Grus: Toward Unified-memory-efficient High-performance Graph Processing on GPU

PENGYU WANG, JING WANG, and CHAO LI, Shanghai Jiao Tong University, China
JIANZONG WANG, Ping An Technology
HAOJIN ZHU and MINYI GUO, Shanghai Jiao Tong University

Today's GPU graph processing frameworks face scalability and efficiency issues as the graph size exceeds GPU-dedicated memory limit. Although recent GPUs can over-subscribe memory with Unified Memory (UM), they incur significant overhead when handling graph-structured data. In addition, many popular processing frameworks suffer sub-optimal efficiency due to heavy atomic operations when tracking the active vertices. This article presents Grus, a novel system framework that allows GPU graph processing to stay competitive with the ever-growing graph complexity. Grus improves space efficiency through a UM trimming scheme tailored to the data access behaviors of graph workloads. It also uses a lightweight frontier structure to further reduce atomic operations. With easy-to-use interface that abstracts the above details, Grus shows up to 6.4× average speedup over the state-of-the-art in-memory GPU graph processing framework. It allows one to process large graphs of 5.5 billion edges in seconds with a single GPU.

CCS Concepts: • **Computer systems organization** → **Single instruction, multiple data**; **Heterogeneous (hybrid) systems**; • **Computing methodologies** → **Massively parallel algorithms**; • **Mathematics of computing** → **Graph algorithms**;

Additional Key Words and Phrases: Graph processing, GPU, unified virtual memory

ACM Reference format:
Pengyu Wang, Jing Wang, Chao Li, Jianzong Wang, Haojin Zhu, and Minyi Guo. 2021. Grus: Toward Unified-memory-efficient High-performance Graph Processing on GPU. *ACM Trans. Archit. Code Optim.* 18, 2, Article 22 (February 2021), 25 pages.
https://doi.org/10.1145/3444844

1 INTRODUCTION

Processing large-scale graph data is a fundamental procedure of machine learning, data mining, scientific computing, and other important areas. The continued growth of graph-structured data today creates a crucial need for fast, scalable computing on parallel architectures that have limited

New article, not an extension of a conference paper.
This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003503) and the National Natural Science Foundation of China (Grant No. 61972247).
Authors' addresses: P. Wang, J. Wang, and C. Li (corresponding author), Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China, 200240; emails: {wpybtw, jing618}@sjtu.edu.cn, lichao@cs.sjtu.edu.cn; J. Wang, Ping An Technology, 20 Keji South 12th Road, Shenzhen, Guangdong, China, 581000; email: wangjianzong347@pingan.com.cn; H. Zhu and M. Guo, Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai, China, 200240; emails: {zhu-hj, guo-my}@cs.sjtu.edu.cn.

1544-3566/2021/02-ART22
https://doi.org/10.1145/3444844

resources. Due to their massive parallelism and high-bandwidth memory access, GPUs have become popular accelerators for modern graph applications. Nevertheless, the available GPU memory capacity is often limited to tens of gigabytes even for some high-end data-center-class GPUs [45]. As graph scales, it can easily outgrow the memory capacity of the GPU.

Traditionally, graph processing framework on a GPU can only access data reside in its device memory. It is programmers' responsibility to manually manage data location, which makes it very difficult to achieve efficiency while ensuring correctness. To solve this issue, vendors have proposed new memory management models. For example, Compute Express Link [51] aims to create a coherent memory pool. NVIDIA [46] and AMD [12] separately introduced the idea of Unified Memory (UM), which defines a common coherent memory space for all processors. With UM, one can over-subscribe GPU memory [49] and access the virtual memory of the CPU. It relies on GPU driver and hardware to automatically control data transfer, thus relieving programmers from manually moving data. In other words, UM allows GPU programs, including graph applications, to conveniently process larger dataset with minor code changes.

Over-subscribing memory resources for graph processing on GPU is not as simple as it seems. There has been prior work [33] showing that naïvely adopting UM brings significant overhead even for in-memory computing workloads. When the working set becomes larger than the GPU memory, pages might thrash between the CPU and GPU, resulting in even higher overhead. CUDA 8.0 has introduced several *data usage hints* [49] that provide certain guidelines for efficient page migration of UM. However, existing GPU graph processing frameworks can hardly benefit form it due to the highly diversified, irregular data access patterns of graph workloads.

To make the best use of the unified memory resource, a graph processing framework must balance space efficiency and execution efficiency. Many prior arts leverage space-inefficient graph format or auxiliary data to assist load balancing and memory coalescing. For example, Cusha [28] introduces two ordered edge-centric formats (G-Shards and Concatenated Windows (CW)) for coalesced memory access. SEP-graph [60] adopts a Push-Pull direction switch optimization, which needs to store both compressed sparse row (CSR) and compressed sparse column (CSC) formats for one graph. These optimizations have doubled the GPU memory consumption, leading to severely limited processing ability (i.e., low scalability) as the size of the graph grows.

In this work, we thoroughly analyze the performance of graph processing on UM-enabled GPUs. We mainly focus on the scenario where the graph datasets are larger than the GPU memory capacity while they can still fit into the main memory. We characterize memory-hungry graph processing applications with UM in the simulated GPU-sharing environment. Our characterization shows several noteworthy patterns and insights. We revisit different layers of the existing GPU graph processing frameworks and propose **Grus**, an elaborately re-designed framework that aims to unleash the performance potential of CPU-GPU architecture.

The key idea behind Grus is to jointly manage the unified memory and atomic operations. In other words, our framework has two wings—one is UM optimization and the other is execution optimization. Not until both wings are equally developed can Grus achieves its best performance. We have devised a special computing abstraction (the body) for Grus: *Prepare-Update-Generate*. It provides a user-friendly interface. Users can write graph applications with tens of lines of code.

Our major contributions are summarized as follows:

- We characterize graph processing workloads on UM-enabled GPUs. We discuss the benefits and limitations of several key UM performance tuning methods.
- We propose a tailored UM management scheme (Grus' left wing). Our design can slim the memory demand for space efficiency, which brings significant benefits for large-scale graph processing in our experiment. We also leverage an adaptive UM management policy to supervise UM page movement between the CPU and GPU.

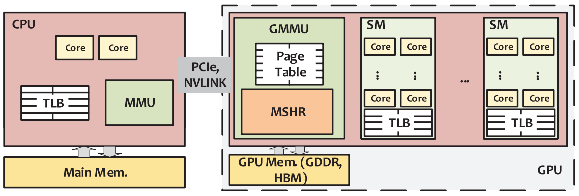


Fig. 1. A simplified CPU-GPU memory architecture that supports Unified-memory page migration.

- We propose a lightweight kernel execution module (Grus' right wing). Grus uses a novel *bitmap-directed frontier* (BDF) structure. It aims to replace part of the atomic operations with low-cost memory write operations. Meanwhile, Grus leverages a lightweight execution model for load balancing.
- We abstract Grus into an easy-to-use interface and experiment with it heavily. We show that Grus achieves up to 6.4× average speedup over several state-of-the-art in-memory graph frameworks, and up to 5×, 19× speedup over two out-of-GPU-memory graph frameworks. We also perform a detailed evaluation to analyze the performance bottleneck.

The rest of this article is organized as follows. Section 2 introduces the background. Section 3 characterizes the performance implications of UM. Section 4 proposes Grus, including UM management, runtime optimization, and programming interface. Section 5 introduces the experimental methodologies. Section 6 presents the results and analysis. Section 7 discusses related work and Section 8 concludes this article.

2 BACKGROUND

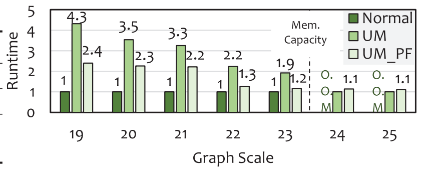
UM defines a common coherent memory space for all processors to simplify the usage of memory. We use NVIDIA's terminology to describe GPU architecture in this article, while most of the ideas apply to GPUs from other vendors as well. NVIDIA supports UM since CUDA 6.0, and terms it Unified Memory. It allows data to migrate across the main memory and GPU memory automatically, via hardware Page Migration Engine for Pascal architecture and newer GPUs.

UM provides a convenient interface to relieve the users of manually moving data from the host main memory to the GPU memory. Figure 1 depicts a simplified CPU-GPU memory architecture supporting UM. When a processor (CPU or GPU) accesses pages that are not in its memory, it triggers page faults and stalls the executions of threads until the requested pages have migrated. GPUs have multiple levels of *translation lookaside buffer* (TLB). The system uses a set of Miss Status Handling Registers (MSHRs) to record the page faults. Page fault locks the TLB for the corresponding streaming multiprocessor (SM), and any new translations of this TLB will be stalled until all faults are resolved. If multiple fault messages are generated for the same page, then the UM driver will process these faults, remove duplicates, updates mappings and transfer the data. The migrated page size ranges from 4 KB to 2 MB (varies on different systems). According to limited open resources [56], Pascal GPUs leverage a tree-based hardware prefetcher to assist page migration. Ganguly et al. [17] deduced that the GPU Memory Management Unit (GMMU) determines the migrated page sizes based on the requested data size and shape in each 2 MB large page. Even further, Pascal or newer architecture GPUs support 49-bit virtual addressing, which covers the 48-bit virtual address spaces of modern CPUs. This allows a GPU to access all the virtual memory of the entire system.

A major limitation of UM is that it does not eliminate the latency overhead between CPU and GPU. Landaverde et al. [33] investigated the performance of UM on the Rodinia Benchmark suite [9] and found that the performance overheads are significant for the majority of applications.

	Memory allocation	Data movement
`Normal`	`cudaMalloc`	`cudaMemcpy`
`UM`	`cudaMallocManaged`	on-demand migration
`UM_PF`	`cudaMallocManaged`	`cudaMemPrefetchAsync`, on-demand migration

(a) Configuration details.



(b) The normalized elapsed time (including data transfer time). O.O.M indicates that GPU runs out of memory.

Fig. 2. Evaluation on BFS of Tigr [54] with different GPU memory configurations. The graph of scale x has 2^x vertices and 2^{x+4} edges, generated with PaRMAT [27]. We limit the available GPU memory capacity to 1 GB to investigate GPU memory over-subscription.

The overhead of UM comes from fault handling and transfer latency. The fault handling overhead latency is around 45 μs [17]. There is a constant activation overhead for every transaction through the PCI-e interface regardless of the transfer size. Accessing large non-resident memory could result in large numbers of page faults, bringing significant overhead.

