© 2022 Seung Won Min

# FINE-GRAINED MEMORY ACCESS OVER I/O INTERCONNECT FOR EFFICIENT REMOTE SPARSE DATA ACCESS

BY

SEUNG WON MIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Wen-mei Hwu, Chair
Professor Deming Chen
Dr. I-Hsin Chung, IBM Research
Assistant Professor Jian Huang
Professor Sanjay Patel

# ABSTRACT

The combination of the growing size and complexity of application datasets is introducing a new challenge to accelerators. The growing size of datasets forces us to place them in a larger CPU memory, and the growing complexity of datasets introduces more irregularity in data access patterns. However, the existing data transfer mechanisms are optimized toward transferring regular and densely accessed datasets, and not for the complex and sparse datasets. With the existing methods, now the accelerators often spend more time accessing complex datasets stored in CPU memory rather than actually doing the computation.

To overcome the limitations of the existing data transfer mechanisms, this dissertation proposes to utilize many fine-grained memory accesses over I/O interconnect, such as the industry standard PCIe, instead of the traditional coarse-grained block data transfer method. While the fine-grained memory access over I/O interconnect poses a danger of introducing high per-packet overhead, the benefit from its flexibility in accessing complex data structures can outweigh the overhead. To accurately evaluate the overhead and benefit of the fine-grained memory access over I/O interconnect, we begin by developing a methodology to directly analyze I/O traffic. While the direct I/O level analysis is not a prevalent approach in the current academic research, the existing indirect application-level analysis approach is insufficient to fully capture the intricacy of I/O behaviors. We fill this gap by designing our own custom I/O analyzer using a field-programmable gate array (FPGA) and demonstrate how the potential overhead of the fine-grained access over I/O interconnect can be identified and avoided. Based on the insights we gained from the analysis, we redesign and optimize several real-world applications using the fine-grained memory access over I/O interconnect and show that we can speed up the applications several times over the existing methods.

Next, this dissertation addresses the question of integrating the fine-grained

memory access over I/O interconnect into the existing software development environment. Since most programmers may not have a deep hardware-level understanding of the fine-grained memory access over I/O interconnect, it is necessary to abstract away any optimizations that require deep understanding of the hardware and provide these optimizations in libraries and frameworks. To achieve this goal, in our work, we abstract away the hardware-level optimizations behind our custom array class called UnifiedTensor, and transparently apply the optimizations whenever remote memory accesses are done through this class. With the help of the abstraction, only about 2-3 lines of code modifications are sufficient to fully utilize our method for most of the existing programs. At the same time, we also provide a flexible development environment to enable quick deployment of new hardware optimizations for the framework developers.

Finally, we conclude this dissertation by proposing a flexible data tiering strategy in modern systems with the fine-grained memory access over I/O interconnect. While there are multiple tiers of memory in modern computer systems, currently partitioning sparse datasets over multiple tiers of memory and seamlessly accessing them in applications requires a lot of programming effort. To overcome the programming difficulty, we unify all data access methods to different tiers of memory with the fine-grained memory access over I/O interconnect. This not only keeps the overall application structure concise but also allows the programmers to quickly try out different data partitioning strategies in favor of achieving better data locality.

*To my family, for their unwavering trust and support.*

# ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor Wen-mei Hwu, who has always been my great supporter and the champion of my work. Professor Hwu not only provided intellectual and experienced guidance over technical matters, but also carefully listened to personal matters. His continuous interest and attention were undoubtedly the best encouragement in my research, and it is difficult to imagine how I would have finished this dissertation without them. I would also like to thank him for teaching the importance of working on impactful matters, which always helped me stay on the right track.

I would also like to thank Professor Chen, Professor Xiong, and Doctor Chung, who provided invaluable thoughts. I have received much help from Professor Chen, Professor Xiong, and Doctor Chung over the last several years, during our collaboration in the IBM-Illinois C3SR center. The collaboration gave me a privileged opportunity to bring my research to the next level, for which I feel very fortunate. Many of the findings in this dissertation simply would not have been possible without their help.

I am grateful to my examination committee members, Professor Huang and Professor Patel as well, for their valuable feedback. I was able to polish my work further with their help.

I cannot miss other industrial collaborators Eiman Ebrahimi, Paul Hartke, and Xiang Song, who all shared important perspectives which are easily forgotten in academia. Their views helped me think about how to make an impact on the real-world problems.

I also had wonderful colleagues in our IMPACT research group, who were all very supportive and helpful. Not only that, talking and making jokes with them were among the funnest experiences during my academic career, and something that I cannot forget. I would also like to thank Marie-Pierre Lassiva-Moulin, Andrew Schuh, and Karoline Kao, our awesome IMPACT research group assistants, for their support.

Lastly, I would like to thank my family for their support. It was a long journey until this point, and only with them was I able to get this far.

# TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION . . . . . . . . . . . . . . . . . . . . . 1

CHAPTER 2 ANALYSIS AND CASE-STUDY . . . . . . . . . . . . 5
2.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Zero-Copy . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.4 EMOGI: Zero-Copy Graph Traversal . . . . . . . . . . . . . 18
2.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.6 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.7 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

CHAPTER 3 FRAMEWORK INTEGRATION . . . . . . . . . . . . 39
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.3 Fine-Grained and GPU-Oriented Data Communication Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
3.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73

CHAPTER 4 DATA ACCESS OPTIMIZATION IN MULTI-LEVEL MEMORY HIERARCHY . . . . . . . . . . . . . . . . . . . . . . 74
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.2 Data Tiering . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.3 Data Placement and Access . . . . . . . . . . . . . . . . . . 81
4.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

CHAPTER 5 CONCLUSION . . . . . . . . . . . . . . . . . . . . . 96

REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98

# CHAPTER 1

# INTRODUCTION

With the rapid growth of compute-intensive applications, using dedicated hardware accelerators such as graphics processing units (GPUs) has become essential for modern computing systems. These accelerators are capable of delivering a massive amount of computation power and are equipped with extremely high bandwidth memory devices to sustain high computation throughput. However, often such high bandwidth memory devices come with small capacity, and their expandability is limited due to their physical properties. To overcome such limitations, user applications must place datasets in a larger CPU memory and feed them to the accelerators during the runtime. While this is a common practice for many existing real-world applications, the increasing use of large sparse datasets is introducing a new challenge in the current paradigm.

