EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs

Seung Won Min
University of Illinois at Urbana-Champaign
Urbana, IL, USA
min16@illinois.edu

Vikram Sharma Mailthody
University of Illinois at Urbana-Champaign
Urbana, IL, USA
vsm2@illinois.edu

Zaid Qureshi
University of Illinois at Urbana-Champaign
Urbana, IL, USA
zaidq2@illinois.edu

Jinjun Xiong
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
jinjun@us.ibm.com

Eiman Ebrahimi
NVIDIA
Austin, TX, USA
eebrahimi@nvidia.com

Wen-mei Hwu
University of Illinois at Urbana-Champaign
Urbana, IL, USA
w-hwu@illinois.edu

ABSTRACT

Modern analytics and recommendation systems are increasingly based on graph data that capture the relations between entities being analyzed. Practical graphs come in huge sizes, offer massive parallelism, and are stored in sparse-matrix formats such as compressed sparse row (CSR). To exploit the massive parallelism, developers are increasingly interested in using GPUs for graph traversal. However, due to their sizes, graphs often do not fit into the GPU memory. Prior works have either used input data preprocessing/partitioning or unified virtual memory (UVM) to migrate chunks of data from the host memory to the GPU memory. However, the large, multi-dimensional, and sparse nature of graph data presents a major challenge to these schemes and results in significant amplification of data movement and reduced effective data throughput. In this work, we propose EMOGI, an alternative approach to traverse graphs that do not fit in GPU memory using direct cache-line-sized access to data stored in host memory.

This paper addresses the open question of whether a sufficiently large number of overlapping cache-line-sized accesses can be sustained to 1) tolerate the long latency to host memory, 2) fully utilize the available bandwidth, and 3) achieve favorable execution performance. We analyze the data access patterns of several graph traversal applications in GPU over PCIe using an FPGA to understand the cause of poor external bandwidth utilization. By carefully coalescing and aligning external memory requests, we show that we can minimize the number of PCIe transactions and nearly fully utilize the PCIe bandwidth with direct cache-line accesses to the host memory. EMOGI achieves 2.60× speedup on average compared to the optimized UVM implementations in various graph traversal applications. We also show that EMOGI scales better than a UVM-based solution when the system uses higher bandwidth interconnects such as PCIe 4.0.

PVLDB Reference Format:
Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal in GPUs. PVLDB, 14(2): 114-127, 2021.
doi:10.14778/3425879.3425883

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425883

PVLDB Availability Tag:
The source code of this research paper has been made publicly available at https://github.com/illinois-impact/EMOGI.

1 INTRODUCTION

Graph workloads are becoming increasingly widespread and common in various applications such as social network analysis, recommendation systems, financial modeling, bio-medical applications, graph database systems, web data, geographical maps, and many more [12–16, 18, 51, 53, 65]. Graphs used in these applications often come in huge sizes. A recent survey conducted by the University of Waterloo [53] finds that many organizations use graphs that consist of billions of edges and consume hundreds of gigabytes of storage.

The main challenge that graph application developers currently face is performing graph traversal computations on large graphs [53]. Because of the massive parallelism present in the graph traversal computation, GPUs are increasingly used to perform graph analytics. However, the ability to process large graphs in GPUs is currently hampered by their limited memory capacity. Thus in this work, we primarily focus on developing an efficient graph traversal system using GPUs that accesses large graph data from host memory.

For efficient storage and access, graphs are stored in a compressed sparse row (CSR) data format as it has low memory overhead. In CSR format, a graph is stored as the combination of a vertex list and an edge list. Even with CSR data format, large graph datasets cannot fit in today's GPU memory. Thus, most prior works store these large graphs in host memory and have GPUs access them through the unified virtual memory (UVM) mechanism [10, 11, 23, 29, 34, 37, 39, 41]. UVM brings both CPU memory and GPU memory into a single shared address space. UVM allows GPUs to simply access the data in the unified virtual address space and it transparently migrates required pages between host memory and GPU memory using a paging mechanism.

However, several prior work [10, 11, 23, 34, 37, 39, 41] have reported that the performance of graph traversal using UVM is not competitive. This is because memory accesses that go to the edge list during graph traversal are irregular in nature. Furthermore, based on our analysis of 1122 graphs that have at least 1M vertices and edges from LAW [13], SuiteSparse Matrix Collection [18], and Network Repository [51], we find the average degree per vertex is about 71. This implies that when those graphs are represented in a compressed adjacency list format such as CSR, each vertex's neighbor edge list is about 71 elements long on average. Thus transferring an entire 4KB page, as in the case of UVM, can cause memory thrashing and unnecessary I/O read amplification.

As a result, prior works have proposed pre-processing of input graphs by partitioning and loading those edges that are needed during the computation [27, 52, 55, 63] or proposing UVM specific hardware or software changes such as locality enhancing graph reordering [23], GPU memory throttling [37, 39], overlapping compute and I/O [25], or even proposing new prefetching policies in hardware that can increase data locality in GPU memory [10, 11, 34].

In this work, we take a step back and revisit the existing hardware memory management mechanism for when data does not fit in GPU memory. Specifically, we focus on zero-copy memory access which allows GPUs to directly access the host memory in cache-line granularity. With zero-copy, no complicated data migration is needed and GPUs can fetch data as small as 32-byte from the host memory. Even with such advantages, unfortunately, zero-copy is known to have underwhelming performance due to the low external interconnect bandwidth [22]. Interestingly, however, we do not find any systematic analysis showing the exact limiting factor of the zero-copy performance or leading to any effort to improve it.

Instead of making a premature conclusion, we build a system with a custom-designed FPGA-based PCIe traffic monitor and explore any opportunity to optimize zero-copy performance. We use the system to address the question of whether a sufficiently large number of overlapping cache-line-sized accesses can be sustained to 1) tolerate the long latency to host memory, 2) fully utilize the available bandwidth, and 3) achieve favorable execution performance for graph traversal applications. To this end, the key goal of our work is to avoid performing any pre-processing or data manipulation on the input graph and allowing GPU threads to directly perform cache-line-sized accesses to data stored in host memory during graph traversals.

By using a toy example, we show that by naively enabling zero-copy, the system cannot saturate the PCIe 3.0 x16 bandwidth (see § 3.3). To address this, we propose two key software optimizations needed to best exploit PCIe bandwidth for the zero-copy access. First, we propose the merged memory access optimization that optimizes for generating maximum-sized PCIe request to zero-copy memory (see § 3.3). Second, we propose forcing memory access alignment by shifting all warps to 128-byte boundaries when there is misalignment. This is because the memory access merge optimization does not guarantee memory request alignment. Such misalignment can result in performance degradation. While these optimizations sacrifice some parallelism and incur additional control divergence during kernel execution, their benefit in terms of improved bandwidth utilization far outweighs the cost. We then apply these two optimizations to popular graph traversal applications including breadth-first search (BFS), single-source shortest path (SSSP), connected components (CC), and PageRank (PR) to enable efficient traversal on large graphs.

Using real-world and synthetic large graphs (see Table 2), we show that EMOGI can achieve 2.93× speedup on average compared to the optimized UVM implementations of BFS, SSSP, CC, and PR benchmarks across a variety of graphs. We also evaluate EMOGI on the latest generation of the NVIDIA Ampere A100 GPU with PCIe 4.0 and show that EMOGI still remains performant and scales better than the UVM solution when using higher-bandwidth interconnect. EMOGI achieves speedups of up to 4.73× over current state-of-art GPU solutions [23, 52] for large out-of-memory graph traversals. In addition, EMOGI does not require pre-processing or runtime page migration engine.

To the best of our knowledge, EMOGI is the first work to systematically characterize GPU PCIe access patterns to optimize zero-copy access and to provide in-depth profiling results of varying PCIe access behaviors for a wide range of graph traversal applications. Overall, our main contributions can be summarized as follows:

(1) We propose EMOGI, a novel zero-copy based system for very large graph traversal on GPUs.
(2) We propose two zero-copy optimizations, memory access merge and memory access alignment, that can be applied to graph traversal kernel code to maximize PCIe bandwidth.
(3) We show EMOGI performance scales linearly with CPU-GPU interconnect bandwidth improvement by evaluating PCIe 3.0 and PCIe 4.0 interconnects.

The rest of the paper is organized as follows: we provide a brief primer on GPU based graph traversal and the challenges in executing graph traversals using UVM in § 2. We then discuss how to enable zero-copy memory with GPUs and discuss the reasons for its poor performance in a naive but common kernel code pattern in § 3. Using the gained insights, we then apply zero-copy optimizations to graph traversal algorithms in § 4. We discuss EMOGI's performance improvement for various graph traversal algorithms on several large graphs in § 5. § 7 discusses how EMOGI differs from prior work and we conclude in § 8.

2 BACKGROUND

In this section, we first provide a brief primer on GPU based graph traversal. Then we will describe techniques used to traverse graphs that cannot fit into the GPU memory.