3 ANALYZING GRAPH PROCESSING WITH UM

In this section, we characterize UM performance under graph processing workload and evaluate the effectiveness of several UM tuning options.

3.1 Impact of Unified Memory

To understand the impact of UM on graph processing, we investigate Tigr [54], a state-of-the-art GPU-based graph processing framework. As shown in Figure 2(a), we mainly consider three different memory management strategies. Denoted as `Normal` in this article, the original Tigr system uses basic memory allocation strategy, namely, explicit CPU-GPU memory copy. To prevent CPU-side data paging-out and achieve the best transfer rate, we pin graph data in the main memory. We also modify Tigr and evaluate two UM-enabled configurations denoted as `UM` and `UM_PF`, respectively. For these two configurations, Tigr can over-subscribe the GPU memory. `UM_PF` indicates UM prefetching is invoked on memory region storing graph data. In this case, data in the UM space can explicitly migrate to the physical memory of the desired device immediately.

Figure 2(b) shows our evaluation results. We present the normalized execution time of Breadth First Search (BFS) on Tigr with different configurations. The graph sizes are smaller than the GPU memory capacity for graphs that have a scale less than 23. We also consider graphs of larger scales (24~25); in this case the system must over-subscribe GPU memory. From the figure, we can see that `Normal` always yield the best performance for graphs of moderate scales (19~23). `UM` takes 1.9~4.3× time, compared to `Normal`. This is because the page fault overhead and PCI-e interface overhead dominate when graph data size is relatively small. When processing a bigger graph, the large working set partially amortizes those overheads. The elapsed time of `UM_PF` becomes closer to `Normal` (e.g., 22 and 23).

An important observation is that `UM_PF` takes about 2.3× execution time compared to `Normal` for graphs of small sizes (i.e., 19 to 21); the measured prefetching throughput is also low. This indicates that prefetching small size of data does suffer from overheads. When the data size becomes larger, UM prefetching achieves similar throughput compared with explicit memory copy. Finally, as for graphs of very large scale (24 and 25), `Normal` fails to process them, since the GPU cannot allocate data larger than the available memory capacity. `UM_PF` is slightly slower than `UM`. This is because a naïve prefetching scheme will bring in graph data as much as possible, and it can exhaust idle

GPU memory space. This results in additional page evictions to CPU before pages are migrated to GPU.

Summary: different kinds of memory allocation methods can achieve entirely different performance. Specifically: (1) Normal allocation achieves the best performance if data can fit into GPU memory. (2) Adopting UM allows one to process large graphs at the cost of performance degradation, depending on the data size. (3) UM may benefit from prefetching due to the reduction of page fault when data can fit into GPU memory. (4) Prefetching could harm UM performance if the GPU memory is over-subscribed.

3.2 Implications of Unified-memory Hints

CUDA introduces several APIs to provide memory usage hints for the runtime since version 8.0. Programmers can specify memory hints through `cudaMemAdvise`. These hints are expected to guide the system driver so that the memory access performance can be improved.

- `cudaMemAdviseSetAccessedBy` (AB) indicates that the data will always be mapped into a specified processor's page tables if possible. When the data is migrated, the mapping will be updated accordingly.
- `cudaMemAdviseSetPreferredLocation` (PL) sets the preferred location (CPU or GPU) for a range of data. If another processor wants to access the memory region, then the data will not migrate from the preferred location.
- `cudaMemAdviseSetReadMostly` (RM) implies that the data will be mostly read and occasionally written to. It lets the driver create read-only copies of that data, and then send copies to other processors rather than migrate.

If applications' data access patterns are known ahead, then the programmer may choose one of the memory hints to optimize performance. However, it is not straightforward to choose appropriate memory hint of graph applications.

To evaluate the optimization effectiveness of memory hints, we evaluate four algorithms on three graphs. These evaluated algorithms are implemented in a queue-based frontier execution style with the warp-centric load-balancing policy on CSR formatted graphs. An array stores the corresponding vertex labels (distance from the source for SSSP, for instance). We set the same hints on graph data, frontier and label array for each configuration. On the same memory region, at most one hint can take effect. Hints can be used along with prefetching, thus we also combine these hints with UM prefetching. We provide detailed evaluation methodology in Section 5.

In Table 1, we show the normalized elapsed time of graph workloads under different UM configurations. One of the key observations is that: *prefetching is good if all the data can fit into the GPU memory, and it almost always brings overhead when the GPU memory is over-subscribed.* This is easy to understand as prefetching makes all data resident on GPU for relatively small graphs, thus the following processing suffers no overhead of fault handling and page migration. However, if the GPU memory is over-subscribed, prefetching the whole range of data may result in a situation that data to be used in the near future cannot reside on GPU due to the indiscriminate prefetching. In other words, PF may aggravate the thrashing. `AB` greatly contribute to performance improvement when over-subscription exists. For example, it can reduce the normalized elapsed time except for BFS on the UK-union dataset. In the meantime, `AB` also increases elapsed time on some graph especially for connected components (CC) on SK-2005 without over-subscription. Differently, `PL` has a minor impact on performance on all of the graphs and algorithms, compared with other hints. `RM` improves performance at a certain level as long as over-subscription exists, but not as much as `AB`. When combined with prefetching, `PL` shows little impact and it has similar performance with

Table 1. Normalized Elapsed Time Under Different Memory Hints

Dataset	Algorithm	Over-sub. Rate	Evaluated UM Configurations							
			UM	PF	AB	PL	RM	PF+AB	PF+PL	PF+RM
Friendster	BFS	-	1.0	**0.62**	1.02	1.00	0.98	0.70	0.63	**0.62**
	CC	-	1.0	**0.51**	0.65	1.02	1.03	0.55	0.53	0.52
	PR	-	1.0	**0.84**	0.85	1.00	1.01	0.87	0.85	0.85
	SSSP	**1.4**	1.0	1.02	**0.15**	0.97	0.68	0.16	0.98	0.66
SK-2005	BFS	-	1.0	**0.55**	1.11	1.08	1.03	0.66	**0.55**	0.56
	CC	-	1.0	**0.43**	2.26	1.01	1.07	0.50	0.45	0.44
	PR	-	1.0	**0.51**	1.29	0.99	1.03	0.56	0.53	0.53
	SSSP	**1.3**	1.0	1.00	0.28	0.96	0.65	**0.13**	1.01	0.67
UK-union	BFS	**2.1**	1.0	1.49	0.99	1.02	0.80	**0.78**	1.52	1.26
	CC	**2.1**	1.0	1.11	**0.25**	0.93	0.71	0.34	1.12	0.80
	PR	**2.1**	1.0	1.05	**0.18**	1.00	0.70	0.20	1.05	0.75
	SSSP	**3.9**	1.0	1.02	**0.15**	0.96	0.61	0.21	1.01	0.70
Geometric mean	-	-	1.0	0.79	0.53	0.99	0.84	**0.40**	0.80	0.67

UM is the UM baseline. PF is with prefetching. AB is for `cudaMemAdviseSetAccessedBy`. PL is for `cudaMemAdviseSetPreferredLocation`. RM is for `cudaMemAdviseSetReadMostly`. *Over-sub. rate* is the ratio between the dataset size and the available memory capacity when GPU memory is over-subscribed.

PF. This is the same for RM for in-memory scenarios. For over-subscribed scenarios, PF+RM is better than PF, but is not as good as AB or PF+AB.

Insights: These hints have significantly different impacts on the performance. AB maps pages into GPU's page table, which greatly reduces the overhead of accessing absent pages. It brings significant speedup for all over-subscription and some in-memory experiments. This indicates that mapping data to GPU's page table in advance to reduce the fault handling overhead is crucial for performance if we cannot prefetch all the data to the GPU memory. RM shows moderate speedup for over-subscription experiments as the pages storing read-only graph structure data need not to be transferred back to the main memory when they are evicted. However, as only one GPU accesses those data during these experiments, there are no other processors (GPUs or CPUs) trying to access those data. Thus, PL shows little impact on the performance compared with UM.

In summary, PF, AB, and PF+AB are more likely to yield better performance. The runtime are strongly correlated with memory over-subscription status as well. If there is enough GPU memory, then prefetching all data into GPU ahead is good for performance. When the GPU memory is over-subscribed, setting data region as AB or AB combining with prefetching may achieve the best performance. Based on these observations, we present our UM management strategy in Section 4.1.2.

4 THE GRUS FRAMEWORK

From the above analysis, we can see that designing UM-efficient, high-performance graph computing system can be a non-trivial task. In this section, we introduce how Grus strives to achieve this goal with a novel, multifaceted approach.

In Figure 3, we depict the system architecture of Grus. On top of the CPU-GPU heterogeneous computing environment, Grus presents two equally important modules: an UM management module (i.e., the left wing of Grus) and an execution optimization module (i.e., the right wing of Grus). The UM management module is to reduce UM overheads especially when the memory is over-subscribed. It includes space-efficient data structures to slim memory usage and an adaptive UM policy to reduce the overhead caused by UM. The execution optimization is responsible for effectively mapping graph processing operations to the GPU hardware at runtime. It includes a novel frontier structure to track active vertices with low overhead and a load-balancing strategy for efficient GPU execution. Eventually, Grus abstracts the two modules into a *Prepare-Update-Generate*

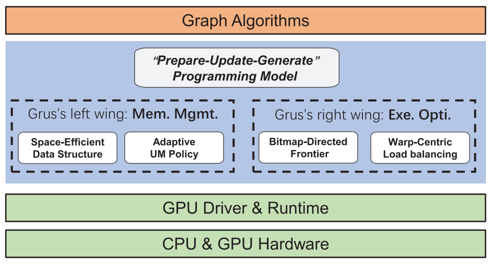

Graph Algorithms
"Prepare-Update-Generate" Programming Model
Grus's left wing: Mem. Mgmt.
Space-Efficient Data Structure
Adaptive UM Policy
Grus's right wing: Exe. Opti.
Bitmap-Directed Frontier
Warp-Centric Load balancing
GPU Driver & Runtime
CPU & GPU Hardware

Fig. 3. Grus system architecture.

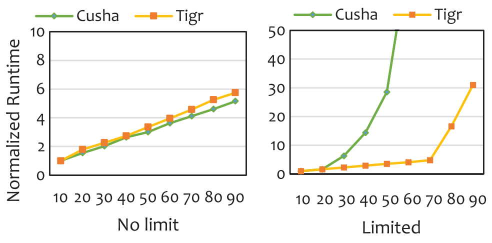


Fig. 4. BFS runtime of two frameworks for graphs with varied sizes. The X-axis is graph size in million-edges. `Limited` indicates there is a constant GPU memory limit. Results are normalized to their runtime for graph size 10, respectively.

interface. With this interface, a graph processing application can be easily implemented in a few lines of code with high efficiency.