Modern analytics and recommendation systems place great emphasis on analyzing not only the individual entities but also the relations of those entities in large-scale databases [1, 2]. To capture such relational information, the use of sparse datasets like graphs is becoming important. However, compared to accessing dense datasets, accessing sparse datasets often generates many irregular and fine-grained memory accesses. Such data access patterns are highly undesirable for the existing CPU-accelerator data communication methods since they only focus on transferring a single large chunk of data to mitigate the overhead of launching direct memory access (DMA).

To address such discrepancy, several alternatives have been introduced in the past [3, 4]. The introduced alternatives either: (1) aim to transfer a large chunk of data regardless of the size of the immediately requested data in the hope of having some spatial locality or (2) attempt to dynamically preprocess the input data into a preferable format for the data transfer. Unfortunately, neither method provides a meaningful efficiency improvement, and both methods leave the accelerators heavily throttled by the CPU-accelerator

data communication.

With the current inefficiency of coarse-grained data access over I/O interconnect, it is inevitable to question the current approach's practicality and study the fundamental interaction between the interconnect and the accelerators from scratch. The suboptimal application performances with the sparse datasets motivate us to look in the opposite direction and use finer-grained memory access over I/O interconnect instead. This dissertation's key contributions lie in analyzing and proving the feasibility of the fine-grained memory access over I/O interconnect (hereafter referred to as "I/O") in real-world applications, as we describe next.

**Systematic Analysis of Fine-Grained Memory Access Over I/O:** While fine-grained memory access over I/O poses a danger of introducing high per-packet overhead, the benefit from its flexibility in accessing complex data structures can outweigh the overhead. Thus, the major technical concern that we need to address in this dissertation is whether a large number of overlapping fine-grained memory accesses over I/O can be sustained to:

- Tolerate the long latency of I/O
- Fully utilize the available I/O bandwidth
- Ultimately, achieve good application performance

Unfortunately, evaluating these items would require a capability to probe low-level I/O interactions, which can be difficult in academia. In the presence of the challenge, several previous works [5, 6] instead attempted to perform high-level analysis by simply measuring application execution times, but we find the application execution time to be insufficient to reveal exact I/O-level details. To overcome this limitation, we introduce a field-programmable gate array (FPGA) based PCIe switch, which is an accessible PCIe research platform. The FPGA switch mimics the behavior of regular PCIe switches, and it can be immediately deployed into the existing systems without incurring additional software and hardware modifications. We also leverage embedded CPUs in FPGAs and allow FPGA programmers to offload several control-heavy functionalities into C/C++ programs for flexible and user-interactive design. The use of embedded CPUs is not mandatory for our FPGA switch design, and FPGAs without embedded CPUs can still utilize our soft-IP-

based implementation. We use this device throughout the experiments in this dissertation to monitor and understand the detailed I/O behaviors.

**Demonstration in Real-World Applications:** To prove the feasibility of fine-grained memory access over I/O in real-world applications, we demonstrate it using GPUs with graph analytic applications. In our demonstration, we reprogram existing graph analytic applications by replacing block data transfer methods with *zero-copy* access. Zero-copy access comes with various I/O access sizes such as 32, 64, 96, and 128 bytes, and it allows GPUs to choose the access size programmatically. With such fine-grained I/O access sizes, we show that zero-copy access is very effective in accessing sparse datasets without making excessive data transfer over I/O. Furthermore, by removing unnecessary data transfer preprocessing, we show that our implementation using zero-copy access provides better performance scalability than the baseline implementation with increasing I/O bandwidth. The use of the FPGA switch additionally reveals several optimization opportunities such as memory access alignment and defragmentation for higher I/O bandwidth utilization.

**Framework Integration:** The use of fine-grained memory access over I/O may require very different flows of memory allocation, memory mapping, and function calls from the traditional block data transfer method. For example, enabling zero-copy access for NVIDIA GPUs requires specific types of CUDA APIs to be called in a particular order, and adding case-specific zero-copy access performance optimizations requires inserting arbitrary code blocks flexibly at the framework level. However, the current implementations of existing Python-based frameworks such as PyTorch, Tensorflow, and MXNet only define the block data transfer method in their programming models.

To overcome such limitations, we introduce a new tensor class called Unified Tensor, which is specialized to support unconventional cross-device memory accesses for the existing frameworks. From the framework development perspective, Unified Tensor provides a more flexible flow of deploying architecture-and-device-specific system APIs and optimizations, which was impossible in the previous block data transfer flow. From the framework users' perspective, Unified Tensor works similarly to the existing array type of data structures, and users can dereference data allocated by Unified Tensor by simply passing indices to it. The enablement of Unified Tensor is easy,

and with several examples, we show that modifying only about 2-3 lines of code in the user-level program is sufficient.

The introduction of Unified Tensor is already making an impact on the industry. Unified Tensor has become one of the core methods for remote data accesses in the Deep Graph Library (DGL) [7] framework, and it replaces the conventional block data transfer method for large-scale graph neural network (GNN) training. The usage of Unified Tensor is expanding now, and several follow-up projects are proposed from the industry based on Unified Tensor to further optimize the data transfer between CPU and accelerator for different types of sparse datasets.

**Data Access Optimization in Multi-Level Memory System:** A modern system's memory hierarchy may contain several types of memory-attached devices such as host, local, and peer devices. Often those memory devices are connected to each other using different grades of interconnects, and optimizing data placement among those memory devices can have a significant impact on application performances [8]. While partitioning dense datasets over multiple memory devices is straightforward, it is less clear to partition sparse datasets over multiple memory devices due to the irregular memory access pattern. However, with the use of fine-grained memory access over I/O, this challenge can be mitigated.