2.1 Parallelizing Graph Traversal on GPUs

The exact workflow of the graph traversal depends on the type of the application and the optimization level, but a general flow can be described with Algorithm 1. First, before the traversal begins, initial active vertices need to be set. In case of BFS, only a single vertex needs to be set as active, which is basically a source vertex. Once all the initial active vertices are set, the graph traversal can begin. Graph traversal is composed of multiple iterations of sub-traversals. In each sub-traversal, all immediately neighboring vertices of the currently active vertices are exhaustively traversed. The condition to set the next active vertices depends on the type of application as well. In case of BFS, any neighboring vertices which are not visited

ever before are marked to be the next active vertices. The traversal ends once there are no more active vertices left in the graph.

The main benefit of the GPU implementation of the graph traversal comes from the massive number of vertices [15, 16, 65]. With a help of several atomic instructions, both the inner loop and the outer loop in Algorithm 1 can be fully parallelized with GPU for various kinds of graph traversal applications [17, 28, 30, 31].

As an input graph format for the GPU graph traversal, we use compressed sparse row (CSR) format. CSR is arguably the most popular way to represent a graph because of its low memory overhead [27, 48, 52, 55, 63, 69]. Certain graph processing frameworks such as nvGRAPH [1] support other input formats like coordinate list (COO), but they internally convert the inputs to the CSR format before the actual processing step. CSR encodes the entire graph with just 2 arrays, as shown in Figure 1. The edge list stores each vertex's neighbor list sequentially, such that all the neighbors of vertex 0 are stored first, then the neighbors of vertex 1, and so on. The vertex list is indexed by a vertex ID and stores the starting offset of that vertex's neighbor list in the edge list. The datatypes of both edge and vertex lists can vary depending on the graph size. For example, using a 4-byte datatype for the edge list can identify at most 4 billion nodes.

2.2 Out-of-Memory Graph Traversal on GPUs

Graphs, even in the CSR format, can be orders of magnitude larger than GPU memory. The easiest way to enable GPU based graph traversal on such graphs is to use the Unified Virtual Memory (UVM) [2, 3, 10, 11, 23, 29, 34, 37, 39, 41, 50]. UVM is a unified memory management module that provides a single memory address space accessible by both CPU and GPU through the page faulting mechanism. UVM reduces the burden on the programmer as they do not have to explicitly manage where the data resides. UVM transparently allows device memory over-subscription with the use of CPU memory, enabling computation on large data sets that exceed GPU device memory capacity. The UVM driver is responsible for on-demand page migration between the CPU and GPU.

The granularity of the data migration may vary depending on the data access pattern, but the minimum granularity is a system page size (4KB). Once the page is migrated, subsequent accesses to the same page do not need additional data migrations and the accesses can directly go to the GPU memory. If the memory footprint of the kernel is larger than the GPU memory, some pages need to be evicted from the GPU memory to host other pages during the kernel runtime. Since the entire management process is single-threaded, the overall performance of the UVM page migration heavily depends on the single-thread performance of the host CPU.

Algorithm 1: High-level Graph Traversal Flow

```
set_initial_active_vertex()
while there exist active vertices in G do
    for all vertices v1 in Graph G do
        if v1 is active then
            set v1 as inactive
            for all neighbors v2 of v1 do
                application_dependant_workload()
                if application_dependant_condition() then
                    set v2 as active
                end
            end
        end
    end
end
```

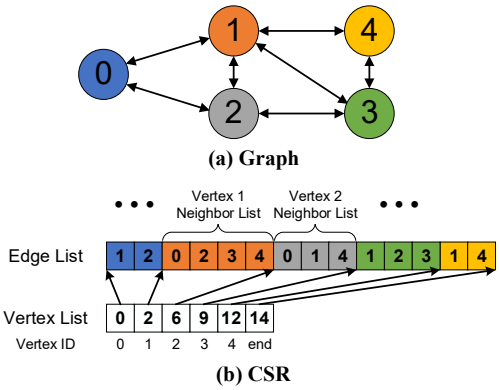

Figure 1: Sample undirected (a) graph and its (b) CSR representation. The edge list contains the neighbor list for each node. The vertex list is indexed by vertex IDs and contains the offsets for the starting position of that vertex's neighbor list in the edge list.

The inefficiency of UVM in graph traversal comes in two ways. First, for the very large graphs, it is hard to exploit temporal locality as the limited GPU memory capacity will cause frequent page thrashing. Second, there is a lack of spatial locality between neighbor lists, causing significant I/O read amplification and more frequent page migrations. For example, in Figure 1, the neighbor lists of the vertex 1 and 3 need to be accessed at the same time we start BFS from the vertex 4. However, as shown in the CSR representation, the lists are non-contiguous in the edge list. In a more realistic case with a large graph, these lists can be separated by millions of elements in the edge list. Therefore, accessing these two lists will likely generate two separate 4KB page migrations. Assuming that all accesses to the different neighbor lists will generate separate 4KB page migrations, all neighbor lists should have least 512 to 1024 of elements (depends on the datatype size) to make the 4KB data transfer 100% efficient, which might be quite challenging. By combining the frequent page migrations caused by the lack of data locality and the high page fault handling overhead of UVM, GPU performance can be severely throttled.

3 ZERO-COPY

To allow GPU threads access to the external memory in smaller granularity than UVM, GPUs support marking memory address ranges as zero-copy memory [4]. Zero-copy, also often referred to as *direct access*, does not require any page migration or duplication between the external and GPU memories. Instead, GPU threads access zero-copy memory as if it was GPU global memory, and the GPU transforms memory requests from the threads to memory requests over an external interconnect like PCIe. The target of the memory requests can be anywhere in the system as long as

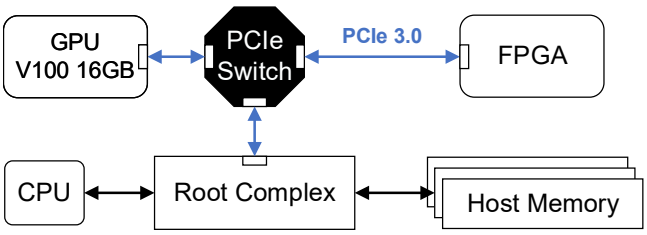

Figure 2: PCIe traffic monitoring environment. The FPGA is used to characterize the zero-copy memory access pattern from GPU.

the location can be memory-mapped into the shared bus address. Common examples include system memory, peer-connected PCIe network interface cards, and peer-connected GPUs. Due to the high latency of the external interconnects, using zero-copy was thought to have low bandwidth [22] and thus used for only accessing small and frequently shared data. In this section, we describe how to enable zero-copy and use a peer-connected FPGA to explore any optimization opportunities available for zero-copy in detail. Based on the analysis, we apply several optimizations and show correctly using zero-copy can nearly saturate the PCIe bandwidth.

3.1 Enabling Zero-Copy

From the system's point of view, zero-copy is enabled as follows: First, the data to be shared with GPU must be pinned in the host memory. Pinned memory cannot be swapped out to the disk or relocated by the host OS memory manager. Second, the corresponding bus address (e.g. PCIe) of the pinned data should be mapped into the GPU page table so the GPU can generate a correct external memory request. Finally, the mapped address should be passed to the user space so the programmer can use pointers in the GPU kernel to access the region.

From CUDA API's point of view, zero-copy can be enabled in three ways. First technique uses `cudaMallocManged()` to allocate UVM space and applies `cudaMemAdviseSetAccessedBy` flag with `cudaMemAdvise()`. The resulting data pointer can be directly used from CUDA kernels to generate zero-copy memory access. One thing worth noting here is that the `cudaMemAdviseSetAccessedBy` flag should not be used with other `cudaMemAdvise()` flags since the other flags override `cudaMemAdviseSetAccessedBy`. Second is by using `cudaMallocHost()`. This is the simplest method since the memory allocated by `cudaMallocHost()` can be directly used in the CUDA kernel to do zero-copy access. The last scheme uses general memory allocators, like `malloc()`, and `cudaHostRegister()` and `cudaGetDevicePointer()` on top of the allocated memory. In this case, the `cudaHostRegister()` pins the allocated memory space and `cudaGetDevicePointer()` returns a CUDA-compatible pointer. Our experiments showed all three techniques provided the same performance.

3.2 Zero-Copy Analysis Setup

In order to understand how GPU accesses external zero-copy memory over PCIe, we designed and built the monitoring system shown in Figure 2. The FPGA is connected to the GPU using a PCIe switch in peer-to-peer mode. Furthermore, the FPGA is programmed to advertise itself as a large memory using the base address register (BAR) region provided by the PCIe specification [8]. This advertised FPGA memory region can be mapped to the user space using the `mmap()` system call. The returned pointer value from the `mmap()` call can be used by CPU to access the FPGA as a zero-copy region. To allow the GPU direct access to the FPGA, we pass the pointer to `cudaHostRegister()` and `cudaGetDevicePointer()` CUDA APIs. The final pointer generated by the two APIs can be passed to the CUDA kernel code and dereferenced by GPU threads, thus allowing zero-copy access to the FPGA. Using this system, we can now analyze the low-level PCIe traffic of zero-copy memory access by the GPU. To this end, we add custom logic in the FPGA to monitor the request count, average/peak number of outstanding memory requests, and request sizes.