4.1 Grus' Left Wing: Unified-memory Management

Large graph processing is known as memory-hungry. It features massive, fine-grained memory access. Minimizing the overhead brought by unified memory is the top priority of our design. In addition to this, Grus also tries to reduce the pre-processing and data-transfer overhead that are largely overlooked in prior works.

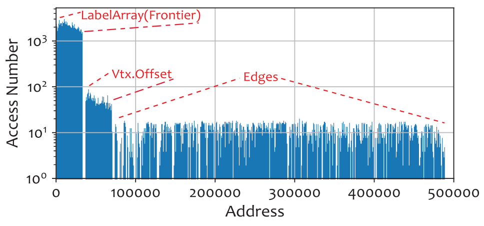
Grus intends to optimize UM in three perspectives: (1) *Minimizing the amount of migrated data.* Grus reduces memory footprint by using space-efficient graph format and eliminating unnecessary auxiliary data. In this way, fewer UM pages will migrate under the limited GPU memory capacity. (2) *Reducing the number of page faults.* Grus selectively prefetches graph data to the GPU so that the high-priority data (or all data if possible) are resident on the GPU to reduce the first-access latency. (3) *Reducing page migration overhead.* When the GPU memory is over-subscribed, Grus maps all the graph data in the GPU page table with the `AccessedBy` hint at the beginning. As a result, GPU can directly access the absent pages and evict pages to the main memory without suffering from the fault handling overhead.

4.1.1 Memory Usage Trimming. To illustrate why space efficiency is important for GPU graph processing with UM, we evaluate the performance of Cusha [28] and Tigr [54], two representative frameworks, using edge- and vertex-centric graph format, respectively. We modify them to enable over-subscription in the context of UM. Cusha proposes two edge-centric graph formats for coalesced memory access. Its G-Shard format represents each edge with a 4-tuple. Tigr focuses on load-balancing, and it leverages a virtually transformed graph format similar to compressed sparse row (CSR). Thus, Cusha takes more than double memory to store the same graph compared to Tigr.

As Figure 4 shows, the evaluated two frameworks have similar runtime trend: It grows linearly if there is no memory limit. When the GPU memory limit is set, the runtimes of both Cusha and Tigr may grow super-linearly at a certain time. This is because they start to over-subscribe GPU memory and suffers from the UM page thrashing severely when the memory consumption is larger than the available memory capacity. The difference is that Cusha starts to over-subscribe from graph with 30 million edges while Tigr starts to over-subscribe from graph with 80 million edges as Cusha needs more memory to process the same graph compared with Tigr. These two frameworks take comparable runtime for small graphs (with 10 or 20 million edges). But when

Name	Size	R or W?	Access Pattern
VertexOffset	$\|V\|$	R	Random, fine-grained
Edge	$\|E\|$	R	Random, varied-sized
VertexProperty	$\|V\|$	R/W	Random, fine-grained
Frontier	$\|V\|$	R/W	Fine-grained

(a) Property of main data structures. 'R' and 'W' indicates that data will be read or written, respectively.



(b) The addresses and access number of memory transactions. Y-axis is in log-scale.

Fig. 5. Access pattern of different data structures for BFS.

Cusha starts to over-subscribe GPU memory, their performance gap becomes significantly large. Therefore, for two comparable graph processing framework designs, the space-efficient one has significant performance advantages for large graph processing with UM over-subscription.

Some graph processing frameworks are not entirely space-efficient even though they do use CSR (or other compressed formats) to store graph. For example, SEP-Graph [60] stores both CSC and CSR format on GPU to enable their Push-Pull direction switch optimization. Through this method, the execution time on GPU is reduced at the cost of twice of the GPU memory requirement. It is justifiable for processing smaller graphs, which takes less than half of the GPU memory, whereas limits the capacity of processing large graphs more severely. Transferring twice of graph memory makes it even worse in terms of the total runtime.

Grus prioritizes space-efficiency in its implementation. It leverages CSR as the underlying graph structure. Besides this, Grus reduces memory consumption for other runtime data structures. For example, SIMD-X [36] leverages a just-in-time task management approach for load-balancing. In its implementation, it reserves more than $4 * 18|V|$ bytes memory for its frontiers in case of frontiers overflow, where $|V|$ is the vertex number and vertex indices are in 4-byte format. However, *bitmap-directed frontier* of Grus only needs less than $5|V|$ bytes of memory. Doing so allows Grus to improve space-efficiency and reduce the overhead resulted from UM over-subscription. Meanwhile, Grus enjoys less data transfer overhead from the main memory to the GPU memory.

4.1.2 Adaptive UM Policy. A typical graph processing application involves three type of data: graph data, objective property data, and runtime data storing active information (frontiers). When using CSR as the graph format, graph data includes vertex offset and edge. The objective property data refers to distance values from the source vertex for BFS, or rank values of vertices for PageRank. These data structures have different memory access patterns, summarized in Figure 5(a).

We characterize a BFS application from the SHOC Benchmark Suit [11] on a real-world graph and collect the memory trace with a GPU simulator [58]. This BFS application uses CSR graph format and warp-centric execution, similar to Grus. As Figure 5(b) shows, the pages of different memory regions have vastly different access numbers. The property array also acts as the frontier to indicates active vertices in their implication. It is most frequently accessed. The vertex offset data is second frequently accessed. The edge data is relatively least frequently accessed. The access numbers of the edge data regions also vary.

Based on the above observations, Grus manages the data structures with priorities. The frontiers and objective property array are most frequently accessed and updated, hence they are given the highest priority. Since the vertex index array of CSR is less frequently accessed and read-only, it

ALGORITHM 1: Priority-based memory management algorithm

```
Function SetMemPolicy():
    GPUIsFull=false;
    availGPUMemSize = getAvailGPUMemoryCapacity();
    for Data in {VertexProperty, VertexOffset, Frontier, Edge} do
        if not GPUIsFull then
            if $Data.Size < availGPUMemSize$ then
                UnifiedMemPrefetchToGPU($Data$, $Data.size$);
                availGPUMemSize -= $Data.size$;
            else
                GPUIsFull=true;
                SetUnifedMemAdvise($Data$, AccessedBy);
                UnifiedMemPrefetchToGPU($Data$, $\tau * availGPUMemSize$);
            end
        else
            SetUnifedMemAdvise($Data$, AccessedBy);
        end
    end
```

has medium priority. The edge array (and the weight array) of CSR is least frequently accessed and large in size; hence, we assign it the lowest priority.

As prefetching UM achieves close bandwidth with explicit memory copy for large memory regions, we allocate CSR representation of graph dataset, property array and the frontier in UM for all graphs. Once Grus has loaded the graph from the storage, it checks the current available GPU memory capacity and chooses the corresponding policy for data structures one by one in order of priority. If there is enough space for the target structure, then Grus prefetches the entire memory region. Otherwise, Grus prefetches the data structure partially to occupy part of the memory, leaving a portion of the memory available. We denote the prefetching ratio as τ. Prefetching data with a threshold τ rather than exhausting the GPU memory is to prevent the evictions resulted from accessing non-resident data while GPU memory is full at the beginning. Graph algorithms can be divided into two categories. The first category includes the traversal algorithms that one source vertex is active at the first iteration (e.g., BFS and SSSP). The second category includes algorithms that all vertices are active at the start (e.g., CC and PageRank), denoted full-active algorithm. For traversal algorithms, Grus prefetches edge data chunk containing edges of the source vertex with a relatively small τ as the actual traversal is not known ahead. For full-active algorithms, Grus prefetches edge data chunk from the start with a relatively large τ as all edges need to be processed in the first iteration. Empirically, we set τ as 0.5 and 0.8 for traversal algorithms and full-active algorithms in this article, respectively. We provide a validation for this in Section 6.3. After that. Grus sets the `AccessedBy` hint for it and the remaining structures with lower priorities. Algorithm 1 shows the priority-based UM management algorithm.

Note that the latest official CUDA (version 10.2) cannot successfully allocate of UM larger than the physical capacity of the main memory yet. This is because UM pages are pinned on the X86 platform. As a proof-of-concept prototype implementation, Grus assumes that the graph data can fit into the main memory at present. There are researches on adding support to map data on NVM SSD to the Unified Memory space (similiar to `mmap`) [7, 40]. Therefore, Grus can be extended to process graph larger than physical memory capacity combining with those works.

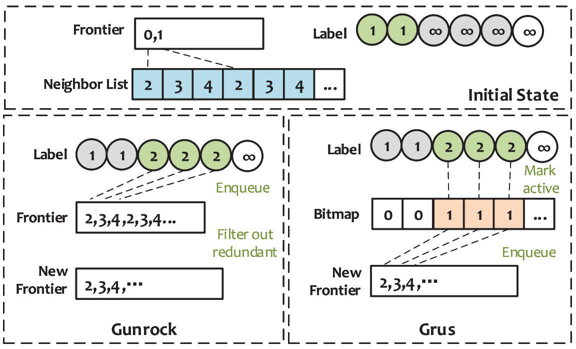


Fig. 6. One iteration of the BFS algorithm in different styles (Gunrock style or our BDF).

4.2 Grus' Right Wing: Execution Optimization

4.2.1 Bitmap-Directed Frontier. In the context of graph processing, the frontier structure has a remarkable impact on performance. There have been several papers [36, 42] investigating the performance of different frontier structures: *Bitmap*, *Unsorted Queue*, *Sorted Queue*, and so on. Bitmap frontier is relatively lightweight as it simply invokes kernel on all vertices and then inspects the active state of vertices. It is inefficient when the majority of vertices are not active. Queue-based frontiers only assign threads to process active vertices. However, enqueuing vertices to the frontier often involves expensive atomic operations. These frontiers have performance advantages in certain scenarios. For example, Gunrock [63] leverages a queue-based frontier. During processing, its *Advance* operation enqueues vertices to the *frontier*. Then, its *Filter* operation filters out undesired items in the frontier. Both *Advance* and *Filter* require a large number of expensive atomic operations, which need to be carefully trimmed to accommodate more computing tasks. Even though there has been prior work [36] on using reduction operations instead of atomic operations, it heavily relies on block-level synchronization and software global barrier, which also introduce extra overhead.