Our work proposes unifying all memory access types with fine-grained memory access over I/O, such as zero-copy access, to simplify memory access to different kinds of memory devices. The use of fine-grained memory access over I/O eliminates the need for runtime sparse dataset transformation for the data transfers and significantly reduces the programming complexity. With the reduced programming complexity, programmers can attempt more advanced data placement strategies for different memory devices to better utilize high-bandwidth interconnects in the system. We demonstrate our work using graph neural networks (GNNs) training with sparse datasets reaching several hundreds of gigabytes and show how the advanced data placement strategies with the fine-grained memory access over I/O can dramatically improve the application performance.

# CHAPTER 2

# ANALYSIS AND CASE-STUDY

## 2.1 Introduction

Graph workloads are becoming increasingly widespread in various applications such as social network analysis, recommendation systems, financial modeling, biomedical applications, graph database systems, web data, geographical maps, and many more [9, 10, 11, 12, 13, 14, 15, 16]. Graphs used in these applications often come in huge sizes. A recent survey conducted by the University of Waterloo [9] finds that many organizations use graphs that consist of billions of edges and consume hundreds of gigabytes of storage.

Graph application developers currently face the main challenge of performing graph traversals on such large graphs [9]. GPUs are increasingly used to perform graph analytics because of the massive parallel computation opportunities in graph traversals. 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 CPU (host) memory.

For efficient storage and access, graphs are typically stored in compressed forms such as the sparse row (CSR) data format to reduce memory overhead. In the 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 [17, 3, 18, 19, 20, 21, 22, 23]. UVM brings CPU and GPU memory into a single shared address space. UVM allows GPUs to access the data in the unified virtual address space simply, and it transparently migrates required pages between CPU memory and GPU memory using a paging mechanism.

However, several prior works [3, 18, 19, 20, 21, 22, 23] 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 with at least 1M vertices and edges from LAW [11], SuiteSparse Matrix Collection [10], and Network Repository [12], we find that 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 page, as in the case of UVM, can cause memory thrashing and unnecessary I/O read amplification.

As a result, prior works have also proposed pre-processing of input graphs by partitioning and loading those edges that are needed during the computation [24, 25, 26, 4], UVM-specific hardware or software changes such as locality enhancing graph reordering [3], GPU memory throttling [18, 19], overlapping compute and I/O [27], or even a new prefetching policies in hardware that can increase data locality in GPU memory [21, 22, 23].

In this work, we take a step back and revisit the existing hardware memory management mechanism for data that does not fit in GPU memory. Specifically, we focus on zero-copy memory access, allowing GPUs to directly access the host memory in cacheline 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, allegedly due to the low external interconnect bandwidth [5]. 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 cacheline-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 allow GPU threads to directly perform cacheline-sized

accesses to data stored in host memory during graph traversals.

Using a toy example, we show that the system cannot saturate the PCIe 3.0 x16 bandwidth by naïvely enabling zero-copy (see Section 2.3.3). To address this shortcoming, 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 requests to zero-copy memory (see Section 2.3.3). Second, we propose the forced memory access alignment which shifts all warp memory accesses to 128-byte boundaries when misalignment occurs. This is because the memory access merge optimization does not guarantee memory request alignment, and such misalignment can result in performance degradation. While these optimizations sacrifice some parallelism and incur additional control divergences during kernel execution, their benefit in improved bandwidth utilization 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 2.2 on page 25), 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 various 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 [3, 4] for large out-of-memory graph traversals. In addition, EMOGI does not require pre-processing or a 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 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, 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 chapter is organized as follows: We provide a brief primer on GPU-based graph traversal and the challenges in executing graph traversals using UVM in Section 2.2. We then discuss how to enable zero-copy memory with GPUs and discuss the reasons for its poor performance in a naïve but common kernel code pattern in Section 2.3. Using the gained insights, we then apply zero-copy optimizations to graph traversal algorithms in Section 2.4. We discuss EMOGI's performance improvement for various graph traversal algorithms on several large graphs in Section 2.5 and we conclude in Section 2.7.

## 2.2 Background

In this section, we first provide a brief primer on GPU-based graph traversal. Then we will describe techniques that can be used to optimize the speed of graph traversal when graphs cannot fit into the GPU memory.

### 2.2.1 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 the case of BFS, only a single vertex needs to be set as active, which is basically a source vertex. The graph traversal can begin once all the initial active vertices are set. 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 the case of BFS, any neighboring vertices which have not been

**Algorithm 1** High-level Graph Traversal Flow

```
1:  set_initial_active_vertex()
2:  while there exist active vertices in G do
3:      for all vertices v1 in Graph G do
4:          if v1 is active then
5:              set v1 as inactive
6:              for all neighbors v2 of v1 do traverse_with_computation()
7:                  if application_dependent_condition() then
8:                      set v2 as active
9:                  end if
10:             end for
11:         end if
12:     end for
13: end while
```

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

The main benefit of the GPU implementation of the graph traversal comes from the massive number of vertices [15, 16, 13]. With the 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 [28, 29, 30, 31].

As an input graph format for the GPU graph traversal, we use the compressed sparse row (CSR) format. CSR is arguably the most popular way to represent a graph because of its low memory overhead [24, 25, 26, 4, 32, 33]. Certain graph processing frameworks such as nvGRAPH [34] 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 two arrays, as shown in Figure 2.1. 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.

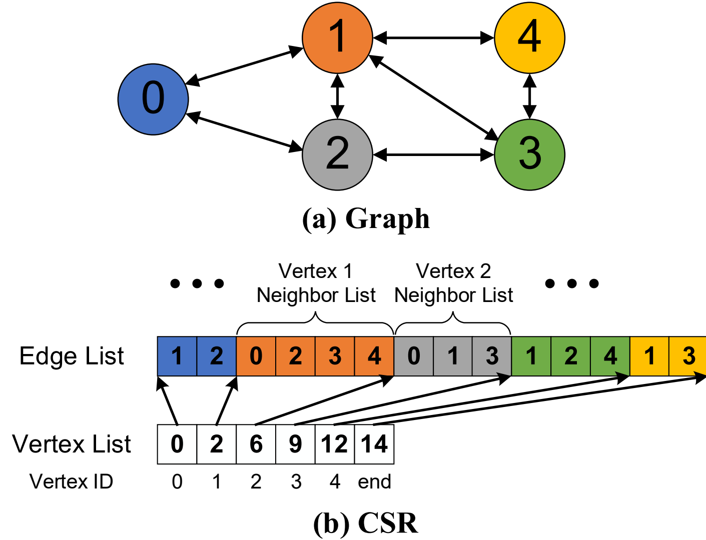


Figure 2.1: A sample undirected (a) graph and its (b) CSR representation. The edge list contains the neighbor list for each node. The vertex list is indexed by vertex IDs and contains the offsets for the starting position of that vertex's neighbor list in the edge list.

