Pull using the same strategy as SEP-Graph). It should be noted that for GC, KC and the backward stage of BC, the Pull mode cannot be used, because the vertex updates of source and destination vertices happen at the same time or the update of a source vertex depends on the values of its destination vertices. HWWL [20] is proposed for accelerating irregular algorithms, including graph algorithms on GPUs. We selected the HWWL-accelerated simple data-driven method as a hardware-based approach for comparison.

For overall performance evaluation, we ran each simulation from the start of the graph processing for 40 million edges or until completion. On the basis of the preceding the architectural parameter sensitivity study, GraphPEG was configured as follows: 2 KB shared memory for edge gathering, 24-entry ALWT, $N = 4$ as the EF load-balancing threshold configuration. To simplify the comparison with HWWL, the on-chip buffer for HWWL in each SM was configured as 48 KB, which is equal to the size of the shared memory, and the inter-SM load-balancing overhead was assumed to be zero. With this configuration, the HWWL used in our experiments should achieve better performance than the one used in the original work [20].

Figure 9 presents the performance of GraphPEG and three other software/hardware implementations for the six benchmarks on six input graph datasets in terms of millions of edges per second, and Figure 10 presents the performance speedup of GraphPEG over other three software/hardware implementations. The load-balancing overhead ($\frac{Cycles_{load-balance}}{Cycles_{total}}$) of Gunrock is also included for convenience of analysis.

As shown in Figure 9, the edge processing rates of various implementations are mainly determined by the average out-degrees of the input datasets. The first four datasets that have large average out-degrees (i.e., AMAZ, COM, COP, and KRON) show relatively higher edge processing rates than the last two datasets (i.e., NACA and USAC). The reason is that with a large number of edges to be processed in each iteration, the utilization of GPU's function units for the first four datasets are considerably higher; whereas the GPU's function units are often underutilized

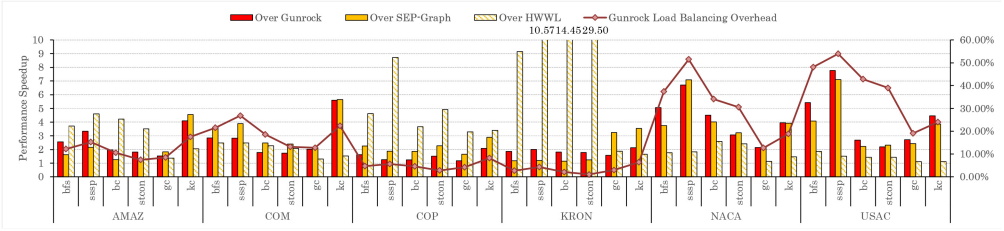


Fig. 10. GraphPEG performance speedup over Gunrock, SPE-Graph and HWWL; and the load-balancing overhead of Gunrock.

for the last two datasets. Another important factor that affects overall performance is memory divergence. The COP dataset has the highest edge processing rates for different benchmarks and implementations. Further investigation reveals that the neighbor indices of a vertex in COP are usually sequentially numbered; whereas the neighbor indices of a vertex in other datasets are usually randomly numbered. For example, suppose vertex #2 has five neighbors, its neighbor indices can be 3, 4, 5, 6, or 8 in COP. Therefore, COP is less likely to incur memory divergence, because the requests generated by COP are more likely to be coalesced due to sequential neighbor indices.