3.3 Zero-Copy Mechanism and Optimization

Now that we have a way to track zero-copy memory requests, we next need to understand the GPU access pattern to zero-copy memory. We create a toy example where the GPU needs to traverse a large 1D array in a zero-copy region and use a GPU kernel to copy its content to the GPU's global memory. The algorithm to solve the toy example can either perform strided access or merged with misaligned access or merged with aligned access. All PCIe traffic generated by these three variants is monitored using the FPGA monitoring platform and Intel VTune [5]. PCIe layer in Figure 3 shows the GPU access patterns we observed with the FPGA monitoring platform while trying different CUDA kernels. We observe that GPU can access the zero-copy memory in four different sizes starting from 32-byte to 128-byte in 32-byte steps. The access size is dependent on the algorithm access pattern and is described next.

Strided Access: In this method, each thread takes a chunk of the 1D-array and iterates over the chunk one element at a time. This access pattern is illustrated in Figure 3 (a). With GPU threads iterating over their own neighbor lists, we find that each thread generates a new 32-byte request every time they cross a 32-byte address boundary. Therefore, if the data type of the array is 4-byte, each PCIe request can serve up to 8 memory accesses.

However, this 32-byte request brings several limitations to the overall system. First, each PCIe 3.0 transaction layer packet (TLP) has at least an 18-byte of header overhead. Thus, fetching 32-byte of data makes the PCIe overhead ratio of at least 36%. Second, considering the PCIe latency, the number of outstanding requests to saturate the PCIe interconnect is non-negligible. With our test platform, we find the PCIe round trip time (RTT) between the GPU and the FPGA is about roughly 1.0us to 1.6us. By the PCIe 3.0 specification, the maximum number of outstanding requests is 256 as the width of the tag field used to record the outstanding request is 8-bit [8]. In this case, the maximum bandwidth we can achieve with only 32-byte requests and 1.0us of RTT is merely 32B / (1.0us / 256) = 7.63GB/s. If we assume the PCIe RTT is always 1.6us, the bandwidth decreases to 4.77GB/s. Third, the minimum memory access size for DDR4 DRAM is 64-byte in the test system. Considering that DDR4 2400MHz DRAM can provide 19.2GB/s of sequential bandwidth, requesting only 32-byte read requests halves the effective DRAM bandwidth to 9.6GB/s. Even the overall DRAM bandwidth can be increased by adding more memory channels,

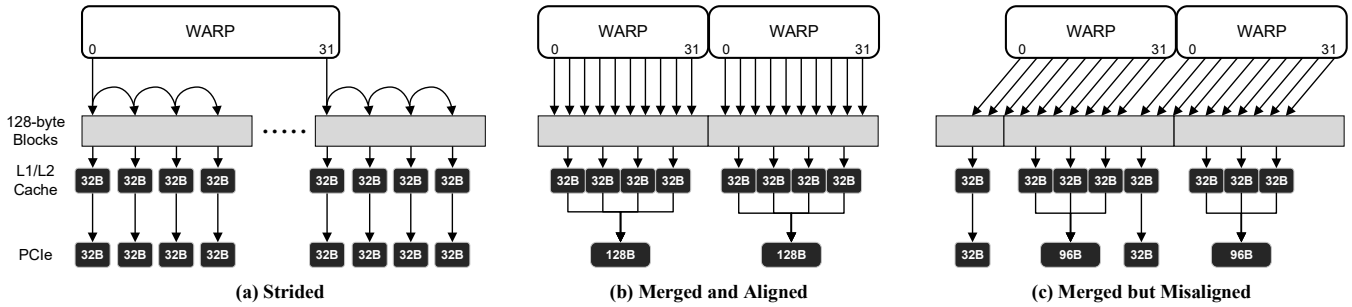

Figure 3: GPU PCIe memory request patterns observed with FPGA. In (a), each thread scans a different 128B block and end up making multiple 32B PCIe memory read requests. In (b), individual 32B memory read requests in a contiguous address space occur at the same time and GPU merges them into a single 128B PCIe memory read request. In (c) each warp is performing a misaligned memory request (off by 32B from 128B boundary) resulting in generating a 32B PCIe and 96B PCIe request. In this figure, we assume each memory access is 4B.

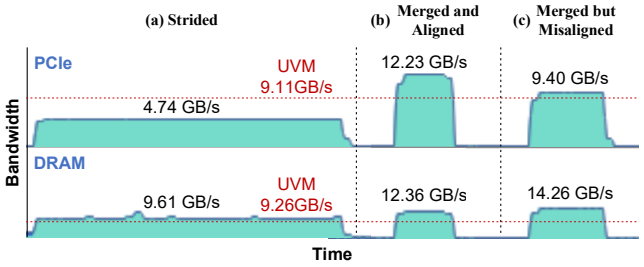

Figure 4: Average PCIe and DRAM bandwidth utilization for the different zero-copy access patterns, as reported by Intel VTune.

this is still very wasteful. Finally, these 32-byte data items will likely occupy GPU cache and can be evicted before all elements are traversed due to cache thrashing.

Figure 4 shows the average PCIe and DRAM bandwidth utilization over time when executing the traversing kernel as reported by Intel VTune. The peak bandwidth we achieved with UVM is drawn as a red dashed line in the figure as a reference. Looking at Figure 4 (a), we can clearly identify the limitations previously described. The amount of data that needs to be read from DRAM is doubled to serve 32-byte PCIe requests. The PCIe bandwidth is also far from the maximum PCIe 3.0 x16 bandwidth as the number of outstanding requests is not enough and the per-request PCIe overhead is significant. Furthermore, it results in transferring more bytes to the GPU compared to the original dataset size due to the frequent cacheline evictions. To address these limitations the key is to align and merge accesses. We analyze the PCIe and DRAM bandwidth utilization with these optimizations next.

Merged and Aligned Access: In this case, threads are grouped into warps, with each warp containing 32 threads, and the threads in a warp access consecutive elements in a 128-byte cacheline of the input array. This allows the GPU coalescing unit to automatically merge the contiguous 32-byte memory requests into a single larger 128-byte PCIe request (Figure 3 (b)). With 128-byte PCIe requests, it becomes much easier to reach the maximum PCIe bandwidth. First, the PCIe TLP overhead ratio decreases from 36% to 12.3% . Second, having only 135 PCIe outstanding requests is sufficient to reach 16GB/s of bandwidth (without considering other PCIe overheads). Lastly, 128-byte is a multiple DRAM request size and therefore there is no inefficiency in the DRAM interface. In Figure 4 (b), we see this approach can saturate the PCIe bandwidth at about 12.23GB/s , matching the measured bandwidth when using the `cudaMemcpy()` API to perform a block data transfer.

Merged but Misaligned Access: However, for all practical purposes, guaranteeing 128-byte alignment for any data structure can be difficult. It is possible that the starting index of a warp is not aligned with the 128-byte boundary. Some warps may need to make two separate PCIe requests to fetch a single 128-byte cacheline. In the worst case, if a warp's memory access is not 128-byte aligned and warps access contiguous regions of memory, the misalignment can be cascaded to all subsequent warps. Unfortunately, this results in all warps generating two PCIe requests. In Figure 3 (c), we show an emulated misaligned case where each warp is intentionally accessing memory offset by 32-byte from 128-byte boundary and therefore all warps end up generating a 32-byte and a 96-byte PCIe request. From Figure 4 (c), we can see the achieved PCIe bandwidth is lower than the aligned case. To avoid this, either the starting index of warps should be shifted or the input data must be shifted in memory so the data accessed first is 128-byte aligned.

4 EMOGI: ZERO-COPY GRAPH TRAVERSAL

Now that we understand zero-copy memory and its characteristics, we discuss how to efficiently use zero-copy memory for graph traversal when the graph cannot fit in the GPU memory. First, we describe the micro data locality we observed in graph traversal applications to justify why zero-copy should perform better than UVM (see § 4.1). Then, we introduce our baseline graph traversal algorithm (see § 4.2) and optimize it for zero-copy memory based on the knowledge we gathered from § 3.3 (see § 4.3).

4.1 Data locality in Graph Traversal

To exploit zero-copy for graph traversal, we preferably need at least 128-byte of spatial locality to best use each memory access. A single 128-byte zero-copy access can have 16 or 32 elements of data if the CSR data type is 8-byte or 4-byte, respectively. Compared to UVM, which requires at least 4KB of spatial locality (512 or 1024 elements

of data), finding 16 to 32 elements of spatial locality is reasonable for the graphs we studied.

Based on our analysis of 1122 graphs from Network Repository [51], SuiteSparse Matrix Collection [18], and LAW [13], we find the average degree per vertex is 71. This means, when those graphs are represented in an adjacency list format like CSR, each vertex's neighbor list is 71 elements long on average. Considering that graph traversal algorithms require scanning the entire neighbor list of a vertex, we can obtain a spatial locality of 71 elements on average in graphs. Such a spatial locality can benefit from efficient 128-byte requests to zero-copy memory. In contrast, it is more difficult to achieve the same level of efficiency using UVM since the available spatial locality is significantly less than the required 512 or 1024 elements.