To overcome the shortcomings of previous frontier structures, we introduce a novel lightweight *BDF* structure. This structure aims to utilize low-cost memory write operation to reduce expensive atomic operations. It consists of a queue-based frontier and a bitmap. As GPU threads cannot concurrently write to adjacent bits without involving atomic operations while preserving correctness, the bitmap uses a byte for each vertex to indicate whether it has been updated in the current iteration or not. When Grus updates the labels of some vertices, it writes the byte corresponding to these vertices to 1, indicating active. After processing all the active vertices, Grus computes the new queue by scanning over the bitmap, and resets the bitmap to 0s.

Note that the space overhead of using one byte rather than one bit for each vertices is negligible in terms of the total memory consumption of graph algorithms as the number of vertices is much smaller than the number of edges for the power-law graphs. Compared with the naïve queue-based frontier, BDF is more space-efficient as it only requires a queue and a bitmap while other implementations require ping-pang queues (i.e., two queues) to store the frontiers for the current and the next iteration. Besides this, BDF is expected to reduce expensive atomic operations.

To illustrate how BDF works, we compare it with Gunrock, using the BFS algorithm as an example. BFS algorithm is to compute the distance of all vertices from the source vertex. As Figure 6 shows, vertex 0 and vertex 1 are in the current frontier at the beginning. Vertices 2, 3, and 4 are neighbors of Vertices 0 and 1, and can be visited by both vertices 0, 1. Thus, they could be added to the frontier with atomic operations multiple times if they are processed by Gunrock. After this, Gunrock filters out redundant items with scan-based operations and generate the new frontier. As for Grus, it updates the new labels of visited vertices, and it writes the indicative byte of updated vertex to 1. When the updating finishes, Grus gets the new frontier by scanning over the bitmap. The effect of repeatedly setting a byte of the bitmap to 1 is identical to a one-time setting, and therefore BDF will not get redundant items in the frontier.

Given the above design, the overhead of generating the frontier in Grus can be expressed as

$$C_{Grus} \approx |v_{updated}| * c_{atomic} + n_{update} * c_{write}, \tag{1}$$

where $|v_{updated}|$ is the number of updated vertices, c_{atomic} is cost of one atomic operation, n_{update} is number of updates made to vertex property data, c_{write} is the cost of one write operation. As for Gunrock, the overhead is

$$C_{Gunrock} \approx n_{update} * c_{atomic} + c_{filter}, \tag{2}$$

where c_{filter} is the cost of filtering redundant items with parallel scan. $|v_{updated}|$ is less or equal to n_{update} as multiple updates may perform on the same vertices. For a skewed real-world graph, $|v_{updated}|$ is generally much less than n_{update}. Besides, c_{write} is much less than c_{atomic} on a GPU (4 cycles versus 36–76 cycles measured on Volta GPUs [25]). In this way, *bitmap-directed frontier* is expected to be more efficient than the traditional "compute and filter" execution style. We provide detailed evaluation in Section 6.3.

4.2.2 Warp-centric Load Balancing. Grus leverages a push-style warp-centric (WC) execution method. Push-style execution is well-used in both CPU- and GPU-based graph processing systems [4, 39, 54], and it can represent most of the graph processing workloads. Warp-centric execution means that Grus assigns one thread warp to process all edges of each vertex. A thread warp continually tries to load and process 32 (the warp size of NVIDIA GPU) remaining edges of one vertex until all edges of that vertex are processed. The benefits of warp-centric execution are two-fold. 1). Memory accesses more likely coalesce on account of continually processing 32 edges of each vertex. NVIDIA GPU has 128 Bytes L1 cache line. Ideally, one memory transaction can satisfy the data requirement of 32 edges (128B in total for 4B data format). Therefore, less memory read transactions will be requested. It could alleviate data thrashing between the main memory and GPU memory when the GPU memory is over-subscribed. 2). The workload is balanced well intra-warp. Threads in the same warp have basically the same amount of edges to process. It results in less thread divergence and higher warp execution efficiency.

Warp-centric execution is first proposed as part of a technique named Virtual Warp-centric (VWC) [23]. Compared to the original VWC, Grus does not virtualize warp size to remain lightweight as the optimal virtual warp size is not known ahead. Using warp-centric execution does have its limitations in theory. (1) Threads would be idle when processing low-degree (less than warp size) vertices. (2) Intra-block load-imbalance exists as one warp in a block may process extremely high degree vertices while other warps in the same block finish their work early. Thread-Warp-CTA (TWC) strategy [43] also includes Warp-centric execution. It utilizes thread, warp or CTA (cooperative thread array, same as thread block) to process vertices with varied degrees, reducing idle threads within warps and the intra-CTA imbalance. However, TWC has its own disadvantages. It needs block-scan operations, block synchronizations and global barrier to gather edges to process cooperatively across the CTA, which involves additional overhead.

Grus currently chooses warp-centric execution as the default load-balancing policy, because warp-centric execution is not only lightweight in terms of memory consumption and execution logic but also shows good performance in practice (as shown in Section 6). Grus will be further extended to support other load-balancing policies (TWC, etc.).

4.3 Grus' Body: The Programming Interface

Grus mainly targets iterative neighbor-based graph algorithms. During processing, vertices continually update their labels based on their neighbors' labels until a certain convergence point is reached. Several steps of computation are executed in one iteration.

As shown in Section 4.2.1, computing and then filtering the frontier is inefficient in terms of massive usage of atomic operations. This procedure is semantically rooted in the widely-used *Advance-Update-Filter* programming abstraction. To overcome this shortcoming while preserving the expressiveness, Grus leverages a straightforward, yet efficient programming interface at the higher level. For a given graph application, programmers only need to specify user-defined functions (UDF) with the following interfaces.

- *Prepare* is to process labels of all vertices at the beginning of each iteration. It is necessary for some algorithms like Data-Driven PageRank [64], for instance.
- *Update* is to process edges of vertices in the frontier. It updates the property of source nodes, destination nodes, or edges according to UDF.
- *Generate* is to add vertices to the new frontier. It determines which vertices to process based on algorithm-specific criteria. For most of the algorithms, it enqueues the vertices whose labels are updated currently.

Grus fully supports UDF on either source nodes, destination nodes or edges to update their labels. Grus can express connectivity-based and data-driven graph analysis algorithms, such as BFS, SSSP, CC, PageRank, and data-driven PageRank [64].

Figure 7 shows the implementation of Single-Source Shortest Path (SSSP) and Data-driven Pagerank (PR) algorithm in Grus' interface. As shown in Figure 7(a), Grus' algorithm skeleton defines the common routines of graph algorithms. It is implemented in a header file so that user can include it in their algorithms. `Kernel` defines how to map workloads to GPU threads. It assigns each active vertex in the current frontier to one warp as Grus leverages Warp-centric Load Balancing. The threads in the same warp process edges in parallel. `main` defines the main function. It initiates and sets the adaptive UM policy for data structures. In each iteration, Grus invokes `Prepare`, `Kernel` and the BDF updating function in sequence. The BDF updates its frontier based on the active information in bitmap, and resets its bitmap at the end of iterations. Figures 7(b) and 7(c) show how to define SSSP and PR in Grus' interface, respectively. The `data` structure is the algorithm-specific vertex labels (and necessary intermediate data). `Update` and `Generate` are implemented as CUDA inline device functions that are used in *kernel*. `Init` is to initialize the objective vertex label (distance for SSSP, for instance) and frontier when the processing starts.

5 EVALUATION METHODOLOGY

This section presents our evaluation results. We are particularly interested in four questions:

- How is the performance of Grus compared to other in-memory processing frameworks?
- How is the performance of Grus compared to out-of-GPU-memory processing frameworks?
- How do the above proposed techniques contribute to Grus' overall performance?
- How is the performance of Grus with limited hardware resources?

Implementation. We implement Grus in around 2,000 lines of CUDA and C++ code. The latest version of Grus supports batched multiple-algorithm processing tasks. Users can write single-algorithm applications with just a few lines of code with our *Prepare-Update-Generate* interface.

Evaluation Platform. We conduct experiments on a Linux server with two 2.40 GHz Intel 20-core, hyper-threaded Xeon 6148 CPUs (80 threads in total). The main memory is 256 GB. One

```
void Kernel(){
  for vtx in frontier{  // Assign each active vertex to one warp
    for edge in vtx.edgeSet{ //Threads in the same warp process edges in parallel
      Update();  // Update labels
      Generate(); // Enqueue vertices based on criteria
}}}
bool CheckConverge(){
    if (frontier.size()==0) return true;
    return false;}
void Main(){
  Init();
  SetMemPolicy();    //Set adaptive UM policy for UM prefetching and hints
  while(!CheckConverge()){
    Prepare();
    Kernel();
    frontier.getNew();    //BDF gets the new frontier and reset the bitmap.
  }}
```

(a) Graph algorithm skeleton

```
struct data{   int label[]; }
void Prepare(){} //not used in SSSP
void Update(){
  if (label[edge.dstV] > label[edge.srcV]+ edge.weight){
    atomicMin(&label[edge.dstV],label[edge.srcV]+edge.weight);
    updated = true;}
  updated = false;}
void Generate(){
  if(updated) frontier.add(edge.dstV);}
void Init(){
  std::fill(label.start(), label.end(), INF);
  label[src]=0;
  frontier.add(src);}
```

(b) The pseudocode for SSSP

```
struct data{   float rank[], delta[], delta2[]; }
void Prepare(){
    for vtx in V{
        res = delta[vtx];
        delta[vtx] = 0.0;
        delta2[vtx] = res;
        rank[vtx] += res;
    }}
void Update(){
    res = delta2[edge.srcV];
    update = res *ALPHA / edge.srcV.outDegree;
    delta_old = atomicAdd(&delta[edge.dstV], update);
    updated = true;}
void Generate(){
  if ((delta_old + update > EPSILON) && (delta_old < EPSILON))
      frontier.add(edge.dstV);}
void Init(){
    for vtx in V{
        rank[vtx] = 1.0 - ALPHA;
        update = ((1.0 - ALPHA) * ALPHA) / vtx.outDegree;
        for dstV in vtx.neighbor{
            atomicAdd(&delta[dstV], update);
        }}
    frontier.add_all();}
```

(c) The pseudocode for Pagerank

Fig. 7. Implementation of SSSP and Pagerank in Grus' interface.