Figure 10 shows that in comparison with HWWL, GraphPEG can achieve an average speedup of 3.9×, a maximum of 29.5×, and 1.2× at minimum. The performance improvement of GraphPEG is largely a result of automatic edge gathering and workload balancing. This can be exemplified by the performance speedups of GraphPEG on the three datasets (i.e., AMAZ, COP, and KRON) that have highly skewed degree distributions. GraphPEG has higher performance speedups over the HWWL than Gunrock, which has a software-based EF load-balancing scheme. In some cases, the large number of processed edges in the first few iterations results in numerous uncoalesced memory requests. Thus, the speedups for GC and KC on the AMAZ and KRON datasets are exceptions due to memory divergences. The load-balancing items of HWWL are the vertices in the worklist. As previously stated, edges are the actual target items that should be balanced between threads for graph processing. As being unaware of this, HWWL even performs worse than a software-based simple data-driven implementation for some cases. Further investigation reveals that although HWWL balances the vertices of worklist between threads, it sometimes happens to move some vertices that have large out-degrees to the same SM, thereby causing severe performance degradation.

In comparison with Gunrock, which has built-in software-based EF load-balancing strategies, GraphPEG can achieve an average speedup of 2.8×, a maximum of 7.3×, and 1.18× at minimum. Several factors contribute to the performance improvements. First, as shown in Figure 10, the average load-balancing overhead of Gunrock is 18.03%, with a minimum overhead of 1.05% and a maximum overhead of 53.95%. Generally, the load-balancing overheads for datasets with small average out-degrees, such as NACA and USAC, are relatively high for Gunrock. However, as a hardware scheme, GraphPEG introduces low overhead to load balancing, which makes it outperform Gunrock on the datasets with small average out degrees. Second, in the cases where Gunrock has low load-balancing overheads, the performance improvement of GraphPEG comes from the automatic edge gathering scheme. GraphPEG automatically gathers edges and distributes them among threads with its hardware support, which greatly reduces the number of instructions used for graph traversal and load balancing. In comparison with Gunrock, GraphPEG reduces the number of executed instructions to approximately 1/8 on average. Therefore, GraphPEG can issue memory requests for graph traversal as early as possible to improve the utilization of memory bandwidth. Third, using shared memory as an on-chip buffer for vertex-lists and the

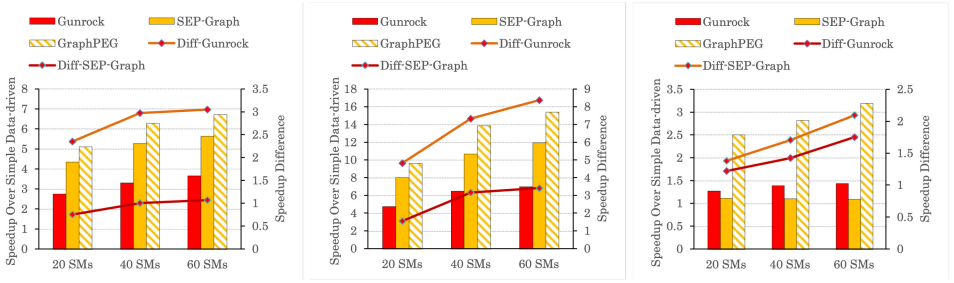
hardware-managed dynamic load-balancing scheme reduce the number of memory requests. However, the load-balancing strategies used in Gunrock usually need to preprocess the neighbor lists of the vertices in the input vertex-list in each iteration to distribute edges between threads, thereby increasing memory requests. On average, GraphPEG reduces 27.39% of the memory requests. One of the obstacles that limit the graph processing performance on GPUs is the large number of irregular memory accesses. GraphPEG alleviates this issue by issuing memory requests early and reducing the number of memory requests, which helps to improve graph processing performance on GPUs. Fourth, for scale-free graphs, GraphPEG benefits from the Pull strategy, which can reduce the write-write conflicts.