4.2 EMOGI Baseline

EMOGI assumes the input graph is stored in the memory using the CSR data layout (see § 2.1). All input data structures are statically mapped during initialization. The edge list is allocated in the host memory as it doesn't fit in GPU memory, but other small data structures such as buffers and the vertex list are allocated in GPU memory. It is worth noting that even for the biggest graphs we evaluated (see § 5.2), the vertex list consumes at most 1GB of memory while the edge list can consume 38GB. Thus, GPU memory is sufficient for the vertex list.

EMOGI adopts vertex-centric graph traversal algorithms. For every vertex that needs to be processed, a worker is assigned and the worker traverses a neighbor list associated with the vertex in the edge list. Listing 1 shows the pseudo code of our naive baseline implementation. Here, the worker is a single GPU thread and each worker is assigned to the neighbor list associated with its corresponding vertex. When each neighbor list is larger than 128-byte, this baseline implementation has a similar memory access pattern to the strided case explained in § 3.3.

Compared with the UVM approach, EMOGI's graph traversal approach removes the page faults from occurring and reduces the I/O amplification as only the needed bytes are moved. In the vertex-centric graph traversal approach, the input graph is traversed by a single vertex depth on every kernel execution. Therefore the total number of kernels launched, say in the case of breadth-first-search (BFS), is equal to the distance between the source vertex to the furthest reachable vertex.

4.3 Optimizations

Since the EMOGI baseline implementation is similar to the strided case presented in § 3.3, it suffers from uncoalesced memory requests. As we noted, without addressing this, one cannot generate efficient PCIe requests to the zero-copy memory. In this subsection, we will discuss how EMOGI addresses this limitation using the insights from § 3.3 and modifying only the GPU kernel code of the traversal application. Thus, it is entirely possible to package the proposed optimizations into a library to lessen the programmer's effort when trying to exploit them.

4.3.1 Merged Memory Access: EMOGI performs merged memory accesses in per vertex granularity, similar to [31]. The difference between EMOGI and [31] is that EMOGI always fixes the worker size to an entire warp (i.e., 32 threads). Thus a whole warp is responsible for traversing the neighbor list of one vertex. The specific implementation of this optimization is explained with red comments in Listing 2. This allows EMOGI to always optimize for generating the maximum sized PCIe request to the zero-copy memory. If the input graph fits in the GPU memory and the average degree of vertices in the graph is small, fine-tuning the worker size could potentially reduce the number of idle threads during each fetch, exploit more memory parallelism, and ultimately utilize GPU global memory bandwidth more efficiently. However, EMOGI's primary goal is to achieve good performance on graphs that do not fit in the GPU memory and it requires fetching data over an external interconnect that is about 10-100× slower than the GPU global memory. In this case, fine-tuning and reducing the worker size cannot add any additional benefit as there is no further room to accept more memory requests in the already constrained interconnect. In fact, making smaller memory requests can have an adverse effect and decrease the effective bandwidth. Empirically we observed when the interconnect bandwidth is low, a large number of threads are idle. Therefore, assigning a 32-thread warp to fetch data for even vertices with very few neighbors results in acceptable performance.

Listing 1: Uncoalesced Memory Access

```
1  void strided(*edgeList, *offset, ...) {
2      thread_id = get_thread_id();
3      ...
4      start = offset[thread_id];
5      end = offset[thread_id + 1];
6
7      // Each thread loops over a chunk of edge list
8      for (i = start; i < end; i++) {
9          edgeDst = edgeList[i]; ...
10     } ...
11 }
```

4.3.2 Aligned Memory Access. As we discussed in § 3.3, a misaligned access to the 1D data array can result in multiple smaller zero-copy requests. To address this, we have to not only merge memory accesses but align them as well. However, doing this on a CSR edge list is not straightforward. This is because CSR doesn't align the edge list as alignment requires padding and thus increases memory footprint. Starting addresses of neighbor lists for graphs stored in CSR can be at any location in the memory.

One way to address this challenge is to pre-process the CSR graphs and align neighbor lists to 128-byte boundaries. However, this might incur excessive memory overhead. More importantly, one of the goals of this work is to avoid any pre-processing.

Therefore, instead of manipulating the input data, we force all warps to start from the closest preceding 128-byte boundary when there is misalignment. For instance, as shown in Listing 2 with blue comments, all starting indices fetched from the offset array is shifted to the closest 128-byte boundary before the list. With this change to the GPU kernel code, all subsequent warp memory accesses are guaranteed to have 128-byte alignment. Of course, some of the threads in the warp must be turned off during the first iteration of data fetching with a conditional statement to prevent

Listing 2: Coalesced Memory Access (Merged + Aligned)

```
#define WARP_SIZE 32

void aligned(*edgeList, *offset, ...) {
    thread_id = get_thread_id();
    lane_id = thread_id % WARP_SIZE;
    // Group by warp
    warp_id = thread_id / WARP_SIZE;
    ...
    start_org = offset[warp_id];
    // Align starting index to 128-byte boundary
    start = start_org & ~0xF; // 8-byte data type
    end = offset[warp_id + 1];

    // Every thread in a warp goes to the same edgelist
    for (i = start; i < end; i += WARP_SIZE) {
        // Prevent underflowed accesses
        if (i >= start_org) {
            edgeDst = edgeList[i + lane_id];
            ...
        }
    } ...
}
```

reading unnecessary bytes. Similar to the memory access merge optimization, this additional conditional statement increases the occurrence of control divergence in CUDA kernels. However, due to the high external interconnect latency, it is more important to not miss any opportunity for generating large memory requests.

5 EVALUATION

Our evaluation shows that (1) EMOGI improves the performance of graph traversal algorithms by efficiently accessing the zero-copy memory for very large graphs, (2) EMOGI is mainly limited by the PCIe bandwidth and it scales almost perfectly linearly when PCIe 3.0 is replaced with PCIe 4.0, (3) EMOGI remains performant even with the latest generation of GPU NVIDIA Ampere A100 [6] and achieves better scaling compared to the UVM optimized implementation.

5.1 Experiment Setup

5.1.1 System Overview: We use a Cascade-lake server machine with two 20 core Intel Xeon Gold 6230 CPUs equipped with 256GB of DDR4 2933MHz memory and an NVIDIA Tesla SXM2 V100 16GB GPU as our evaluation platform. The system is configured as shown in Figure 2. We use the FPGA only to analyze the zero-copy memory access pattern across different graphs. The detailed system specification is provided in Table 1. Graph edge lists are stored in the host memory while the vertex list and other temporary data structures are stored in the GPU memory.

5.1.2 Systems Compared: To show the performance benefit of EMOGI, we use three different graph traversal algorithms: Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank (PR) [46]. We base our initial implementation of BFS and SSSP from [32, 43], CC baseline implementation from [64], and PR baseline implementation from [44]. We compare EMOGI with the following systems:

Table 1: Evaluation system configuration.

Category	Specification
CPU	Dual Socket Intel Xeon Gold 6230 20C/40T
Memory	DDR4 2933MHz 256GB in Quad Channel Mode
GPU	Tesla V100 HBM2 16GB, 5120 CUDA cores
OS	CentOS 8.1.1911 & Linux kernel 5.5.13
S/W	NVIDIA Driver 440.82 & CUDA 10.2.89

(a) UVM implementation stores the CSR edge list in the UVM address space while the vertex list is kept in the GPU memory. In addition, the CSR edge list in the UVM address space is marked as `cudaMemAdviseSetReadMostly` using the `cudaMemAdvise()` CUDA API call. This optimization allows the GPU to create a read-only copy of the accessed pages in the GPU's memory. We also tested other available UVM driver flags but did not observe notably differences. **(b) Naive** implementation is the baseline implementation of EMOGI using zero-copy memory and is identical to Algorithm 1. In this implementation, the vertex list is stored in the GPU memory while the edge list is kept in the zero-copy host memory. **(c) Merged** implementation of EMOGI merges the memory requests to the zero-copy memory, as discussed in § 4.3.1. However, in this implementation, there is no guarantee that accesses to the zero-copy memory are aligned. **(d) Merged+Aligned** implementation is the fully optimized version of EMOGI where the memory accesses are not only merged but we force all warps to shift to the 128-byte boundary when there is a misalignment. This implementation is discussed in § 4.3.2.

5.2 Evaluation Datasets

For the evaluation, we use the graphs listed in Table 2. GK, GU, FS, and ML are the largest four graphs from SuiteSparse Matrix Collection [18] and SK, and UK5 are commonly used large graphs from LAW [13]. This collection of graphs covers data from different areas such as biomedicine, social networks, web crawls, and even synthetic graphs. All the graphs, except for SK and UK5, are undirected. We use the default weights for GU, GK, and ML graphs while we randomly initialize weights for the rest of the graph from the integer values between 8 to 72. The average degree of the graphs is 38, except for the ML graph, which has an average degree of 222. For fair BFS and SSSP performance evaluations, we pick 64 random vertices from each graph as the starting sources and reuse the selected vertices for all measurements. The final execution time is calculated by averaging the execution times of the 64 cases, but some results are removed from the average when the selected vertices have no outgoing edges. Edge weight values are only used by the SSSP algorithm.