### 2.2.2 Out-of-Memory Graph Traversal on GPUs

Even in the CSR format, graphs 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) [17, 35, 36, 3, 18, 19, 20, 37, 21, 22, 23]. 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 manage where the data resides explicitly. UVM transparently allows device memory oversubscription 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 (4 kB). 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 kernel's memory footprint 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.

The inefficiency of UVM in graph traversal comes in two ways. First, it is hard for the very large graphs 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 2.1, 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 4 kB page migrations. Assuming that all accesses to the different neighbor lists will generate separate 4 kB page migrations, all neighbor lists should have at least 512 to 1024 elements (depending on the datatype size) to transfer the 4 kB data 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.

## 2.3 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 [38]. 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 were GPU global memory. The GPU translates 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 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 [5] and thus used only

for accessing small and frequently shared data. This section describes how to enable zero-copy and use a custom FPGA PCIe switch to explore any optimization opportunities available for zero-copy in detail. Based on the analysis, we propose several optimizations and show that correctly using zero-copy can nearly saturate the PCIe bandwidth and achieve much higher levels of application performance than previous approaches.

### 2.3.1 Enabling Zero-Copy

From the system's point of view, zero-copy is enabled as follows: First, the data to be shared with the 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. The 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`. The 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()` + `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 that all three techniques provided the same performance.

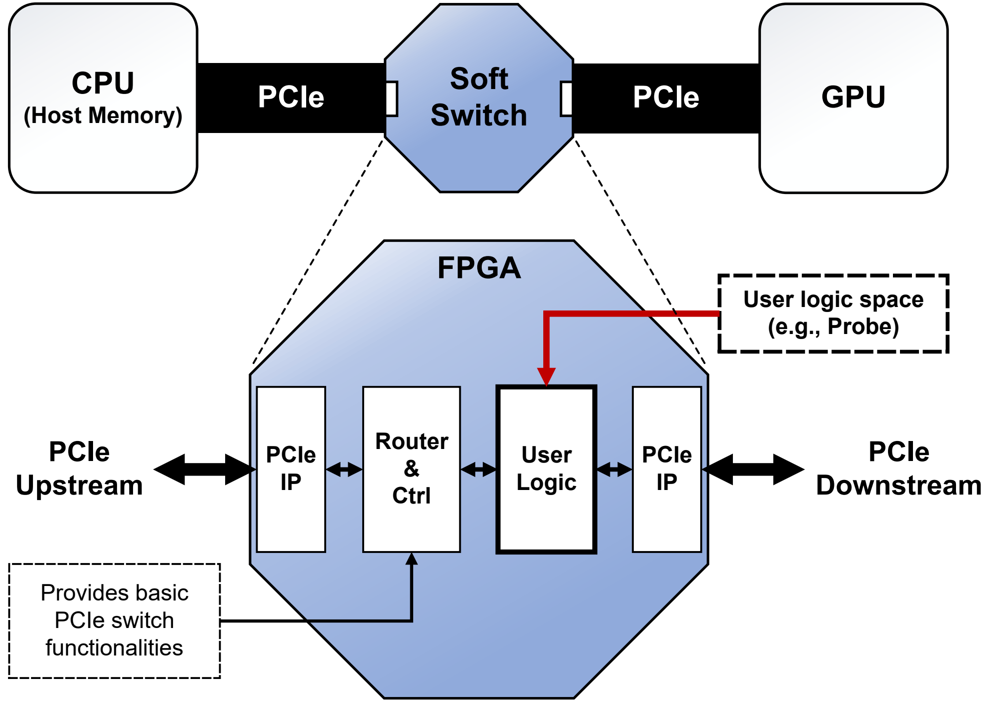


Figure 2.2: PCIe traffic monitoring environment. The FPGA is used to characterize the zero-copy memory access pattern from GPU.

### 2.3.2 Zero-Copy Analysis Setup

To understand how GPU accesses external zero-copy memory over PCIe, we design and build a monitoring system shown in Figure 2.2. The FPGA sits between the GPU and the CPU as a PCIe switch. The internal design of our FPGA switch is mainly divided into: (1) Upstream/Downstream PCIe IPs, (2) Router and control logic, and (3) User logic. The PCIe IPs are advertised as PCIe upstream/downstream switch ports to other PCIe devices. Therefore, functionality-wise, the FPGA switch is just a bridge between two PCIe devices, and it looks transparent to the user applications. The router and control logic handles miscellaneous PCIe-specific controls that the PCIe IPs do not deal with. For example, if an incoming PCIe configuration space request from the host is targeted for the downstream PCIe IP, the request should not be passed to the GPU but should be terminated at the downstream PCIe IP. In such a case, the router and control logic modifies the incoming packet appropriately so the request is served and the downstream PCIe IP returned to the host if it was a read request. The PCIe IPs provided by Xilinx do not include such a level of complexity; thus, we should explicitly