NVIDIA RTX 2080Ti GPU with 11 GB GDDR6 memory is connected to this system through PCI-e ×16 interface. The NVIDIA RTX 2080Ti GPU has 68 multiprocessors (SMs), each with 64 CUDA cores. The operating system is Ubuntu 16.04 with Linux kernel 4.15.0. We use the NVCC compiler version 10.2.89 (g++ version 5.4.0) to compile.

Baseline Frameworks. Table 2 summarizes the evaluated frameworks. For in-memory baselines, we evaluate Cusha, Gunrock, SEP-Graph and SIMD-X. Cusha [28] is a framework optimized for memory coalescing. Gunrock [63] is a frontier-based framework using high-level primitives. Tigr [54] is a vertex-centric framework utilizing a virtual transformation technique to optimize degree-irregular graphs. SEP-Graph [60] is a hybrid framework automatically switching between synchronous or asynchronous execution mode, Push or Pull communication mechanism, and data-driven or topology-driven traversing scheme for optimized performance. SIMD-X [36] is a framework that leverages atomic-free task management. We download the source code of these frameworks from Github and compile them following their guidelines.

For out-of-GPU-memory baselines, Graphie [21], Garaph [37], and Subway [55] are the most related works. Unfortunately, Graphie and Garaph are not publicly accessible, thus we cannot directly evaluate them. We reference the published results in their papers instead. We limit the GPU hardware resource to run Grus for a relatively fair comparison. More details are shown in Section 6.2. Subway is a most recent work on out-of-GPU-memory graph processing. It generates

Table 2. Baseline Comparison

Name	Venue	Optimization highlight
Cusha	HPDC'14	Coalescing memory accesses
Gunrock	PPoPP'16	High-level abstraction
Graphie	PACT'17	Asynchronous streaming processing
Garaph	ATC'17	Collaborative CPU-GPU execution
Tigr	ASPLOS'18	Degree-optimized load-balancing
SEP-Graph	PPoPP'19	Adaptive optimization switching
SIMD-X	ATC'19	Atomic-free, kernel fusion
Subway	EuroSys'20	Asynchronous subgraph generation

Table 3. Graph Datasets in Evaluation from References [6, 66]

Dataset	$\|V\|$	$\|E\|$	Size (GB)	Domain
Livejournal	5M	69M	1.4	Social
Orkut	3M	117M	2.0	Social
UK-2005	39M	936M	18	Web
Twitter	41M	1.4B	27	Social
Friendster	65M	1.8B	35	Social
SK-2005	50M	1.9B	38	Web
UK-union	133M	5.5B	110	Web

The sizes are graphs in weighted edgelist format.

the subgraph containing active vertices at runtime. We get its source code from Github and select several large datasets for comparison.

Workload. We select four widely used graph algorithms and evaluate their performance: (1) BFS, (2) SSSP, (3) CC, and (4) PR [64].

For BFS and SSSP, we start with the first source node of each dataset for fair comparison. For PageRank, we use the same terminal condition with 0.85 as damping factor and 0.01 as error tolerance. We run on each framework until convergent or iteration number reaches 100. For all the frameworks, graph datasets are transformed into their required data format in advance. For Cusha, we experiment with its two new processing methods (*G-Shards* and *Concatenated Windows*) and report the best results for comparison.

When comparing with in-memory frameworks, we measure the elapsed times of execution on GPU until convergence (excluding data-transfer, preprocessing) to better show the execution performance on GPU. Note that Grus prefetches all data in UM to GPU if the graph can fit into GPU memory due to the adaptive UM policy, and prefetching UM to GPU yields the same bandwidth as explicit memory copy for large memory regions on the tested platform, therefore Grus will not suffer the overhead of UM for in-memory evaluations compared with using normal pinned *cudaMemcpy* or *cudaMemcpyAsync*. When comparing with out-of-GPU-memory frameworks, we measure the elapsed times including data-transfer. We repeat the experiment five times and report the average value of the obtained results.

Graph Dataset. We conduct experiments on a variety of widely used graph datasets listed in Table 3. LiveJournal (LJ) [66], Orkut (OK) [66], Twitter (TW) [6], and Friendster (FD) [66] are social networks. UK-2005 [6], SK-2005 (SK) [6], and UK-union [6] are web graph snapshots. We generate the edge weight ranging from 1 to 64 uniformly. Their sizes varies from 1.4 to 110 GB.

6 EVALUATION RESULTS

This section presents the evaluation results and analysis.

6.1 Comparison with In-memory Frameworks

In Table 4, we present detailed performance results for Grus and other representative GPU-based frameworks. We analyze the results from the following two perspectives:

Large Graph Processing Capability. As the graph data size grows, Cusha, Gunrock, Tigr, SIMD-X, and SEP-Graph face increasing difficulty in graph processing due to GPU memory allocation failure or other runtime errors. As shown in Table 4, Cusha cannot process a graph larger than the Orkut dataset, since its G-Shard and Concatenated Window data structure takes over two times space compared to the CSR format. In addition, Gunrock cannot process BFS, SSSP, and CC on graphs larger than Orkut; it cannot process PageRank on graphs larger than UK-2005. Further, Tigr cannot process graphs larger than Friendster for BFS, CC, and PR, or UK-2005 for SSSP due to

Table 4. Detailed Performance

		Elapsed time (ms)							
Alg.	Frameworks	LJ	Orkut	UK-2005	Twitter	SK-2005	Friendster	UK-union	Avg. Speedup
	Cusha	25	49	-	-	-	-	-	**4.4**
	Gunrock	24	50	-	-	-	-	-	**4.4**
BFS	Tigr	**9**	**7**	448	**247**	274	1,248	-	**2.2**
	SEP-Graph	10	**7**	**96**	-	-	-	-	**1.0**
	SIMD-X	14	15	-	-	-	-	-	**1.8**
	Grus	**9**	**7**	103	333	**77**	**667**	**1,387**	×
	Cusha	50	51	-	-	-	-	-	**1.8**
	Gunrock	96	164	-	-	-	-	-	**4.4**
SSSP	Tigr	37	**25**	738	-	-	-	-	**1.3**
	SEP-Graph	63	50	-	-	-	-	-	**2.0**
	SIMD-X	48	95	-	-	-	-	-	**2.4**
	Grus	**32**	**25**	**382**	**1,160**	**3,822**	**1,816**	**15,147**	×
	Gunrock	21	59	-	-	-	-	-	**1.5**
CC	Tigr	**18**	**22**	565	561	369	1,248		**1.0**
	Grus	21	26	**445**	**497**	**325**	**1,205**	**7,925**	×
	Cusha	625	449	-	-	-	-	-	**4.3**
	Gunrock	425	1,400	1,111	-	-	-	-	**3.7**
PR	Tigr	769	475	5,108	40,912	20,142	54,708	-	**6.4**
	SEP-Graph	243	124	-	-	-	-	-	**1.4**
	SIMD-X	631	552	-	-	-	-	-	**4.8**
	Grus	**219**	**69**	**895**	**12,016**	**1,061**	**6,136**	**8,757**	×

The results are in milliseconds (the less the better). The best results of each algorithm on one graph are in bold. "–" indicates out of memory error on GPU or other runtime errors. The average speedup is the geometric mean of Grus' speedup compared with that framework.

the memory issue of the edge weight data. Since SEP-Graph uses both CSR and CSC format to store the graph for switching execution directions at runtime, it doubles the GPU memory requirement in exchange for performance. SIMD-X reserves large space for frontier and loads graph weight into GPU memory for all algorithms in their implementation, thus cannot process graph larger than Orkut. In contrast, Grus can properly process all tested graphs whose edgelist sizes reach 110 GB.

Overall Performance. In general, Grus achieves the best performance on the majority of evaluated datasets and algorithms. For BFS, Grus achieves the best performance on most of the graphs except for UK-2005 and Twitter. On LJ and Orkut, Tigr, SEP-Graph, and Grus achieve similar performance. On UK-2005, the elapsed time of SEP-Graph is slightly smaller. This is because SEP-Graph is heavily optimized for Push-Pull dual-direction BFS execution on the cost of twice of memory consumption. For SSSP, Grus achieves the best performance on all graphs except for Orkut. It shows the same elapsed times on Orkut with Tigr. The reason behind this is that Tigr leverages a technique named Edge-array Coalescing. This technique interleaves edge destination node and edge weight data in the memory to coalesce memory access but introduces significant pre-processing overhead. Therefore, it is undesirable considering the overall time consumption. For CC, Tigr achieves slightly better performance for LJ and Orkut. As for PageRank, Grus outperforms all the baselines on all graphs. Even though SIMD-X utilizes block-level synchronization and inter-thread communication to avoid atomic operations, Grus still outperforms SIMD-X in all experiments.

Table 5. GPU Specification Used for Graphie, Garaph, and Grus

Name	Graphie	Garaph	Grus
GPU	NVIDIA Titan Z	NVIDIA GTX1070	NVIDIA RTX2080Ti
GPU Memory	6 GB GDDR5	6 GB GDDR5	11 GB GDDR6
CUDA cores	2,880	1,920	4,352
Bus Interface	PCIe 3.0 ×16	PCIe 3.0 ×16	PCIe 3.0 ×16

In summary, Grus achieves 1.8× to 4.4× average speedup over Cusha, 1.5× to 4.4× average speedup over Gunrock, up to 6.4× average speedup over Tigr, 2.0× average speedup over SEP-Graph and 1.8× to 4.8× average speedup over SIMD-X on evaluated algorithms. Note that experiments in this section are comparing Grus with in-memory frameworks in terms of on-GPU execution time, excluding data-transfer time between the main memory and GPU memory. The graph data is already in the GPU memory when GPU kernels start to execute. Considering that execution on UM resident on GPU and normal allocated GPU memory is identical, we can conclude that the speedups of Grus are mainly contributed by the Right Wing of Grus (i.e., execution optimizations) for in-memory experiments.

6.2 Comparison with Out-of-GPU-memory Frameworks

Comparison with Graphie and Garaph. As stated before, Graphie and Garaph are not publicly accessible, and we do not have the same GPU used in their paper. We choose to limit the hardware resources of our GPU for comparison as fair as possible. The specifications of GPUs used in the Graphie and Garaph paper are listed in Table 5. The GPU we use have more memory and cores than GPUs used for Graphie and Garaph. Specifically, as the GPUs used in Graphie and Garaph both have 6 GB memory, we limit Grus to use 6 GB GPU memory by allocating 5 GB dummy data on our GPU in advance. As their used GPUs has less CUDA cores, we limit Grus to use 30 SMs (1,920 cores) out of 68 SMs by involving dummy kernel to fully occupy the rest 38 SMs, trying to simulate a GTX1070 GPU. We also add a configuration using only 8 SMs to simulate a low-end GPU.