5.3 Case-Study: Breadth-First Search

In this section, we take BFS as an example and thoroughly evaluate PCIe traffic for request size distribution, achieved bandwidth, and the total amount of data transferred. Throughout the evaluation, we use the UVM implementation as the baseline.

Table 2: Graph Datasets. V = Vertex, E = Edge, and w = Weight.

Sym.	Graph	Number		Size (GB)	
		$\|V\|$	$\|E\|$	$\|E\|$	$\|w\|$
GK	GAP-kron [12]	134.2M	4.22B	31.5	15.7
GU	GAP-urand [12]	134.2M	4.29B	32.0	16.0
FS	Friendster [65]	65.6M	3.61B	26.9	13.5
ML	MOLIERE_2016 [61]	30.2M	6.67B	49.7	24.8
SK	sk-2005 [14–16]	50.6M	1.95B	14.5	7.3
UK5	uk-2007-05 [15, 16]	105.9M	3.74B	27.8	13.9

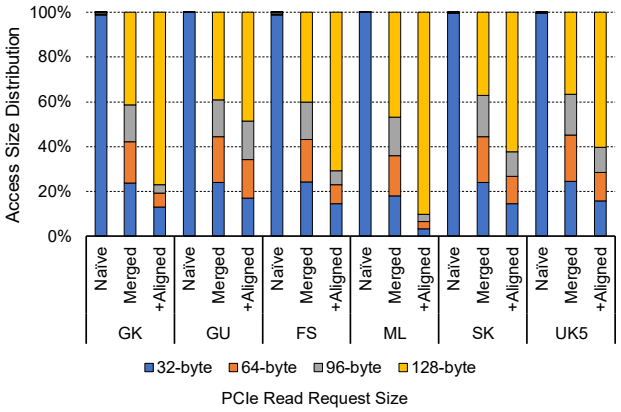

Figure 5: Distribution of PCIe read request sizes in BFS. `+Aligned` is abbreviation for `Merged+Aligned`. As the merged and aligned optimizations are added, the BFS application generates more 128-byte requests for efficient access.

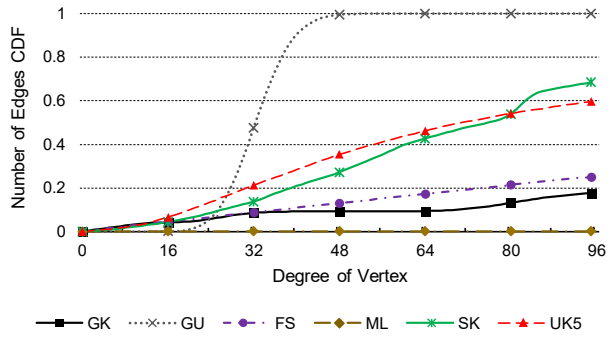

Figure 6: Number of edges CDF of evaluation graph. This plot provides us a better understanding of the distribution of the neighbor list sizes in the graphs. For example, the GU graph has all of its edges associated with vertices with degree between 16 and 48, meaning the neighbor lists contain at most 48 neighbors.

5.3.1 Zero-copy Request Size Distribution: In this evaluation, we show the impact of optimizing the memory access pattern from § 3.3 on generating different sizes of PCIe request. The histogram of the PCIe request size is gathered using the FPGA monitoring platform explained in § 3.2. In Figure 5, we show the breakdown of request

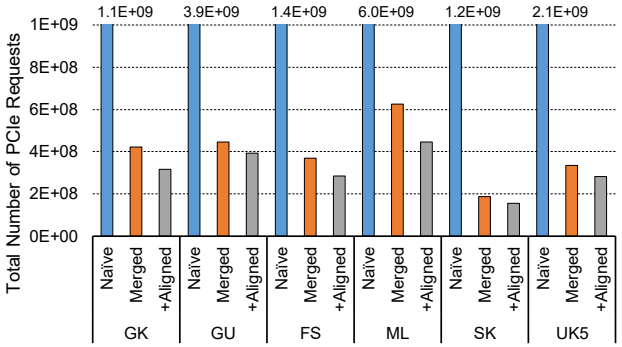

Figure 7: Number of PCIe requests sent for `Naive`, `Merged` and `Merged+Aligned` implementations while executing BFS on various graph. Collected from FPGA. `Merged` optimization reduces the PCIe memory requests by up to 83.3% compared to the `Naive` implementation. `Merged+Aligned` optimization can further reduce the PCIe memory requests by up to 28.8%. `+Aligned` is abbreviation for `Merged+Aligned`.

sizes for all the PCIe requests from the three implementations: `Naive`, `Merged`, and `Merged+Aligned`.

We observe in Figure 5 that nearly all PCIe requests in the case of `Naive` implementation are of 32-byte granularity. This is because it is only possible to generate a PCIe request larger than 32-byte in the `Naive` implementation when multiple neighbor lists happen to be spatially near in the edge list and they are accessed by multiple threads in a single warp. However, such a scenario is extremely unlikely. For example, we observe that only 1.3% of the PCIe requests from BFS on the FS graph are of a size bigger than 32-bytes.

When we analyze the request size distribution for the `Merged` and `Merged+Aligned` optimized implementations, we observe the following. First, although with the `Merged` approach the percent of 128-byte requests increases to about 40% on average, the percent of 128-byte requests is slightly higher than average for the ML graph, at about 46.7%. Second, when using the `+Aligned` approach on graphs that have most of their edges associated with high-degree vertices, we expect that most zero-copy memory requests should be for 128-bytes. This is expected because in the `+Aligned` implementation, zero-copy memory requests are merged and aligned to 128-byte granularity whenever possible. We observe this behavior for most graphs in Figure 5. For example, the percent of 128-byte requests improves by 1.86× for the GK graph between the `Merged` and `+Aligned` implementations. However, the percent of 128-byte requests improves by only 1.25× between the two implementations on the GU graph, a graph that has a similar number of edges and vertices as the GK graph.

To further analyze these behaviors, we plot in Figure 6, the cumulative distribution function (CDF) on the number of edges in each graph. CDF on the number of edges provides us a better understanding of the distribution of the neighbor list sizes in the graph. The horizontal axis of this CDF is cut to 96 as many of the graphs have vertices with an extremely high degree. From Figure 6, we see that the ML graph has nearly no edges associated with small degree vertices. Thus, with the `Merge` optimization many requests can be merged to 128-bytes for the ML graph. The other graphs, like

FS, have some edges associated with small degree vertices. Thus not all of their requests can be merged. Due to the fact that most vertices have long neighbor lists in the ML graph, the `+Aligned` optimization further maximizes the 128-byte zero-copy accesses, as shown in Figure 5, and, as a result, reduces the total number of zero-copy memory requests by 28.8%, as shown in Figure 7.

To understand why the request size distribution of GK and GU graphs are significantly different for the `+Aligned` optimization, we need to understand the neighbor list size distributions of these graphs. The neighbor lists of the GK graph are extremely unbalanced while the GU graph has uniformly low degrees varying from 16 to 48. If we assume the starting location of each neighbor list is uniformly random, then the chance of each neighbor list starting at the exact 128-byte boundary is only 6.25% when the data type size is 8-bytes. Therefore, in most cases, the neighbor lists of graphs are not aligned at the 128-byte boundary by default. If the neighbor list sizes are extremely unbalanced, like in GK, then the misalignment is less problematic since the vertices with high degrees can amortize the cost of the one-time misalignment fix. However, if all vertices have uniformly low degrees, like in GU, then there is no opportunity to amortize the cost of the one-time misalignment fix per vertex. Due to this, among all the graphs evaluated, only GU shows very little improvement with the `+Aligned` optimization.

5.3.2 PCIe Bandwidth Analysis: The bandwidths we measured are more or less aligned with PCIe request size distributions. In Figure 8, we show the average achieved PCIe bandwidth while executing BFS. We measured the maximum achievable PCIe bandwidth with `cudaMemcpy()` to be 12.3GB/s. Because of the page faulting overhead present in the UVM, it can only achieve PCIe bandwidth of 9GB/s. EMOGI's `Naive` implementation of BFS can only reach up to 4.7GB/s PCIe bandwidth. This is in sync with what we observed using the toy example in Figure 4. With the `merge` optimization, the PCIe bandwidth utilization increased up to 11GB/s, reaching about 90% of the peak `cudaMemcpy()` bandwidth. With the `Merged+Aligned` optimization, we add about 0.5 to 1GB/s of additional bandwidth utilization on top of `merge` optimization in all cases. The GU graph has the least amount of improvement from the alignment optimization among all graphs. This is because BFS on the GU graph cannot send enough number of 128-byte requests to saturate PCIe interconnect bandwidth. By comparing Figure 5 and Figure 8, we can clearly see the correlation between the distribution of PCIe request sizes and the achieved bandwidths in a real application, thus confirming our the analysis in § 3.3.