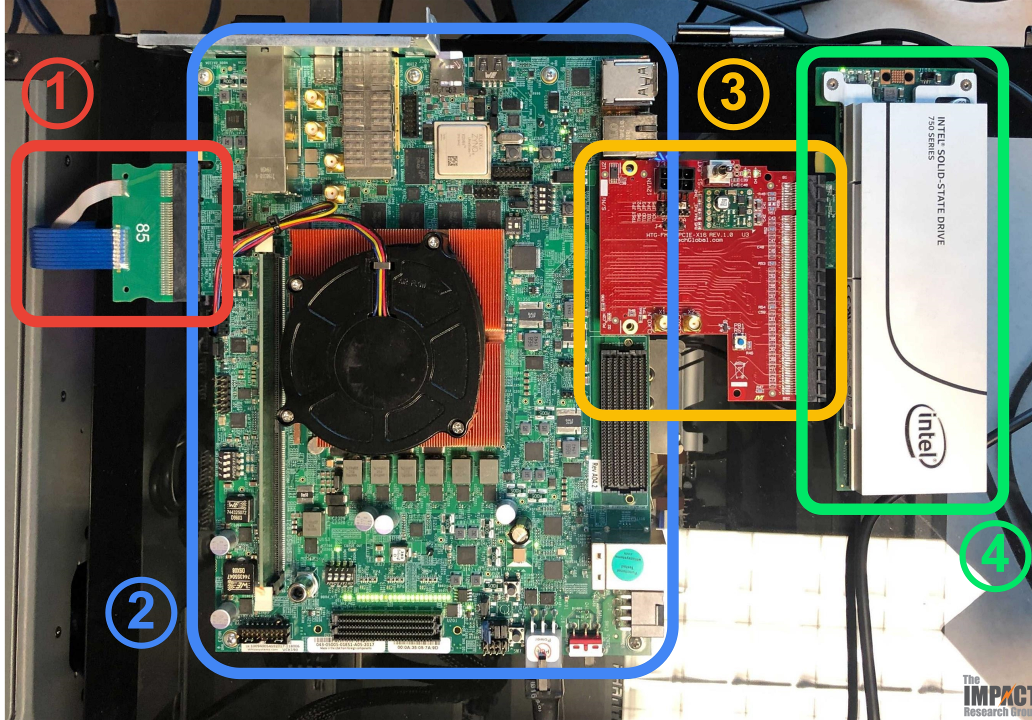


Figure 2.3: The FPGA switch setup in a real system. (1) Connection to a host system. (2) FPGA switch. (3) Connection to a test device. (4) A test device. In this example setup we are using an NVMe SSD as the test device.

implement those functionalities.

The FPGA can function as a standard PCIe switch with the two PCIe IPs and the router and control logic. However, we would like to utilize this design to monitor further the PCIe traffic between the host and the GPU, and therefore we add a custom traffic probe. The probe is connected to the external control server running Linux, and the probe provides user-interactive functionality so the users can initialize, trigger, and view the probe at will. In Figure 2.3, we show the real-world setup of our FPGA switch using an NVMe SSD device. From the user application side, we do not need any specific modification and the regular zero-copy memory allocation techniques can be reused.

### 2.3.3 Zero-Copy Mechanism and Optimization

Now that we can track zero-copy memory requests, we next need to understand the GPU access pattern to zero-copy memory. We create a toy example

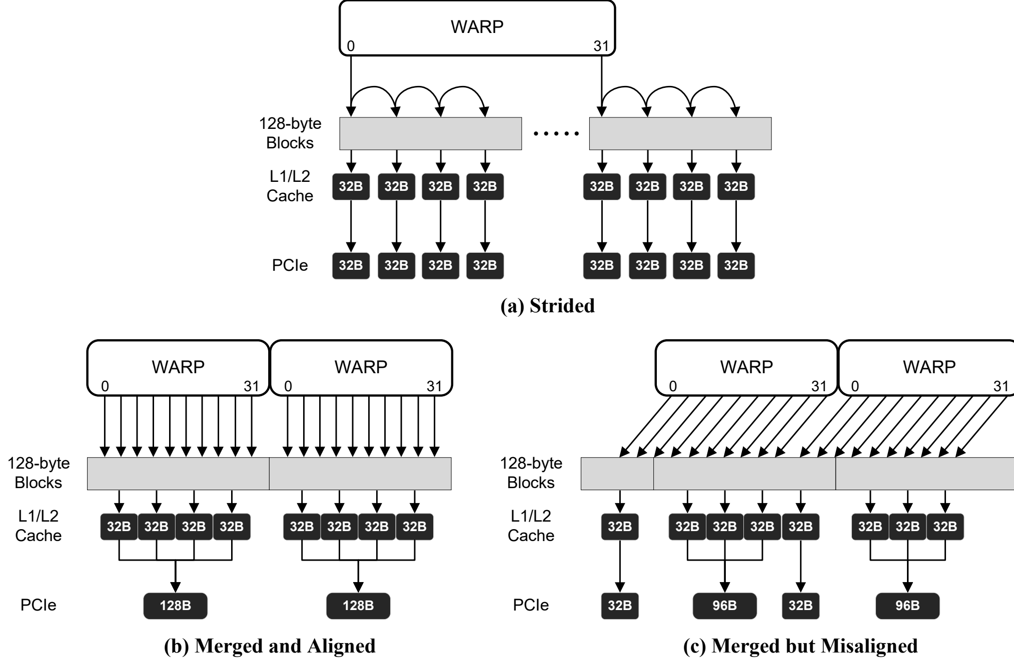


Figure 2.4: GPU PCIe memory request patterns observed with FPGA. In (a), each thread scans a different 128 B block and ends up making multiple 32 B PCIe memory read requests. In (b), individual 32 B memory read requests in a contiguous address space occur at the same time and GPU merges them into a single 128 B PCIe memory read request. In (c) each warp is performing a misaligned memory request (off by 32 B from 128 B boundary) resulting in generating a 32 B PCIe and 96 B PCIe request. In this figure, we assume each memory access is 4 B.