In comparison with SEP-Graph, which uses a hybrid execution implementation to adopt to different types of graphs, GraphPEG can achieve an average speedup of 2.5×, a maximum of 7.0×, and 1.14× at minimum. Once again, several factors contribute to the performance speedups. As previously described, when processing scale-free graphs, SEP-Graph adopts the *cooperative blocks* load-balancing strategy. The *cooperative blocks* load-balancing strategy only balances work within a thread block. However, it may introduce inter-block load imbalance due to unbalanced amount of work processed by different blocks, which can be shown by the experimental results that Gunrock performs better than SEP-Graph on GC and KC where the Pull mode cannot be used. The EF load-balancing scheme of GraphPEG can balance workload across thread blocks, thereby making GraphPEG also achieve better workload balance than SEP-Graph. For sparse graphs, the workload size of each iteration is usually small, GraphPEG can make vertex-list stay on shared memory, thereby reducing memory requests and improving performances. One interesting observation is that, Gunrock performs better than SEP-Graph on the COP dataset, although COP is a scale-free graph. The reason for this has been described previously, i.e., the neighbor indices of a vertex in COP are usually sequentially numbered, which results in less write-write conflicts than other datasets in Push mode. The hybrid execution mode of SEP-Graph has switching overheads, thereby making SEP-Graph performs worse than Gunrock on this dataset. In fact, GraphPEG can also achieve better performance on COP when the hybrid execution mode is turned off.

The above experimental evaluations were made with the whole shared memory being utilized by GraphPEG. As there can be situations that a graph application may use shared memory, we evaluated three maximal shared memory size configurations for GraphPEG, that is, 8, 16, and 32 KB. In comparison with the experimental results that GraphPEG can utilize the whole shared memory (i.e., the GraphPEG performance results of Figure 9, the average performance degradations are 6.57%, 2.74%, and 0.68%, respectively. We observed that the graph processing performance on the COP dataset is more susceptible to small shared memory size. Other datasets are less affected. The performance degradation for COP can be up to 18.36% on the BFS benchmark for the 8 KB shared memory configuration. For other datasets (exclude COP) on the 8 KB configuration, the performance degradations are all below 8%. As previously described, the COP dataset has the highest edge processing rates for different benchmarks due to less memory divergences than other datasets. Small shared memory size leads to more vertex spills and refills, thereby increasing the number of memory requests. For the NACA and USAC datasets on the BFS, SSSP, BC and STCON benchmarks, we observed no performance variations due to the fact that 8 KB is large enough to hold the vertices in the vertex-list for each iteration.

4.5 Scalability

To test the scalability of GraphPEG, we evaluated the normalized speedup (to a simple data-driven implementation) of Gunrock, SEP-Graph and GraphPEG with 20, 40, and 60 SMs. Due to limited space, we only present results of three benchmarks (BFS, SSSP, KC) across the KRON dataset. The results are shown in Figure 11. It is shown that GraphPEG scales well for all cases, while Gunrock



(a) BFS (b) SSSP (c) KC

Fig. 11. Scalability of Gunrock, SEP-Graph and GraphPEG.

and SEP-Graph scales well for BFS and SSSP except KC. Besides, GraphPEG outperforms Gunrock significantly with more SMs on three applications, and outperforms SEP-Graph significantly with more SMs on KC. This demonstrates that, in comparison with Gunrock and SEP-Graph, GraphPEG utilizes GPU resources more efficiently and incur less severe load-balancing overhead with increasing SMs.

4.6 Hardware Overhead

The primary hardware overhead of GraphPEG comes from the ALWT with an entry size of 98 bits. The storage requirement of a 24-entry ALWT is 98 bits $\times$ 24 = 294 bytes. In view of some additional control states, the total storage of GraphPEG in an SM is approximately 1 KB, which translates to an overhead of 0.39% of the register file per SM (256 KB). To further estimate the overhead, we model GraphPEG with the McPAT tool integrated into GPGPU-Sim. All models are based on a 28-nm technology. Results show that GraphPEG requires only 0.051 mm^2 of area per SM. Compared to the simulated GPU (426.90 mm^2), GraphPEG introduces approximately 0.18% more area, which is within reach of modern GPU architectures. Moreover, the latencies of the different building blocks of GraphPEG can reach to 1.5 GHz.