5.3.3 Analysis of Zero-copy Optimizations: We next evaluate the performance difference between `Naive`, `Merge` and `Merge+Aligned` implementation of BFS on various graphs and compare it with the UVM implementation. The performance is measured based on the traversed edges per second (TEPS) and is inversely proportional to the execution time. As shown in the Figure 9, the `Naive` implementation's performance is 0.73× of that of UVM on average. As discussed in § 3.3, this is expected as the `Naive` implementation does not use the PCIe bandwidth efficiently. On the other hand, merging requests that go to zero-copy memory with the `Merged` implementation provides a speedup of 3.24× over the UVM baseline on average. For the SK graph, the performance gain using the `Merged` optimization is only 1.21× over UVM. This is because the SK graph

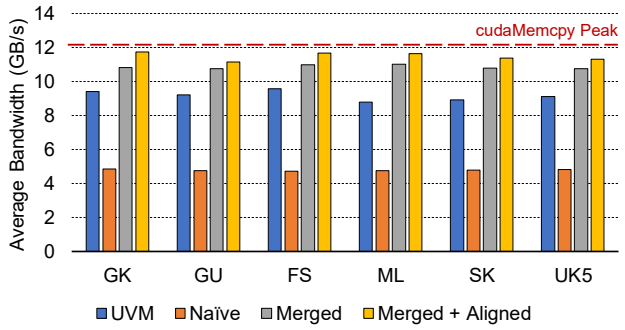

Figure 8: Average PCIe 3.0 x16 bandwidth utilization of the different implementations executing BFS.

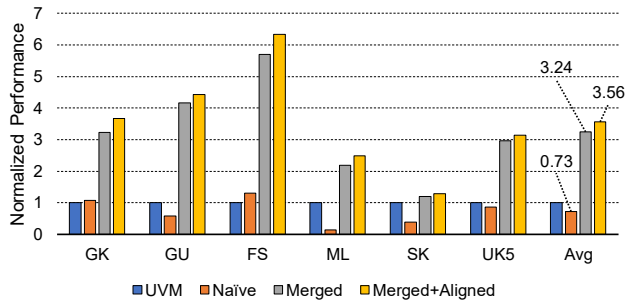

Figure 9: BFS performance of the `Naive`, `Merged` and `Merged+Aligned` implementations against the UVM baseline.

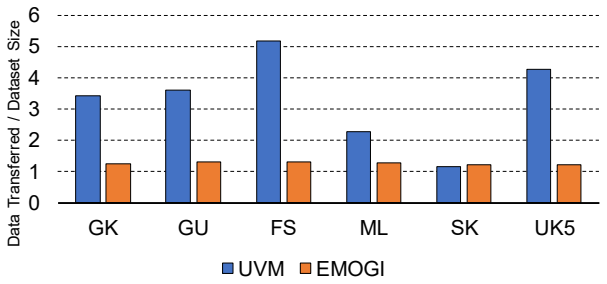

Figure 10: I/O Read Amplification of EMOGI and the UVM baseline while performing BFS.

can almost fit in the 16GB GPU memory. When we add memory access alignment optimization on top of merging of request with the `Merged+Aligned` implementation, we notice a 1.10× improvement in performance over the `Merged` implementation on average. This improvement can be associated with the reduced number of PCIe requests that go out to the zero-copy memory because of the `Merged+Aligned` optimization, as was shown in Figure 7.

5.3.4 I/O Read Amplification: We now demonstrate the I/O read amplification benefit of EMOGI's fine-granular data accesses over the 4KB page movement in UVM in BFS graph traversal. For this experiment, we chose the `Merge+Aligned` EMOGI implementation to represent EMOGI as it provides the best performance. Figure 10 shows the ratio of data read from the host memory over the dataset size while performing BFS using UVM and EMOGI on each graph. UVM generally has a very high I/O read amplification factor, up to 5.16× for the FS graph, as for these graphs, the neighbor lists

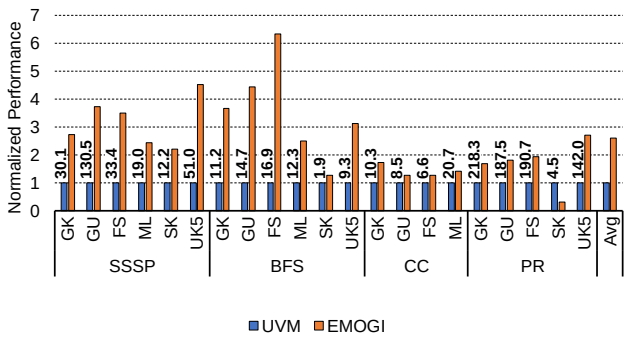

Figure 11: Performance comparison between UVM and EMOGI with different graph traversal applications with V100. Actual execution times of `UVM` cases are written on top of the bars (in seconds).

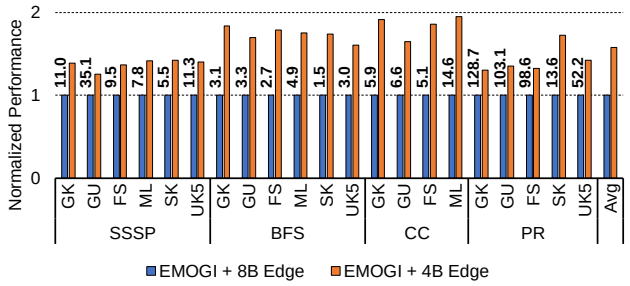

Figure 12: Performance comparison between using 4B edge and 8B edge for EMOGI with V100. Actual execution times of `EMOGI + 8B` cases are written on top of the bars (in seconds).

accessed during traversal are in different locations in memory and thus there is very little spatial locality exploited for each 4KB page moved. However, the two notable exceptions to this are the ML and SK graphs as UVM's I/O read amplification factor for them is 2.28× and 1.14×, respectively. This is because the average degree of a vertex in the ML graph is 222 and the SK graph is so small that it can almost fit in GPU memory, thus making UVM's page movements a little more efficient in both cases. In contrast, EMOGI's I/O read amplification factor doesn't exceed 1.31×. This is because the fine-granular, merged, and aligned data access to zero-copy memory allow EMOGI to efficiently move only the necessary bytes over the slow PCIe interconnect.

5.4 Beyond BFS

In this section, we apply EMOGI's optimization techniques to other graph traversal applications and measure their execution time. In addition to BFS from the previous sections, we add the single-source shortest path (SSSP), connected components (CC), and PageRank (PR) applications. We do not evaluate the performance of CC with SK and UK5 graphs as these graphs are directed. For PR, we do not evaluate ML graph as it is a multigraph. The overall performance results are shown in Figure 11.

EMOGI provides the best performance for all the graph traversal applications and graph datasets we studied. On average, EMOGI is 2.60× faster than UVM. For CC and PR, EMOGI shows relatively lower speed-ups over UVM than the other applications. In the case of SSSP and BFS, a specific vertex is selected as a root vertex and the applications start traversing the entire graph from the root vertex. However, with CC and PR, instead of picking a specific vertex to start with, all vertices are set as root vertices and the entire edge list is traversed. In this case, the application data access pattern is similar to streaming the edge list resulting in more spatial locality when compared to the other applications and less I/O read amplification for UVM.

Using smaller datatypes can reduce the overall PCIe traffic and therefore reduce the overall execution time as well. In Figure 12, we show the performance comparison of EMOGI when using 4-byte edge list vs. 8-byte edge list. On average, we observe about 1.57× of performance difference when using 4-byte over 8-byte for EMOGI. In case of GK and ML graphs in CC, the performance differences are nearly 2×. The performance differences in SSSP are relatively smaller compared to the other applications since SSSP needs to transfer the weight values as well. Due to the higher computation to memory ratio in PR [9], PR also shows relatively smaller performance differences.

5.5 Comparison with Previous Works

In this section, we compare EMOGI with the current state-of-the-art GPU solutions for out-of-memory graph traversals, HALO [23] and Subway [52]. Due to their varying runtime requirements, we also modify our EMOGI testing environment for accurate comparisons. The details of the modifications are described in the following sections.

5.5.1 HALO. HALO proposes a new CSR reordering method to improve data locality and data transfer during graph traversal with UVM. Since the source code of the HALO is not publicly available, we compare EMOGI with the results available in the published paper. As HALO's results were gathered using a Titan Xp GPU, we also use a Titan Xp instead of V100 for fair comparison and re-measure our execution times. The comparison results are shown in the Table 3. Overall, EMOGI shows 1.34× to 3.19× of speedups against HALO.

5.5.2 Subway. Subway proposes a design of graph partitioning that preprocesses to determine the activeness of a vertex. Instead of relying on UVM, Subway focuses on generating a temporal small CSR (also called subgraph) that fits in the GPU memory in each iteration. Since the original CSR is located in the host memory, CPUs need to be in charge of generating the temporal CSR for every iteration. To transfer the new temporal CSR, `cudaMemcpy()` is called by the host program.