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, merge with misaligned access, or merge with aligned access. All PCIe traffic generated by these three variants is monitored using the FPGA monitoring platform and Intel VTune [39]. The PCIe layer in Figure 2.4 shows the GPU access patterns we observed with the FPGA monitoring platform while trying different CUDA kernels. We observe that the GPU can access the zero-copy memory in four 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 2.4 (a). With GPU threads iterating over their own

neighbor lists, we find that each thread generates a new 32-byte PCIe request every time they cross a 32-byte address boundary. Therefore, if the datatype 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 header overhead. Thus, fetching 32-byte of data makes the PCIe overhead ratio 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 roughly 1.0 $\mu$s to 1.6 $\mu$s. 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 [40]. In this case, the maximum bandwidth we can achieve with only 32-byte requests and 1.0 $\mu$s of RTT is merely 32 B / ( 1.0 $\mu$s / 256 ) = 7.63 GB/s. If we assume the PCIe RTT is always 1.6 $\mu$s, the bandwidth decreases to 4.77 GB/s. Third, the test system's minimum memory access size for DDR4 DRAM is 64-byte. Considering that a DDR4 2400MHz DRAM channel can provide 19.2 GB/s of sequential bandwidth, requesting only 32-byte read requests halves the effective per-channel DRAM bandwidth to 9.6 GB/s. Even if the overall DRAM bandwidth can be increased by adding more memory channels, this is still very wasteful. Finally, these 32-byte data items will likely occupy the GPU cachelines and can be evicted before all elements are traversed due to cache thrashing.

Figure 2.5 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 2.5 (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, using UVM results in transferring more bytes to the GPU compared to the original dataset size due to the frequent cacheline evictions. The key to addressing these limitations is to align and merge accesses. We will analyze the PCIe and DRAM bandwidth utilization with these optimiza-

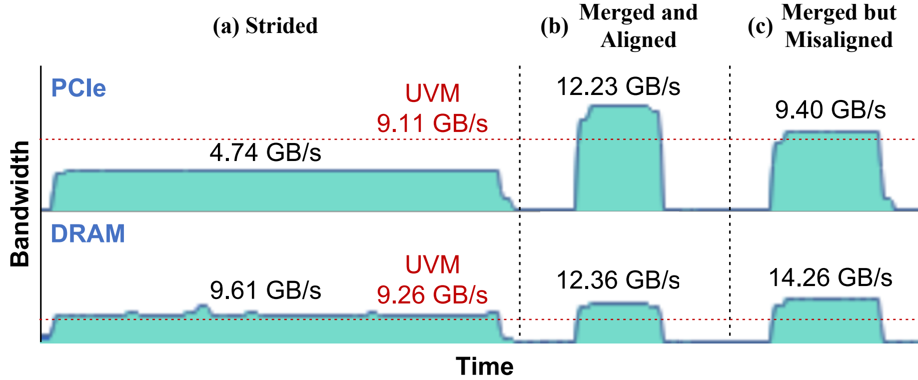


Figure 2.5: Average PCIe and DRAM bandwidth utilization for the different zero-copy access patterns, as reported by Intel VTune.

tions 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 2.4 (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 16 GB/s of bandwidth (without considering other PCIe overheads). Lastly, because 128-byte is a multiple of the DRAM request size (64-byte), there is no wasted bandwidth in the DRAM interface. In Figure 2.5 (b), we see this approach can saturate the PCIe bandwidth at about 12.23 GB/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 memory regions, the misalignment can be cascaded to all subsequent warps. Unfortunately, this results in all warps generating two PCIe requests. In Figure 2.4 (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 2.5 (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.

## 2.4 EMOGI: Zero-Copy Graph Traversal

Now that we understand zero-copy memory and its characteristics, we show 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 Section 2.4.1). Then, we introduce our baseline graph traversal algorithm (see Section 2.4.2) and optimize it for zero-copy memory based on the knowledge we gathered from Section 2.3.3 (see Section 2.4.3).

### 2.4.1 Data Locality in Graph Traversal

To exploit zero-copy for graph traversal, we prefer 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 datatype is 8-byte or 4-byte, respectively. Compared to UVM, which requires at least 4 kB of spatial locality (512 or 1024 elements of data), finding 16 to 32 elements of spatial locality is much easier for the graphs we studied.

Based on our analysis of 1,122 graphs from Network Repository [12], SuiteSparse Matrix Collection [10], and LAW [11], 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, with a standard deviation of 30. 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 efficiency level using UVM

since the available spatial locality is significantly less than the required 512 or 1024 elements.

### 2.4.2 EMOGI Baseline

EMOGI assumes the input graph is stored in the memory using the CSR data layout (see Section 2.2.1). All input data structures are statically mapped during initialization. The edge list is allocated in the host memory as it does not 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 Section 2.5.2), the vertex list consumes at most 1 GB of memory while the edge list can consume 38 GB. 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. In the vertex-centric graph traversal approach, the input graph is traversed by a single vertex depth on every kernel execution. Listing 2.1 shows the pseudo-code of our naïve baseline kernel 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 memory access pattern similar to that of the strided case explained in Section 2.3.3.

Compared with the UVM approach, EMOGI's graph traversal approach removes the page faults from occurring and reduces the I/O amplification as now we do not need to enforce the excessive amount of data movement.

### 2.4.3 Optimizations

Since the EMOGI baseline implementation is similar to the strided case presented in Section 2.3.3, 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 Section 2.3.3 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.

### 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 green comments in Listing 2.2. This allows EMOGI to always optimize for generating the maximum sized PCIe request to the zero-copy memory. Suppose the input graph fits in the GPU memory and the average degree of vertices in the graph is small. In that case, fine-tuning the worker size could 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 whose speed is about 1 to 10 percent of 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 adversely affect and decrease the effective bandwidth. Empirically, we observed that when the interconnect bandwidth is low, many threads are idle. Therefore, assigning a 32-thread warp to fetch data for even vertices with very few neighbors results in acceptable performance.

```
void strided(*edgeList, *offset, ...) {
    thread_id = get_thread_id();
    ...
    start = offset[thread_id];
    end = offset[thread_id + 1];

    // Each thread loops over a chunk of edge list}
    for (i = start; i < end; i++) {
        edgeDst = edgeList[i]; ...
    } ...
}
```

Listing 2.1: Uncoalesced Memory Access

```
#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];
            ...
        }
    } ...
}
```

Listing 2.2: Coalesced Memory Access (Merged + Aligned)

### Aligned Memory Access

As we discussed in Section 2.3.3, each misaligned access to the 1D data array can result in multiple smaller zero-copy requests. To address this, we have to merge memory accesses and align them as well. However, doing this on a CSR edge list is not straightforward, and this is because CSR does not 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 misalignment occurs. For instance, as shown in Listing 2.2 with blue comments, all starting indices fetched from the offset array are 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 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 not to miss any opportunity for generating large memory requests.