In Table 6, we list the reported results of the commonly evaluated graphs and algorithms in their paper. Note that Graphie reports the time of CC until convergence while Garaph reports 10-iteration time for twitter and 5-iteration time for UK-union. For Twitter, all results take into account both CPU-GPU memory transfer and execution on GPU. For UK-union, the runtime starts from reading graphs from the storage. We follow their evaluation methodologies in their papers. Note that Garaph also fully leverages a 20-thread CPU for collaborative execution in their paper while Grus processes graph only on GPU.

As Table 6 shows, Grus shows significant speedup over Graphie and Garaph even with only 8 SMs. When using 30 SMs, Grus achieves up to 5.0× speedup over Graphie and 4.4× speedup over Garaph for Twitter. As for UK-union, Grus achieves 19× speedup over Garaph. With only 8 SMs, Grus achieves 3.3× speedup over Graphie, 2.8× speedup over Garaph for Twitter and 19× speedup over Garaph for UK-union. Grus' significantly high speedup over Garaph for UK-union is because Garaph needs to process UK-union while reading graph shards from the SSD as the 64 GB main memory on their platform cannot hold all the necessary data. In contrast, Grus can load the whole graph in the main memory and process with peak memory usage less than 45 GB owing to space-efficiency. Thus, Grus could yield even higher speedup if we compare the CC runtime till convergence for UK-union. Since Grus can achieve higher performance with even less GPU resources, we conclude that Grus is more efficient than Graphie and Garaph.

Table 6. Runtime on the Commonly Evaluated Datasets (Including Data Transfer Time from the Main Memory to the GPU)

Dataset	Alg.	Time (s)				Speedup	
		Graphie	Garaph	Grus (30 SMs)	Grus (8 SMs)	30 SMs	8 SMs
Twitter	BFS	5.42	-	1.16	1.77	(4.7, –)	(3.1, –)
	SSSP	14.67	12.75	2.93	4.50	(5.0, 4.4)	(3.3, 2.8)
	CC	4.21	-	1.51	2.96	(2.8, –)	(1.4, –)
	CC(10itr.)	-	3.32	1.50	2.97	(–, 2.2)	(–, 1.1)
UK-union	CC(5itr.)	-	157	8.3	10.7	(–, 19)	(–, 15)

Grus is limited to use 6 GB GPU memory and 30 SMs (1920 cores) to simulate their configurations, and 8 SMs to simulate a lower-end GPU. The speedup is the achieved speedup of Grus over Graphie and Garaph respectively. Sign "-" indicates not evaluated.

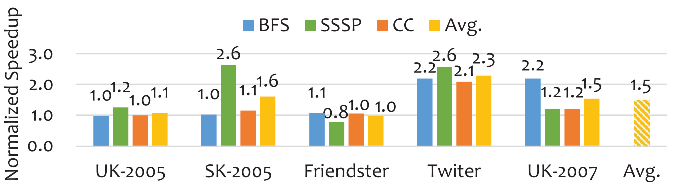
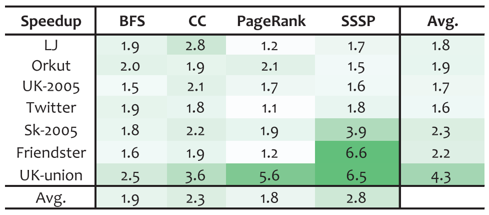


Fig. 8. Normalized speedup of Grus over Subway.



Speedup	BFS	CC	PageRank	SSSP	Avg.
LJ	1.9	2.8	1.2	1.7	1.8
Orkut	2.0	1.9	2.1	1.5	1.9
UK-2005	1.5	2.1	1.7	1.6	1.7
Twitter	1.9	1.8	1.1	1.8	1.6
Sk-2005	1.8	2.2	1.9	3.9	2.3
Friendster	1.6	1.9	1.2	6.6	2.2
UK-union	2.5	3.6	5.6	6.5	4.3
Avg.	1.9	2.3	1.8	2.8	

Fig. 9. Normalized speedup of Grus over Grus without the proposed adaptive UM management policy.

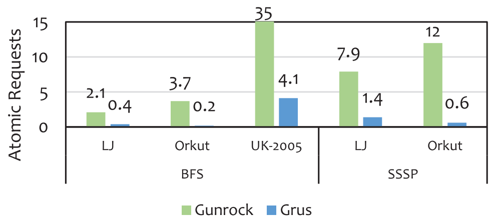
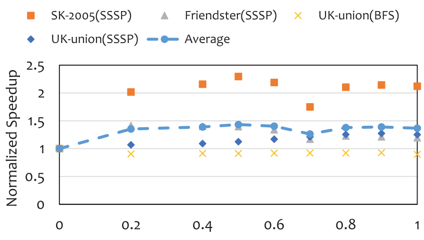


Fig. 10. The requested global atomic operations (in millions) of Gunrock and Grus.

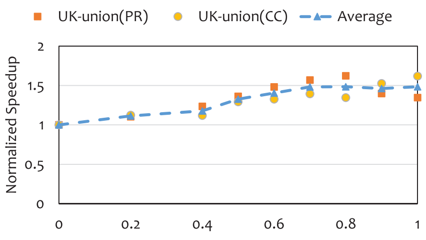
Comparison with Subway. We also compare Grus with Subway, the most recent design that is closely related to our work. As Subway is not optimized for PageRank, we mainly evaluate the performance of BFS, CC and SSSP on Subway. We find that Subway fails to process UK-union correctly due to overflow. Alternatively, we use UK-2007, a graph with 110 million vertices and 3.9 billion edges instead. We list the results on Twitter, SK-2005, Friendster, and UK-2007. As Figure 8 shows, Grus achieves up to 2.6× speedup on Twitter and SSSP on SK-2005. Grus has similar if not better performance on other datasets expect for CC on Friendster. Grus yields an average speedup of 1.5× on the evaluated datasets.

6.3 Performance Breakdown

Unified Memory Performance. To understand the impact of Grus UM optimization, we evaluate Grus with and without its adaptive memory management. Figure 9 shows the normalized speedup of Grus' adaptive memory management. Note that these results are based on the total time including UM prefetching overhead. In other words, it shows the overall speedup during the whole procedure. In summary, the adaptive memory management provides 1.1× to 6.6× overall speedup.



(a) Speedup of traversal algorithms.



(b) Speedup of full-active algorithms.

Fig. 11. Normalized speedup of prefetching ratio (τ) from 0 to 1 over no prefetching ($\tau = 0$). We group the test cases into two groups based on their algorithm type.

Among the four algorithms, SSSP gets the most performance gain from the adaptive memory management while PageRank gets the least. This is reasonable because SSSP is more memory-hungry while PageRank is more computing-intensive. Processing UK-union shows most performance gain, since UK-union is the largest graph in our experiment and the GPU memory is heavily over-subscribed.

To show the effectiveness of our algorithm-based prefetching ratio (τ) selection, we perform a sensitive analysis for it. We evaluate the performance of UM-over-subscribed test cases with different τs. As Figure 11 shows, prefetching provides speedup in most of test cases except for BFS on UK-union over no prefetching ($\tau = 0$). The average speedup of both kinds of algorithms grows from 1 to nearly 1.5 with fluctuations. For traversal algorithms, prefetching with $\tau = 0.5$ achieves the highest average speedup (1.43×). For full-active algorithms, prefetching with $\tau = 0.8$ achieves the highest average speedup (1.48×). This validates that our algorithm-based heuristic τ selection is efficient in these cases. We leave more sophisticated solutions to further research.

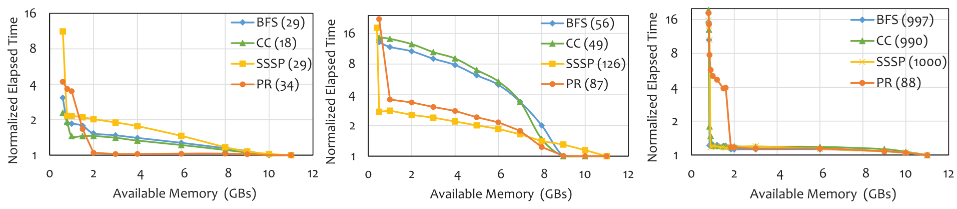
Atomic Operation Reduction. To evaluate the effectiveness of our proposed *Prepare-Update-Generate* interface, we measure the total requested global atomic operations during processing. We collect the `l1tex__t_set_accesses_pipe_lsu_mem_global_op_atom.sum` metric of NVIDIA Nsight Compute CLI [47] as the result. As Figure 10 shows, Grus takes significantly less atomic operations than Gunrock. Specifically, Grus saves 80% to 82%, 95%, and 88% atomic operations for LJ, Orkut, and UK-2005 compared to the Gunrock style execution, respectively. The results of BFS and SSSP on the same graph are similar, since the patterns of both algorithms are highly related to the graph structure. The average degree of vertices in Orkut is higher than the average degree of LiveJournal and UK-2005, and therfore processing Orkut benefits more from BDF.

Hardware Resource Utilization. To evaluate the execution efficiency of different frameworks, we characterize the hardware resource utilization using NVIDIA Nsight Compute. We collect the achieved percentage of utilization for SM, memory, L1/Tex cache and L2 cache with respect to the theoretical maximum for the most time-consuming iteration (expect for SIMD-X, since it uses one single kernel for all iterations) of BFS on Orkut.

As Table 7 shows, these frameworks have significant different results as their designs have different priorities. Gunrock has surprisingly low SM, L1/Tex and L2 utilization using its two load-balancing strategies. Tigr has the highest SM utilization, since it invokes threads on each vertice at all iterations no matter whether they are active or not. Cusha has the highest memory utilization due to its edge-centric graph format, which basically doubles the memory traffic compared with other vertex-centric frameworks. Sep-graph has the highest L1/Tex cache utilization, since it switches to pull-style execution, which has good locality in this iteration with the cost of duplicate graph structure data (using both CSR and CSC). Among these, Grus achieves the highest

Table 7. Achieved Percentage Utilization of GPU for BFS on Orkut

Frameworks	**Achieved percentage utilization**			
	SM	Memory	L1/Tex	L2
Gunrock-LB	3.0	26.3	8.5	7.0
Gunrock-TWC	2.0	23.9	7.2	6.0
SIMD-X	11.9	11.9	23.4	5.8
Tigr	**25.9**	31.7	21.9	23.6
Cusha	8.0	**54.2**	16.0	21.0
Sep-graph	13.0	28.1	**56.3**	24.6
Grus	8.1	28.7	28.4	**28.7**



(a) Friendster (b) SK-2005 (c) UK-union

Fig. 12. Normalized elapsed times for four algorithms of Grus with variant available memory capacities (the smaller the better), compared with using all 11 GB GPU. Numbers in bracket are the iteration number.