We evaluate Subway using all the publicly available source codes (SSSP, BFS, and CC) with our platform described in § 5.1. Since one of the goals of EMOGI is to avoid any data manipulation, we include the subgraph generation time of Subway as well in our measurements. The publicly available implementation of Subway fails to execute on the GU graph due to unidentified CUDA out-of-memory errors and it cannot execute on the ML graph as the framework currently supports a maximum of 2^{32} edges. The comparison results are shown in the Table 4. Overall, across all the graph datasets and graph traversal algorithms, EMOGI observes speedups of 1.57× to 4.73×.

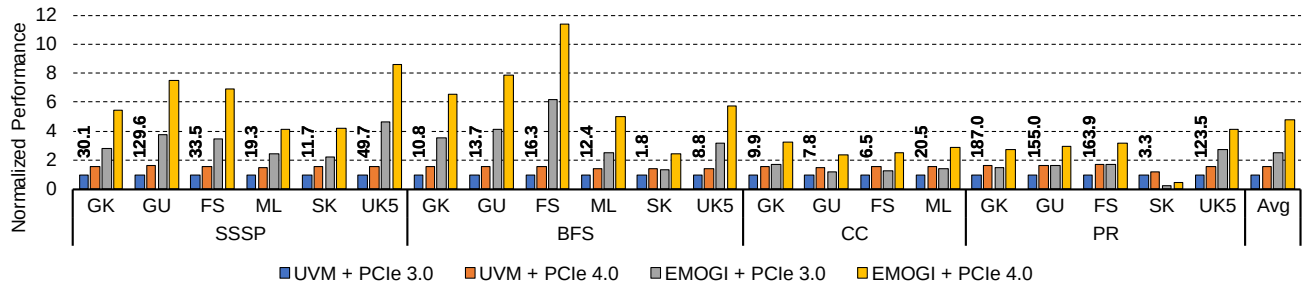

Figure 13: Performance comparison between UVM and EMOGI using PCIe 3.0 and PCIe 4.0. All results are measured in DGX A100. Actual execution times of `UVM + PCIe 3.0` cases are written on top of the bars (in seconds).

5.6 Performance Scaling with PCIe 4.0

As was shown in § 5.3.2 and § 5.3.3, EMOGI can nearly saturate the PCIe 3.0 bandwidth while out-performing the UVM implementation. NVIDIA's latest GPU, the Ampere A100, communicates with the host memory over the PCIe 4.0 interconnect. PCIe 4.0's measured peak bandwidth, approximately 24GB/s, is twice as much as PCIe 3.0's peak measured bandwidth of approximately 12GB/s. In this section, we study the ability of both UVM and EMOGI to take advantage of the increased bandwidth in accessing the host memory. To this end, we use a DGX A100 machine [7] with the A100 GPU and Dual AMD Rome 7742 CPUs paired with 1TB of system memory. This machine allows us to switch the root port to run in either PCIe 3.0 mode or PCIe 4.0 mode. Neither the EMOGI implementation nor the UVM implementation was re-optimized for the A100 GPU in these experiments. A100 memory is throttled to 16GB.

The overall evaluation results comparing the performance of UVM and EMOGI on the DGX A100 system are shown in Figure 13. Here, we normalize the performance speed-up achieved by each configuration to the UVM implementation running on the A100 GPU with the PCIe 3.0 interconnect. While EMOGI's performance scales by 1.88× on average with the faster interconnect, UVM's performance scales by only 1.53× on average. This is because the UVM implementation suffers from page fault handling overhead when accessing pages of the edge list in host memory. The page fault handler is part of the UVM driver running on the CPU and can't keep up to make use of the higher bandwidth of the PCIe 4.0 interface. However, EMOGI doesn't suffer any page faulting overhead as the edge list is pinned in host memory, leading to EMOGI's performance scaling almost linearly with the PCIe bandwidth.

Table 3: Execution time comparison with HALO [23]. NVIDIA Titan Xp (12GB) used.

Application	Graph	Exe. Time		Speedup
		HALO	EMOGI	
BFS	ML	9.54s	4.43s	2.15×
	FS	8.27s	2.59s	3.19×
	SK	2.17s	1.62s	1.34×
	UK5	6.03s	4.00s	1.51×

Table 4: Execution time comparison with Subway [52]. NVIDIA Tesla V100 (16GB) used. 4-byte edge used due to the Subway requirement.

Application	Graph	Exe. Time		Speedup
		Subway	EMOGI	
SSSP	GK	20.96s	7.94s	2.64×
	FS	14.95s	6.97s	2.14×
	SK	8.99s	3.92s	2.30×
	UK5	25.78s	8.08s	3.19×
BFS	GK	6.88s	1.66s	4.14×
	FS	4.22s	1.49s	2.83×
	SK	1.69s	0.85s	1.99×
	UK5	8.75s	1.85s	4.73×
CC	GK	6.34s	3.11s	2.04×
	FS	4.31s	2.75s	1.57×

6 DISCUSSION

Extending to other input formats: In this paper, EMOGI is targeting CSR which is used by many popular graph processing frameworks [1, 43, 44, 58, 63], but the main idea of EMOGI can be extended to different formats as well. The most immediate applicable format is compressed sparse column (CSC). The edgelists in CSR represent outgoing edges (push-based), but the edgelists in CSC represent incoming edges (pull-based). Although the directions are different between the CSR and CSC, the memory access pattern to the edgelists in both input formats is identical. Another interesting format expansion for EMOGI would be dynamic graphs [56] and compressed graphs [33, 57]. The graph input formats used in these works are not strictly identical to the classical CSR format, but their fundamental structures resemble CSR to retain some level of data locality for better bandwidth utilization. Therefore, the EMOGI's zero-copy memory access optimization strategies can be applied to these formats as well.

Additional optimizations: There are several additional optimizations available for EMOGI such as data compression and data caching in GPU global memory. Data compression on graph [33, 57] can reduce the total amount data transferred to GPU and we can obtain a similar effect to increasing the external interconnect bandwidth. As discussed in § 4.3, GPU is severely underutilized due to the low external interconnect bandwidth. Thus the idling GPU cores can be potentially used to decompress data from the host

memory, without interfering with the original graph traversing process. For data caching, a work similar to [38, 66] can be applied to exploit data locality further. For this optimization, we expect that a workload with high vertex revisits, such as PR, would benefit the most. However, one thing to note is that currently there is no full hardware-based mechanism to naturally use the GPU global memory as a large cache and therefore a software-based caching mechanism needs to be implemented.

7 RELATED WORKS

Graph Analytics on CPU: Efficient graph traversal algorithm implementation on CPUs have been extensively studied in the past[26, 44, 49, 54, 58–60]. We note that from these studies two popular frameworks Galois [44] and Ligra [58] have emerged. To compensate the low parallelism of CPU, these frameworks support the workload scaling to multi-CPU systems such as non-uniform memory access (NUMA) nodes. However, the focus of these prior work has been on efficient graph partition and data allocation schemes across different nodes to minimize the node-to-node data movements. Therefore, it is inevitable to do a pre-processing for efficient data organization. Beside the frameworks, there are other numerous works [26, 49, 54, 59, 60] based on CPU.

Graph Analytics on GPU: Graph traversal algorithms such as BFS exhibit a massive amount of parallelism. This has led to increasing research in leveraging the massive computation power offered by GPUs to speed up graph analytics. Prior work focused on improving the performance of graph traversal algorithms either by making GPU specific algorithmic improvements [20, 31, 36, 42, 63, 67, 69] or by performing data transformations [45, 47]. However, most of these works assume graphs fit in the GPU memory.

Practical graphs, on the other hand, often cannot fit into the GPU memory. Web graphs [15, 16], social network graphs [65] and biomedical graphs [12] can be significantly larger than available GPU memory (see Table 2). To address this, prior works have proposed either to partition the input graph and loading only those edges that are needed during computation [27, 52, 55, 63] or leveraging automatic memory oversubscription using UVM [19, 23, 25, 37, 39, 41, 50]. For example, GraphReduce [55] partitions the oversized graphs and does explicit memory management between the GPU and the host memory. Recently, Subway [52] further improved the design of the input partitioning scheme using GPU-accelerated subgraph generation pre-processing technique that tracks activeness of a vertex and also by generating subgraphs asynchronously. EMOGI does not perform any explicit memory management or pre-processing of the graph.

Alternatively, to support large graphs in GPU, programmers can use UVM which does automatic memory oversubscription [2, 3]. Prior works such as [10, 11, 19, 23, 25, 34, 37, 39, 41, 50] have observed significant overhead from UVM and have proposed optimizations such as overlapping IO and computation [25], memory spaces[19], memory throttling [39], modifying driver to support larger page fault batch sizes [37] and reordering of graphs to enhance locality in UVM [23]. Instead of leveraging the previously proposed optimizations, EMOGI takes a step back and revisits the reasoning behind the performance degradation with UVM. Like [68], EMOGI initially observes the PCIe bandwidth utilization being low for graph traversal applications. As shown in § 5, by carefully re-orchestrating the memory access pattern using direct access, EMOGI is able to boost graph traversal execution performance for large graphs without any additional optimizations. Indeed the prior proposed software and hardware optimizations can be exploited by EMOGI. We leave this as future research. Also, EMOGI could be easily incorporated into a library to lessen the programmer's effort and provide out-of-the box performance improvements.