## 2.5 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 [41] and achieves better scaling compared to the UVM optimized implementation.

### 2.5.1 Experiment Setup

#### System Overview

We use a Cascade-lake server machine with two 20 core Intel Xeon Gold 6230 CPUs equipped with 256 GB of DDR4 2933 MHz memory and an NVIDIA Tesla SXM2 V100 16 GB GPU as our evaluation platform. The system is configured as shown in Figure 2.2. We use the FPGA only to analyze the zero-copy memory access pattern across different graphs. The detailed system specification is provided in Table 2.1. Graph edge lists are stored in the host memory while the vertex list and other temporary data structures are stored in the GPU memory.

#### Systems Compared

To show the performance benefit of EMOGI, we use four different graph traversal algorithms: Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank (PR) [42]. We base our initial implementation of BFS and SSSP from [43, 44], CC baseline implementation from [45], and PR baseline implementation from [46]. We compare EMOGI with the following systems:

**(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 UVM driver flags but did not observe notable differences **(b) Naïve** implementation is the baseline implementation of EMOGI using zero-copy memory and is identical to Algorithm 2.1. The vertex list is stored in the GPU memory in this implementation, 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 Section 2.4.3. However, there is no guarantee that accesses to the zero-copy memory are aligned in this implementation. **(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

Table 2.1: Evaluation system configuration.

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

there is a misalignment. This implementation is discussed in Section 2.4.3.

### 2.5.2 Evaluation Datasets

For the evaluation, we use the graphs listed in Table 2.2. GK, GU, FS, and ML are the largest four graphs from SuiteSparse Matrix Collection [10], and SK and UK5 are commonly used large graphs from LAW [11]. 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 randomly initializing weights for the rest of the graph from the integer values between 8 and 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.

### 2.5.3 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 2.2: Graph Datasets. $V$ = Vertex, $E$ = Edge, and $w$ = Weight.

| Abbreviation | Graph | Number | | Size (GB) | |
|---|---|---|---|---|---|
| | | $\|V\|$ | $\|E\|$ | $\|E\|$ | $\|w\|$ |
| GK | GAP-kron [14] | 134.2M | 4.22B | 31.5 | 15.7 |
| GU | GAP-urand [14] | 134.2M | 4.29B | 32.0 | 16.0 |
| FS | Friendster [13] | 65.6M | 3.61B | 26.9 | 13.5 |
| ML | MOLIERE_2016 [47] | 30.2M | 6.67B | 49.7 | 24.8 |
| SK | sk-2005 [15, 16, 11] | 50.6M | 1.95B | 14.5 | 7.3 |
| UK5 | uk-2007-05 [15, 16] | 105.9M | 3.74B | 27.8 | 13.9 |

Zero-copy Request Size Distribution:

In this evaluation, we show the impact of optimizing the memory access pattern from Section 2.3.3 on generating different sizes of PCIe request. The histogram of the PCIe request size is gathered using the FPGA monitoring platform explained in Section 2.3.2. In Figure 2.6, we show the breakdown of request sizes for all the PCIe requests from the three implementations: `Naïve`, `Merged`, and `Merged+Aligned`.

We observe in Figure 2.6 that nearly all PCIe requests in the case of `Naïve` implementation are of 32-byte granularity. This is because it is only possible to generate a PCIe request larger than 32-byte in the `Naïve` implementation when multiple neighbor lists happen to be spatially near in the edge list, and multiple threads in a single warp access them. 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.

We observe the following when we analyze the request size distribution for the `Merged` and`Merged + Aligned` optimized implementations. First, although the `Merged` approach's portion of 128-byte requests increases to about 40% on average, the portion 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 with most of their edges associated with high-degree vertices, we expect that most zero-copy memory requests should be 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 2.6. For example, the percent of 128-byte requests improves by 1.86× for the GK graph be-

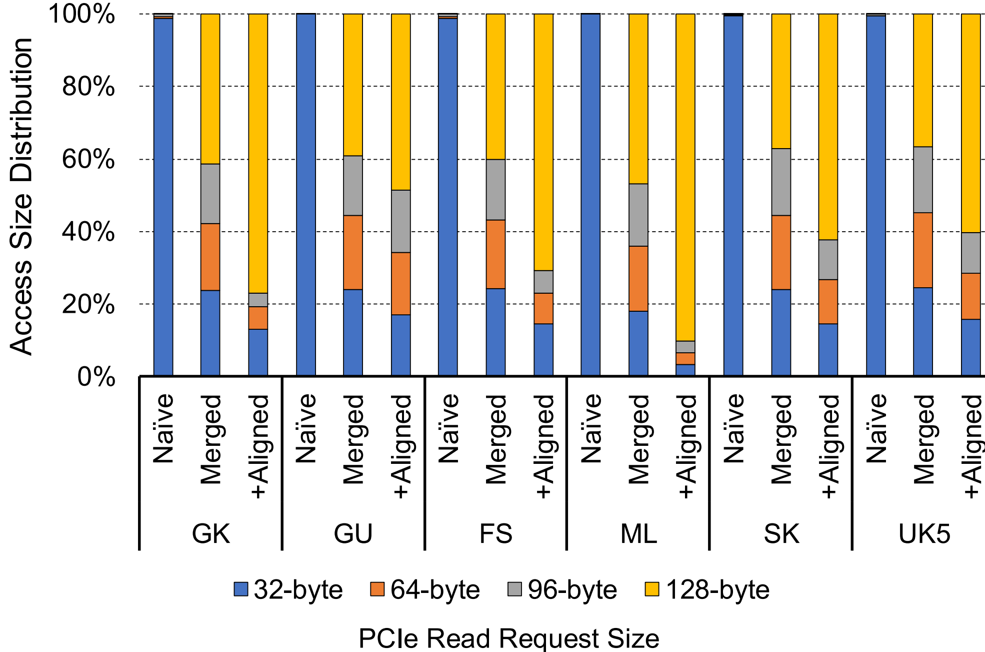


Figure 2.6: Distribution of PCIe read request sizes in BFS. `+Aligned` is an abbreviation for `Merged+Aligned`. As the merged and aligned optimizations are added, the BFS application generates more 128-byte requests for efficient access.

tween the `Merged` and `+Aligned` implementations. However, the percentage of 128-byte requests improves by only 1.25× between the two implementations on the GU graph, which has a similar number of edges and vertices as the GK graph.