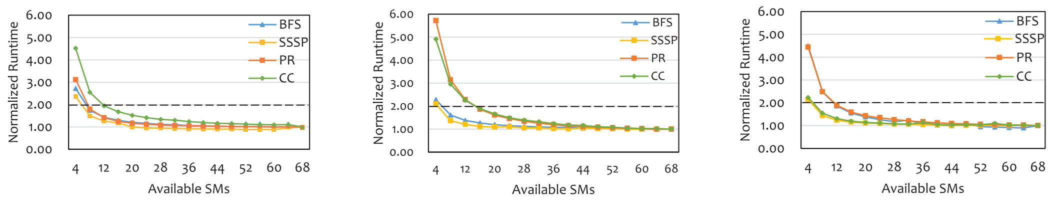
L2, second-highest L1/Tex and moderately high memory utilization. This shows that even though Grus prioritizes space efficiency in the design, its execution strategy (the Right Wing) still allows to outstandingly utilize hardware resources without sacrificing parallelism nor data locality.

6.4 Hardware Sensitive Analysis

With the evolving technology like GPU virtualization and GPU multi-tasking, sharing GPUs becomes more common in multi-tenant datacenters and HPC clusters. We are interested in how is the performance of Grus with limited hardware resources to simulate GPU-sharing environment. Due to space limitation, our extensive evaluation focuses on performance with varying memory capacities and computing resources. These results provide insights as Multi-Instance GPU (MIG) technology of the recent NVIDIA Ampere architecture GPU provides isolation of SMs, L2 cache banks and DRAM between GPU instances [48]. We leave the interference of co-allocated workloads in UM environment as further research topic.

Performance with Varied Memory Capacity. We investigate how does available GPU memory capacity affect the performance of processing large graphs with over-subscription. We limit the available memory capacity from less than 1 to 11 GB by launching a dummy kernel to occupy the rest of GPU memory in advance. BFS, PageRank and CC process graph topology data while SSSP needs additional graph edge weight data. The topology data size of Friendster, SK-2005 and UK-union is around 8, 9, and 15 GB, respectively.

As Figure 12 shows, the results have some patterns. For BFS, PageRank and CC on Friendster and SK-2005, the results are barely influenced by the change of available memory capacity as long as the GPU memory is not over-subscribed. When GPU memory is over-subscribed, elapsed time grows as the available memory size becomes smaller. This is straightforward to understand that less amount of data can resident on GPU with less available memory, and thus more page faults occur.



(a) Friendster (b) SK-2005 (c) UK-union

Fig. 13. Normalized elapsed times for four algorithms of Grus with variant computing units (SMs), compared with using all 68 SMs of a RTX 2080Ti GPU.

The changes of PageRank on Friendster and all algorithms on UK-union are very gentle when the available memory is larger than 2 GB. SK-2005 seems to be more sensitive to memory capacity changes. When available memory capacity is lower than a certain number (depending on graphs and algorithms), the runtime grows explosively. This is because memory capacity is so small that memory storing ever-changing objective vertex label data and frontier over-subscribes available GPU memory already. It is worth noting that the processing requires around 2 GB memory for Friendster, 8 GB memory for SK-2005, and less than 2 GB memory for UK-union with twice the runtime compared to using all the GPU memory.

Performance with Varied Computing Resources. We investigate how is the performance of Grus processing large graphs with limited GPU computing resources. We limit the number of the available SMs from 4 to 68 by launching a dummy kernel to occupy the rest of SMs. As Figure 13 shows, the runtimes are very insensitive to the number of SMs. It only requires 16 SMs to achieve half the performance using all 68 SMs (8 SMs is enough for most of the cases).

Insight. As the above results show, the runtimes of Grus processing graphs with over-subscribed UM is affected by the available memory capacity and computing resources. However, severely limiting memory capacity and computing resources results in a relatively little slowdown (depending on dataset and algorithm). The performance bottleneck of Grus is not CUDA cores or GPU memory capacity but CPU-GPU interface bandwidth or page handling overhead. The above results also indicate that there is an opportunity to balance the processing runtime with GPU memory budget and computing resources in the GPU-sharing environment. We leave this for further research.

7 RELATED WORK

GPU Graph Processing Frameworks. Merrill et al. used prefix sum to construct fine-grained tasks and leveraged vertex/edge frontier to store vertices (or edges) to be processed [43]. Hong et al. [23] proposed the virtual-warp-centric programming method to process graph by assigning jobs to threads in a warp for better load-balancing. Totem [18] system is a GPU-CPU hybrid platform for graph processing. It uses CPU to handle high-degree nodes for fast sequential processing and allocate numerous low-degree nodes on GPU for massive parallelism processing. MapGraph [16] combines three different scheduling strategies together and dynamically chooses the most suitable one for each vertex based on its degree. Cusha [28] uses two novel data structures, named G-Shards and CW, to avoid non-coalesced memory access. Gunrock [63] performs operations on the frontier with data-centric abstraction. Tigr [54] proposes a virtual transformation to transform skewed graphs into *virtual vertices* for load-balancing. EtaGraph [62] proposes a lightweight virtual graph transformation for load-balance. This work leverages UM to overlap the computing and data-transferring. AsynGraph [68] processes important vertices more times within each round so that the vertex states are propagated more efficiently among the graph.

Hetero-mark [59] and Chai [20] are benchmark suites for CPU-GPU applications including several graph algorithms. They present preliminary results on UM (or shared virtual memory in OpenCL [29]). Mailthody et al. [38] proposed a CPU-GPU collaborative algorithm for Triangle Counting and Truss Decomposition with UM. Pearson et al. [50] utilized UM to simultaneous reading graph data from disk and construct graph in GPU memory with a double-buffering technique.

There are works on processing graph with spare linear algebra representation [5, 8, 65, 67]. SMASH [26] is a hardware-software solution for sparse matrix operations. It leverages a hierarchy of bitmaps to encode spare matrices along with hardware support.

Large Graph Processing on GPU. To enable large graph processing capability on GPU, multi-GPU-based schemes [4, 22, 41, 69] and distributed processing methods [24, 31] are proposed. However, these methods have certain limitations. Some of them distribute graph duplicates on GPUs to accelerate processing [41], still being limited by the memory capacity of a single GPU. The rest of them leverage the aggregated memory of multiple GPUs to process large graphs by distributing partitions among GPUs. To process larger graphs, these methods require more GPUs for larger aggregate GPU memory to hold the graph data. To hide communication overhead and maintain scalability, distributed processing relies on high-speed inter-GPU (e.g., NVLINK) and inter-node interconnect for better performance. Thus, these methods are only feasible in datacenters or on supercomputers and have huge cost in terms of capital expenditure and power consumption, which is not accessible for many enterprises and researchers. Grus makes efforts to effectively leverage the GPU of off-the-shelf severs and PCs to process large-scale graph.

There are also efforts [21, 31, 37] on leveraging the out-of-core processing method to process large-scale graphs on GPU. Similar to CPU-based frameworks like GraphChi [32] and X-Stream [53], these frameworks generally partition graphs into chunks to fit into GPU memory for processing. To hide the data-transfer overhead between CPU and GPU, they use asynchronous GPU streams to overlap data movement and kernel execution. One major drawback of this approach is that the statically partitioned graph chunks are less flexible to process. Chunks need to be fully transferred to the GPU memory no matter how much data is actually used. Recently, Subway [55] chooses to generate subgraphs based on the vertex active information at runtime and processes them asynchronously. Scaph [70] utilizes two graph processing engines for high-value subgraphs and low-value subgraphs at each iteration, respectively.

Oversubscribing GPU Memory. *vDNN* [52] is a runtime memory manager to handle memory allocation, movement between CPU and GPU memory for DNN workload. Song et al. [57] studied on characterizing the performance of GPU acceleration system for CNN applications. They proposed a tuned GPU acceleration framework to handle the gap caused by the uneven computing loads at different CNN layers and fixed computing capacity provisioning. Wang et al. [61] proposed a Fine-tune Structured Sparsity Learning method to take advantage of the CSR format to encode the large sparse matrix for sparse neural networks. Buddy Compression [10] splits memory-entries to high-bandwidth GPU memory or a slower-but-larger buddy memory for HPC and DL workloads.

Optimizations for UM. MASK [3] presents several address-translation-aware cache and memory management mechanisms to reduce the overhead of address translation. Mosaic [2] presents a mechanism to support multiple page sizes for GPU virtual memory. Ganguly et al. [17] proposed two locality aware pre-eviction policies to help with hardware prefetcher for UM oversubscription. Li et al. [35] leveraged compiler techniques to achieve adaptive implicit and explicit data transfer and prevent data thrashing. ETC [34] is a memory management framework to mitigate the oversubscription overhead using proactive eviction. Kim et al. [30] proposed a runtime and hardware solution to improve the efficiency of UM for irregular applications. This work leverages a Virtual Thread technique to oversubscribe GPU threads and improve the batch size of UM page fault handling, and thus to amortize the fault handling overhead. Besides this, they proposed a unobtrusive eviction technique to reduce the page migration latency.

Optimizing Atomics on GPU Elteir et al. [14] characterized the performance of atomic operations on AMD GPUs and proposed a software-based atomic operation. Gómez-Luna et al. [19] studied atomic additions on shared memory through a microbenchmark-based analysis. Nasre et al. [44] proposed two high-level methods, namely, leveraging barrier-based processing and exploiting algebraic properties, to eliminate atomics in irregular programs. Franey et al. [15] presented a mechanism for implementing low-cost coherence and speculative acquisition of atomic data on the GPU. Adinets [1] introduced a technique to aggregate atomic operations inside each warp so that the total number of atomics is reduced. Egielski et al. [13] proposed two principles for efficient reduction of atomic collisions and a set of reduced-collision atomic algorithms.