Multi-GPU and Collaborative CPU-GPU: Aside from single GPU graph traversal, prior works have proposed using multi-GPU [35, 48, 63, 69] and collaborative CPU-GPU computation to meet the needs of large graphs computation [24, 25, 40, 41, 62]. Multi-GPU and collaborative CPU-GPU computing are enabled using UVM where hardware moves the pages on-demand across different computing blocks. EMOGI can be extended to support both multi-GPU and hybrid CPU-GPU computing and we leave it as future research.

Architectural support for improving UVM: Besides algorithm and system-level changes, prior works also propose hardware changes that can enable executing graph traversal algorithms efficiently on large graphs. Specifically, memory compression techniques to reduce the memory footprint in the GPU [39], efficient migration policies using hardware counters, and optimized prefetching schemes [10, 11, 21], and software-hardware co-design using memory hints are proposed [3, 6]. These techniques are orthogonal to EMOGI and can be leveraged by EMOGI to gain further performance improvements in future GPU architectures.

8 CONCLUSION

In this work, we present EMOGI, a new method for optimizing the traversal of very large graphs with a GPU using zero-copy. Using thorough analysis of fine-grained GPU memory access patterns over PCIe to zero-copy memory, we identified key optimizations to best utilize bandwidth to zero-copy memory: merged and aligned memory accesses. We applied these optimizations to key graph traversal applications to enable efficient GPU traversal of graphs that do not fit in GPU memory. Our experiments show that EMOGI out-performs the state-of-the-art solutions for traversing larges graphs. This is because EMOGI avoids I/O read amplification by leveraging efficient fine-grained accesses to fetch only the needed bytes from zero-copy memory. Furthermore, EMOGI's performance scales almost linearly with the improved bandwidth of newer interconnects as it is not bottle-necked by the page fault handling overhead of traditional methods using UVM.

ACKNOWLEDGMENTS

This work was partially supported by the Applications Driving Architectures (ADA) Research Center and Center for Research on Intelligent Storage and Processing-in-memory (CRISP), JUMP Centers co-sponsored by SRC and DARPA, IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR). This work would not have been possible without the generous hardware donations from Xilinx and NVIDIA. We also thank the IMPACT members, anonymous reviewers, and a shepherd for their constructive feedbacks, and the chief editors' coordination in the shepherding process.

REFERENCES

[1] [n.d.]. nvGRAPH. https://developer.nvidia.com/nvgraph.

[2] 2016. NVIDIA Tesla P100 Architecture Whitepaper. https://www.nvidia.com/object/pascal-architecture-whitepaper.html.

[3] 2017. NVIDIA Tesla V100 GPU Architecture Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[4] 2020. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[5] 2020. Intel® VTune™ Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[6] 2020. NVIDIA A100 GPU Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[7] 2020. NVIDIA DGX A100 Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[8] 2020. PCIe 3.0 Specification. https://members.pcisig.com/wg/PCI-SIG/document/download/8257.

[9] T. K. Aasawat, T. Reza, and M. Ripeanu. 2018. How Well do CPU, GPU and Hybrid Graph Processing Frameworks Perform?. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 458–466.

[10] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page Placement Strategies for GPUs within Heterogeneous Memory Systems. *SIGARCH Comput. Archit. News* 43, 1 (March 2015), 607–618.

[11] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 136–150.

[12] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[13] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: a scalable fully distributed Web crawler. *Software: Practice and Experience* 34, 8 (2004), 711–726.

[14] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. 2004. UbiCrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34, 8 (2004), 711–726.

[15] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th international conference on World Wide Web*, Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar (Eds.). ACM Press, 587–596.

[16] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. ACM Press, Manhattan, USA, 595–601.

[17] S. Che. 2014. GasCL: A vertex-centric graph model for GPUs. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6.

[18] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages.

[19] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[20] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: EXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 121–131.

[21] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami 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* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 224–235.

[22] Debashis Ganguly, Z Zhang, J Yang, and Rami Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In *Proceedings of the Thirty-forth International Conference on Parallel and Distributed Processing (IPDPS)*.

[23] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. 2020. Traversing Large Graphs on GPUs with Unified Memory. *Proceedings of the VLDB Endowment* 13, 7 (March 2020), 1119–1133.

[24] 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 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 345–354.

[25] J. Gómez-Luna, I. E. Hajj, L. Chang, V. García-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Peña, and W. Hwu. 2017. Chai: Collaborative heterogeneous applications for integrated-architectures. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 43–54.

[26] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13.

[27] W. Han, D. Mawhirter, B. Wu, and M. Buland. 2017. Graphie: Large-Scale Asynchronous Graph Traversals on Just a GPU. In *Proceedings of 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 233–245.

[28] Pawan Harish and P. J. Narayanan. 2007. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing* (Goa, India) *(HiPC'07)*. Springer-Verlag, Berlin, Heidelberg, 197–208.

[29] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://devblogs.nvidia.com/unified-memory-cuda-beginners/.

[30] K. A. Hawick, A. Leist, and D. P. Playne. 2010. Parallel Graph Component Labelling with GPUs and CUDA. *Parallel Comput.* 36, 12 (Dec. 2010), 655–678. https://doi.org/10.1016/j.parco.2010.07.002

[31] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 267–276.

[32] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. 2011. Accelerating CUDA Graph Algorithms at Maximum Warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 267–276.

[33] Krzysztof Kaczmarski, Piotr Przymus, and Paweł Rzążewski. 2015. Improving high-performance GPU graph traversal with compression. In *New Trends in Database and Information Systems II*. Springer, 201–214.

[34] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping. *SIGPLAN Not.* 50, 7 (March 2015), 65–77.

[35] F. Khorasani, R. Gupta, and L. N. Bhuyan. 2015. Scalable SIMD-Efficient Graph Processing on GPUs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 39–50.

[36] 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* (Vancouver, BC, Canada) *(HPDC '14)*. Association for Computing Machinery, New York, NY, USA, 239–252.

[37] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1357–1370.

[38] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna. 2017. ReCALL: Reordered Cache Aware Locality Based Graph Processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 273–282.

[39] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 49–63.

[40] 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 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 195–207.

[41] 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 *2018 IEEE High Performance extreme Computing Conference (HPEC'18)*. Boston, USA.

[42] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-Performance and Scalable GPU Graph Traversal. *ACM Transactions on Parallel Computing* 1, 2, Article 14 (Feb. 2015), 30 pages.

[43] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 69, 12 pages.

[44] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*.

Association for Computing Machinery, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[45] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. *SIGPLAN Not.* 53, 2 (March 2018), 622–636.

[46] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report 1999-66. Stanford InfoLab. http://ilpubs.stanford.edu:8090/422/ Previous number = SIDL-WP-1999-0120.

[47] Sreepathi Pai and Keshav Pingali. 2016. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. *SIGPLAN Not.* 51, 10 (Oct. 2016), 1–19.

[48] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens. 2017. Multi-GPU Graph Analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 479–490.

[49] R. Pearce, M. Gokhale, and N. M. Amato. 2010. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[50] Carl Pearson, Mohammad Almasri, Omer Anjum, Vikram S Mailthody, Zaid Qureshi, Rakesh Nagi, Jinjun Xiong, and Wen-mei Hwu. 2019. Update on Triangle Counting on GPU. In *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*. Boston, USA.

[51] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. http://networkrepository.com

[52] 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 Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 16 pages.

[53] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proceedings of the VLDB Endowment* 11, 4 (Dec. 2017), 420–431.

[54] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1, 3-4 (2016), 145–164.

[55] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. 2015. GraphReduce: Processing Large-Scale Graphs on Accelerator-Based Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 28, 12 pages.

[56] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 107–120. https://doi.org/10.14778/3151113.3151122

[57] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-Based Graph Traversal on Compressed Graphs. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 775–792. https://doi.org/10.1145/3299869.3319871

[58] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[59] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 451–462.

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

[61] Justin Sybrandt, Michael Shtutman, and Ilya Safro. 2017. MOLIERE: Automatic Biomedical Hypothesis Generation System. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Halifax, NS, Canada) *(KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1633–1642.

[62] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "Think like a Vertex" to "Think like a Graph". *Proceedings of the VLDB Endowment* 7, 3 (Nov. 2013), 193–204.

[63] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: A High-Performance Graph Processing Library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 11, 12 pages.

[64] Zhen Xu, Xuhao Chen, Jie Shen, Yang Zhang, Cheng Chen, and Canqun Yang. 2017. GARDENIA: A Domain-specific Benchmark Suite for Next-generation Accelerators. *CoRR* abs/1708.04567 (2017). arXiv:1708.04567 http://arxiv.org/abs/1708.04567

[65] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities based on Ground-truth. *CoRR* abs/1205.6233 (2012). arXiv:1205.6233 http://arxiv.org/abs/1205.6233

[66] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302.

[67] 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 Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 601–614.

[68] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357.

[69] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Transactions on Parallel and Distribution Systems* 25, 6 (June 2014), 1543–1552.