Also, we compared the total GPU energy of GraphPEG and Gunrock using GPUWattch, a GPU power analysis tool integrated in the GPGPU-Sim simulator. Results show that although GraphPEG consumes 11.4% and 14.8% more dynamic power than Gunrock and SEP-Graph, GraphPEG's total energy is only 52.6% and 69.3% of Gunrock and SEP-Graph, due to the significant performance improvement.

5 RELATED WORK

A large body of research on graph processing, including CPU- and GPU-based graph programming frameworks, as well as hardware-based graph processing acceleration, is available.

CPU-based Graph Programming Frameworks: Software parallel graph processing frameworks aim to provide high-level, programmable, and improved performance abstractions. The Boost Graph Library [12] is among the first efforts towards this goal. Pregel [28] is Google's effort at large-scale graph processing. GraphLab [26] is a graph processing library that relies on the shared memory abstraction and the GAS programming model. PowerGraph [11] is an improved version of GraphLab for power-law graphs. Galois [36] is an asynchronous task-based graph programming framework that supports priority scheduling and dynamic graphs.

GPU-based Graph Programming Frameworks: Several works target the construction of a high-level GPU graph processing library to deliver good performance and good programmability. Medusa [52], which uses a message-passing model, is a pioneering work on a high-level

GPU-based system for graph processing. VertexAPI2 [9] is the first GPU high-level graph processing framework that strictly follows the GAS model. MapGraph [10] and CuSha [19] also adopt the GAS abstraction. MapGraph integrates several load-balancing strategies to improve its performance. CuSha implements the parallel-sliding-window graph representation on the GPU to avoid non-coalesced memory access. Both Tigr [37] and GraphCage [7] are graph transformation frameworks that aim to reduce the irregularity of graphs for highly efficient graph processing on GPUs. The transformation of graphs usually introduce extra preprocessing overheads and increasing memory footprint. The work of this article focuses on the solutions without graph format transformation. Garaph [37] is a graph processing framework that utilizes both CPU and GPU to accelerate graph processing on a single machine. DiGraph [51] focuses on improving the performance of iterative directed graph processing on multiple GPUs. The work in this article focuses on improving graph processing performance on a single GPU. Gunrock [48] uses a high-level, bulk-synchronous, and data-centric programming model and integrates several state-of-the-art high-performance GPU computing primitives and optimization strategies. SEP-Graph is a recently proposed highly efficient GPU graph processing framework. It adopts a hybrid execution mode that can switch between Push and Pull during execution. Both Gunrock and SEP-Graph have comparable performance to the fastest GPU hardwired primitives. On this basis, we used Gunrock and SEP-Graph as the software-based graph processing implementations and compared GraphPEG with them.

Hardware-based and Architectural Support for Graph Processing Acceleration: A few FPGA-based accelerators are designed for specific graph processing algorithms, including SSSP [53], belief propagation [42], and PageRank [29]. Recently, some general-purpose graph analytics accelerators have been proposed, including GraphGen [38], GraphOps [39], Graphicionado [13], and the graph analytics accelerator architecture by Ozdal et al. [41]. Recently, Yan et al. proposed *GraphDynS* [49], which is a hardware/software co-design proposal that can achieve remarkable speedup than a GPU-based solution. In addition to specialized hardware accelerators, architectural support for CPU-based graph processing acceleration has also been proposed. Tesseract [1] and GraphPIM [34] are architectural support proposals for graph processing acceleration based on processing-in-memory system. Minnow [50] is a hardware engine augmented to each CMP core to provide hardware-managed worklist and worklist-directed prefetching for the Galois system to accelerate graph processing. Meanwhile, hardware-based support proposals for GPU-based graph processing acceleration are relatively few. Lakshminarayana et al. proposed spare register-aware prefetching for graph algorithms on GPUs [22], which detects target loads in hardware and injects instructions into the pipeline to prefetch data into spare registers. The load imbalance is not addressed; thus, its performance improvements for graph algorithms are limited in comparison with GraphPEG. HWWL [20] is designed for accelerating irregular algorithms on GPUs. As discussed in the previous section, GraphPEG outperforms HWWL by utilizing graph-aware techniques, such as automatic edge gathering and EF load balancing.