8 CONCLUSION

Graph application has drawn great attention in recent years. Graph processing on GPU-accelerated machines demands better performance, efficiency, and scalability. In this work, we develop Grus, a graph processing system on Unified Virtual Memory enabled GPUs. Grus is systematically optimized in memory management and kernel execution for processing large graphs on GPUs. It leverages both trimmed memory access and lightweight frontier structure to reduce the overhead rooted in prior counterparts. Extensive evaluation on Grus demonstrate the effectiveness of our design in both in-memory and out-of-memory graph processing scenarios.

REFERENCES

[1] Andy Adinets. 2014. CUDA Pro Tip: Optimized Filtering with Warp-Aggregated Atomics | NVIDIA Developer Blog. Retrieved from https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/.

[2] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2018. Mosaic: Enabling application-transparent support for multiple page sizes in throughput processors. *Operat. Syst. Rev.* 52, 1 (2018), 27–44.

[3] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU memory hierarchy to support multi-application concurrency. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. 503–518.

[4] 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*. 235–248.

[5] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. 2017. SlimSell: A vectorizable graph representation for breadth-first search. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'17)*. 32–41.

[6] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph framework I: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference (WWW'04)*. ACM Press, New York, NY, 595–601.

[7] Tanya Brokhman, Pavel Lifshits, and Mark Silberstein. 2019. GAIA: An OS page cache for heterogeneous systems. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'19)*. 661–674.

[8] Shuai Che, Bradford M. Beckmann, and Steven K. Reinhardt. 2017. Programming GPGPU graph applications with linear algebra building blocks. *Int. J. Parallel Program.* 45, 3 (2017), 657–679.

[9] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. 44–54.

[10] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David W. Nellans, and Stephen W. Keckler. 2019. Buddy compression: Enabling larger memory for deep learning and HPC workloads on GPUs. Retrieved from https://arxiv:1903.02596.

[11] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU'10)*. Association for Computing Machinery, New York, NY, 63–74. DOI: https://doi.org/10.1145/1735688.1735702

[12] Advanced Micro Devices. 2017. Radeon's next-generation Vega architecture. Retrieved from https://www.techpowerup.com/gpu-specs/docs/amd-vega-architecture.pdf.

[13] Ian J. Egielski, Jesse Huang, and Eddy Z. Zhang. 2014. Massive atomics for massive parallelism on GPUs. In *Proceedings of the International Symposium on Memory Management (ISMM'14)*. 93–103.

[14] Marwa K. Elteir, Heshan Lin, and Wu-chun Feng. 2011. Performance characterization and optimization of atomic operations on AMD GPUs. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'11)*. 234–243.

[15] Sean Franey and Mikko H. Lipasti. 2013. Accelerating atomic operations on GPGPUs. In *Proceedings of the 7th IEEE/ACM International Symposium on Networks-on-Chip (NoCS'13)*. 1–8.

[16] Zhisong Fu, Harish Kumar Dasari, Bradley R. Bebee, Martin Berzins, and Bryan B. Thompson. 2014. MapGraph—Graphprocessing at 30 billion edges per second on NVIDIA GPUs. In *Proceedings of the International Semantic Web Conference*.

[17] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami G. Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. 224–235.

[18] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei Ripeanu. 2012. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. 345–354.

[19] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides Benítez, and Nicolás Guil Mata. 2013. Performance modeling of atomic additions on GPU scratchpad memory. *IEEE Trans. Parallel Distrib. Syst.* 24, 11 (2013), 2273–2282.

[20] Juan Gómez-Luna, Izzat El Hajj, Li-Wen Chang, Victor Garcia-Flores, Simon Garcia De Gonzalo, Thomas B. Jablin, Antonio J. Peña, and Wen-mei W. Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17)*. 43–54.

[21] Wei Han, Daniel Mawhirter, Bo Wu, and Matthew Buland. 2017. Graphie: Large-scale asynchronous graph traversals on just a GPU. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17)*. 233–245.

[22] Changwan Hong, Aravind Sukumaran-Rajam, Jinsung Kim, and P. Sadayappan. 2017. MultiGraph: Efficient graph processing on GPUs. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT'17)*. 27–40.

[23] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA graph algorithms at maximum warp. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'11)*. 267–276.

[24] Zhihao Jia, Yongkee Kwon, Galen M. Shipman, Patrick S. McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-GPU system for fast graph processing. *Proc. Very Large Data Base Endow.* 11, 3 (2017), 297–310.

[25] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. Retrieved from https://arXiv:1804.06826.

[26] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri-Ghiasi, Taha Shahroodi, Juan Gómez-Luna, and Onur Mutlu. 2019. SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. 600–614.

[27] Farzad Khorasani. 2014. Multi-threaded Large-Scale RMAT Graph Generator. Retrieved from https://github.com/farkhor/PaRMAT.

[28] 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 (HPDC'14)*. 239–252.

[29] Khronos OpenCL Working Group. 2020. The OpenCL C 3.0 Specification (Provisional). Retrieved from https://www.khronos.org/registry/OpenCL/specs/3.0-unified/html/OpenCL_C.html.

[30] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-aware unified memory management in GPUs for irregular workloads. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 1357–1370.

[31] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. GTS: A fast and scalable graph processing method based on streaming topology to GPUs. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. 447–461.

[32] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 31–46.

[33] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin C. Herbordt. 2014. An investigation of unified memory access performance in CUDA. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'14)*. 1–6.

[34] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. 49–63.

[35] Lingda Li and Barbara M. Chapman. 2019. Compiler assisted hybrid implicit and explicit GPU memory management under unified address space. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19)*. 51:1–51:16.

[36] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and processing of graph algorithms on GPUs. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'19)*. 411–428.

[37] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'17)*. 195–207.

[38] Vikram S. Mailthody, Ketan Date, Zaid Qureshi, Carl Pearson, Rakesh Nagi, Jinjun Xiong, and Wen-Mei Hwu. 2018. Collaborative (CPU + GPU) algorithms for triangle counting and truss decomposition. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'18)*. 1–7.

[39] 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 (SIGMOD'10)*. 135–146.

[40] Pak Markthub, Mehmet E. Belviranli, Seyong Lee, Jeffrey S. Vetter, and Satoshi Matsuoka. 2018. DRAGON: Breaking GPU memory capacity limits with direct NVM access. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. 32:1–32:13.

[41] 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 (SC'14)*. 572–583.

[42] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. 201–213.

[43] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *SIGPLAN Not.* 47, 8 (Feb. 2012), 117–128. DOI: https://doi.org/10.1145/2370036.2145832

[44] 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'13)*. 96–107.

[45] NVIDIA. 2017. NVIDIA Tesla V100 GPU architecture. *White Paper*v1.1 (2017), 53. Retrieved from http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[46] NVIDIA. 2018. NVIDIA Pascal Architecture. https://www.nvidia.com/en-us/data-center/pascal-gpu-architecture/.

[47] NVIDIA. 2020. Nsight Compute CLI :: Nsight Compute Documentation. Retrieved from https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html.

[48] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. Retrieved from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[49] NVIDIA. 2020. Programming Guide: CUDA Toolkit Documentation. Retrieved from https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[50] Carl Pearson, Mohammad Almasri, Omer Anjum, Vikram S. Mailthody, Zaid Qureshi, Rakesh Nagi, Jinjun Xiong, and Wen-Mei W. Hwu. 2019. Update on triangle counting on GPU. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'19)*. 1–7.

[51] Usman Pirzada. 2019. Intel Hints Towards An Xe "Coherent Multi-GPU" Future With CXL Interconnect. Retrieved from https://wccftech.com/intel-xe-coherent-multi-gpu-cxl/.

[52] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. 18:1–18:13.

[53] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP'13)*. 472–488.

[54] 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)*. 622–636.

[55] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing data transfer during out-of-GPU-memory graph processing. In *Proceedings of the 15th EuroSys Conference (EUROSYS'20)*. 12:1–12:16.

[56] Nikolay Sakharnykh. 2018. Everything you need to know about unified memory. In *Proceedings of the NVIDIA GPU Technology Conference (GTC'18)*.

[57] Mingcong Song, Yang Hu, Yunlong Xu, Chao Li, Huixiang Chen, Jingling Yuan, and Tao Li. 2016. Bridging the semantic gaps of GPU acceleration for scale-out CNN-based big data processing: Think big, see small. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT'16)*, Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu (Eds.). ACM, 315–326.

[58] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David R. Kaeli. 2019. MGPUSim: Enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. 197–209.

[59] Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David R. Kaeli. 2016. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*. 13–22.

[60] 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 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'19)*. 38–52.

[61] Jianzong Wang, Zhangcheng Huang, Lingwei Kong, Jing Xiao, Pengyu Wang, Lu Zhang, and Chao Li. 2019. Performance of training sparse deep neural networks on GPUs. In *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC'19)*. IEEE, 1–5.

[62] Pengyu Wang, Lu Zhang, Chao Li, and Minyi Guo. 2019. Excavating the potential of GPU for accelerating graph traversal. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'19)*. 221–230.

[63] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2015. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. 265–266.

[64] Joyce Jiyoung Whang, Andrew Lenharth, Inderjit S. Dhillon, and Keshav Pingali. 2015. Scalable data-driven PageRank: Algorithms, system issues, and lessons learned. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing: Parallel Proceesing (EUROPAR'15)*. 438–450.

[65] Carl Yang, Yangzihao Wang, and John D. Owens. 2015. Fast sparse matrix and sparse vector multiplication algorithm on the GPU. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPS'15)*. 841–847.

[66] Jaewon Yang and Jure Leskovec. 2012. Defining and evaluating network communities based on ground-truth. In *Proceedings of the 12th IEEE International Conference on Data Mining (ICDM'12)*. 745–754.

[67] Peter Zhang, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. 2016. GBTL-CUDA: Graph algorithms and primitives for GPUs. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPS'16)*. 912–920.

[68] Yu Zhang, Xiaofei Liao, Lin Gu, Hai Jin, Kan Hu, Haikun Liu, and Bingsheng He. 2020. AsynGraph: Maximizing data parallelism for efficient iterative graph processing on GPUs. *ACM Trans. Archit. Code Optim.* 17, 4 (2020), 29:1–29:21.

[69] 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)*. 601–614.

[70] Long Zheng, Xianliang Li, Yaohui Zheng, Yu Huang, Xiaofei Liao, Hai Jin, Jingling Xue, Zhiyuan Shao, and Qiang-Sheng Hua. 2020. Scaph: Scalable GPU-accelerated graph processing with value-driven differential scheduling. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'20)*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 573–588.

Received May 2020; revised November 2020; accepted December 2020