6 CONCLUSIONS AND FUTURE WORK

This article proposed a hardware architecture named GraphPEG to improve the performance of graph processing on GPUs through automatic edge processing and low overhead inter-SM load balancing. Compared with Gunrock and SEP-Graph, GraphPEG improves graph processing throughput by 2.8× and 2.5× on average, and up to 7.3× and 7.0×.

For our future work, there are two promising directions: (1) The first is combining graph format transformation techniques such as Tigr and GraphCage with GraphPEG to further improve graph processing performance; (2) The second is exploring how to accelerate graph processing in the environment of CPU-GPU and multi-GPU.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2016. A scalable processing-in-memory accelerator for parallel graph processing. *ACM SIGARCH Comput. Architect. News* 43, 3 (2016), 105–117.

[2] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. Graph partitioning and graph clustering, 10th DIMACS implementation challenge workshop. *Contemp. Math.* 588 (2013).

[3] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*. IEEE, 163–174.

[4] Jiri Barnat, Petr Bauch, Luboš Brim, and Milan Ceška. 2011. Computing strongly connected components in parallel on CUDA. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*. IEEE, 544–555.

[5] S. Baxter. 2013. Modern GPU library.

[6] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'17)*. Association for Computing Machinery, New York, NY, 235–248. DOI: https://doi.org/10.1145/3018743.3018756

[7] Xuhao Chen. 2019. GraphCage: Cache aware graph processing on GPUs. Retrieved from http://arxiv.org/abs/1904.02241.

[8] Andrew Alan Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of 28th International Parallel and Distributed Processing Symposium*. IEEE, 349–359.

[9] Erich Elsen and Vishal Vaidyanathan. 2014. VertexAPI2–A vertex-program API for large graph computations on the GPU. Retrieved from https://github.com/gunrock/vertexAPI2.

[10] 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 the Workshop on GRAph Data Management Experiences and Systems*. ACM, 1–6.

[11] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Vol. 12. 2.

[12] Douglas Gregor and Andrew Lumsdaine. 2005. The parallel BGL: A generic library for distributed graph computations. *Parallel Object-Orient. Sci. Comput.* 2 (2005), 1–18.

[13] Tae Jun 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 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–13.

[14] Pawan Harish and P. J. Narayanan. 2007. Accelerating large graph algorithms on the GPU using CUDA. In *Proceedings of the International Conference on High-performance Computing*. Springer, 197–208.

[15] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: A DSL for easy and efficient graph analysis. In *ACM SIGARCH Comput. Architect. News*, Vol. 40. ACM, 349–362.

[16] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 267–276.

[17] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. 2011. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. IEEE, 78–88.

[18] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. 2015. Scalable simd-efficient graph processing on gpus. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. IEEE, 39–50.

[19] 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*. ACM, 239–252.

[20] Ji Kim and Christopher Batten. 2014. Accelerating irregular algorithms on gpgpus using fine-grain hardware worklists. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 75–87.

[21] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. *ACM SIGARCH Comput. Architect. News* 35, 2 (2007), 162–173.

[22] Nagesh B. Lakshminarayana and Hyesoon Kim. 2014. Spare register aware prefetching for graph algorithms on GPUs. In *Proceedings of the IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. IEEE, 614–625.

[23] Junghee Lee, Chrysostomos Nicopoulos, Hyung Gyu Lee, Shreepad Panth, Sung Kyu Lim, and Jongman Kim. 2013. IsoNet: Hardware-based job queue management for many-core architectures. *IEEE Trans. Very Large Scale Integr. Syst.* 21, 6 (2013), 1080–1093.

[24] Jure Leskovec and Andrej Krevl. 2015. SNAP datasets: Stanford Large network dataset collection. Retrieved from https://snap.stanford.edu/index.html.

[25] Hang Liu and H. Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of theInternational Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. IEEE, 1–12.

[26] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E. Guestrin, and Joseph Hellerstein. 2014. Graphlab: A new framework for parallel machine learning. Retrieved from *https://arXiv:1408.2041*.

[27] Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*. ACM, 52–55.

[28] 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 ACM SIGMOD International Conference on Management of Data*. ACM, 135–146.

[29] Seamas McGettrick, Dermot Geraghty, and Ciaran McElroy. 2008. An FPGA architecture for the Pagerank eigenvector problem. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'08)*. IEEE, 523–526.

[30] Adam McLaughlin and David A. Bader. 2014. Scalable and high performance betweenness centrality on the GPU. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 572–583.

[31] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU implementation of inclusion-based points-to analysis. *ACM SIGPLAN Notices* 47, 8 (2012), 107–116.

[32] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 117–128.

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

[34] 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 the IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. IEEE, 457–468.

[35] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. 2013. Data-driven versus topology-driven irregular computations on GPUs. In *Proceedings of the IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS'13)*. IEEE, 463–474.

[36] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 456–471.

[37] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for GPU-friendly graph processing. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. Association for Computing Machinery, New York, NY, 622–636. DOI: https://doi.org/10.1145/3173162.3173180

[38] 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 the IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'14)*. IEEE, 25–28.

[39] Tayo Oguntebi and Kunle Olukotun. 2016. Graphops: A dataflow library for graph analytics acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 111–117.

[40] Marc S. Orr, Bradford M. Beckmann, Steven K. Reinhardt, and David A. Wood. 2014. Fine-grain task aggregation and coordination on GPUs. *ACM SIGARCH Comput. Architect. News* 42, 3 (2014), 181–192.

[41] Muhammet Mustafa 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 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. IEEE, 166–177.

[42] Jesús M. Pérez, Pablo Sánchez, and Marcos Martínez. 2009. High memory throughput FPGA architecture for high-definition Belief-Propagation stereo matching. In *Proceedings of the 3rd International Conference on Signals, Circuits and Systems (SCS'09)*. IEEE, 1–6.

[43] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-conscious wavefront scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 72–83.

[44] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible architectural support for fine-grain scheduling. In *ACM Sigplan Notices*, Vol. 45. ACM, 311–322.

[45] Jyothish Soman, Kothapalli Kishore, and P. J. Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*. IEEE, 1–8.

[46] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. 2009. Fast minimum spanning tree for large graphs on the GPU. In *Proceedings of the Conference on High Performance Graphics*. ACM, 167–171.

[47] Hao Wang, Liang Geng, Rubao Lee, Kaixi Hou, Yanfeng Zhang, and Xiaodong Zhang. 2019. SEP-graph: Finding shortest execution paths for graph processing under a hybrid framework on GPU. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. Association for Computing Machinery, New York, NY, 38–52. DOI: https://doi.org/10.1145/3293883.3295733

[48] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel et al. 2017. Gunrock: GPU graph analytics. *ACM Trans. Parallel Comput.* 4, 1 (2017), 3.

[49] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52)*. Association for Computing Machinery, New York, NY, 615–628. DOI: https://doi.org/10.1145/3352460.3358318

[50] Dan Zhang, Xiaoyu Ma, Michael Thomson, and Derek Chiou. 2018. Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 593–607.

[51] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery, New York, NY, 601–614. DOI: https://doi.org/10.1145/3297858.3304029

[52] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1543–1552.

[53] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Accelerating large-scale single-source shortest path on FPGA. In *Proceedings of International Parallel and Distributed Processing Symposium Workshop*. IEEE, 129–136.

Received June 2020; revised January 2021; accepted February 2021