To further analyze these behaviors, we plot in Figure 2.7, the cumulative distribution function (CDF) on the number of edges in each graph. CDF on the number of edges provides 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 2.7, we see that the ML graph has nearly no edges associated with small degree vertices. Thus, many requests can be merged to 128-bytes for the ML graph with the `Merge` optimization. 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 2.6, and, as a result, reduces the total number of zero-copy memory requests by 28.8%, as shown in Figure 2.8.

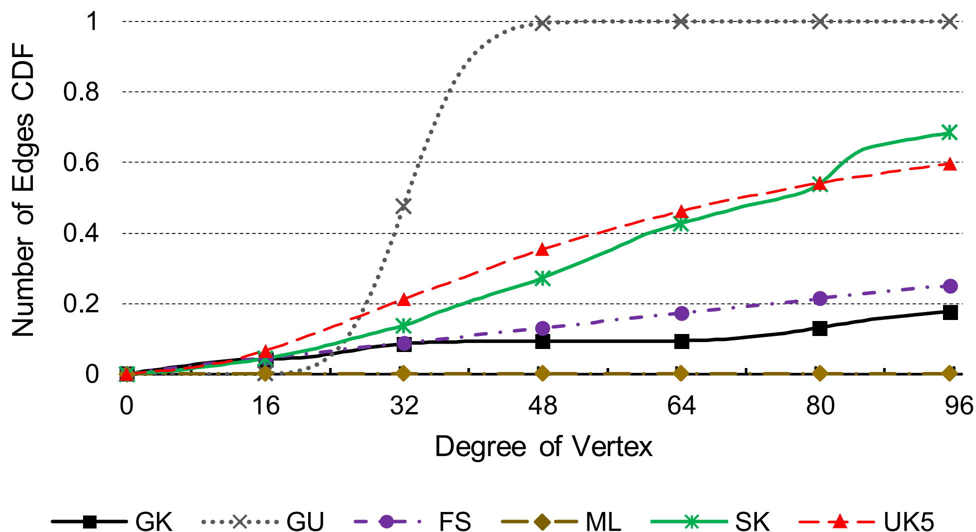


Figure 2.7: Number of edges CDF of evaluation graph. This plot provides a better understanding of the distribution of the neighbor list sizes in the graphs. For example, the GU graph has all of its edges associated with vertices with degree between 16 and 48, meaning the neighbor lists contain at most 48 neighbors.

To understand why the request size distribution of GK and GU graphs is significantly different for the `+Aligned` optimization, we need to understand these graphs' neighbor list size distributions. 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 datatype 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 high degree vertices can amortize the one-time misalignment cost over a large number of subsequent aligned accesses. 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.

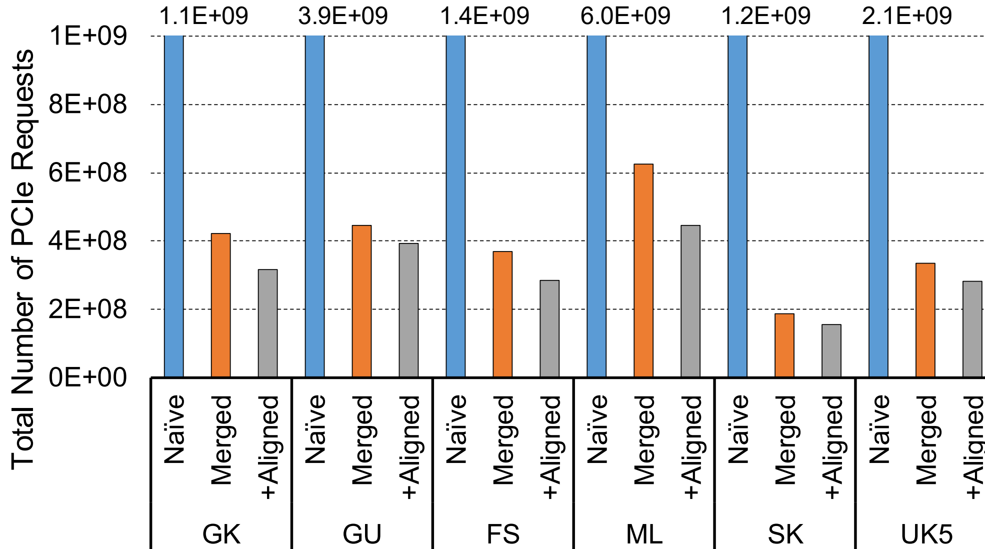


Figure 2.8: Number of PCIe requests sent for `Naïve`, `Merged` and `Merged+Aligned` implementations while executing BFS on various graphs. Collected from FPGA. `Merged` optimization reduces the PCIe memory requests by up to 83.3% compared to the `Naïve` implementation. `Merged+Aligned` optimization can further reduce the PCIe memory requests by up to 28.8%. `+Aligned` is an abbreviation for `Merged+Aligned`.

PCIe Bandwidth Analysis

The bandwidths we measured are more or less aligned with PCIe request size distributions. In Figure 2.9, we show the average achieved PCIe bandwidth while executing BFS. As a measure of merit, we use the maximal achievable PCIe bandwidth by `cudaMemcpy()` as the upper limit of what practical data access methods can hope to achieve. When transferring large data blocks, `cudaMemcpy()` incurs little overhead beyond the PCIe TLP headers. We measured the maximum achievable PCIe Gen3 bandwidth with `cudaMemcpy()` to be 12.3 GB/s, which is achieved when transferring data blocks of 16 megabytes and beyond. Because of the page faulting overhead present in the UVM, it can only achieve PCIe bandwidth of 9 GB/s.

EMOGI's `Naïve` implementation of BFS can only reach up to 4.7 GB/s PCIe bandwidth. This is consistent with what we observed using the toy example in Figure 2.5. With the `Merge` optimization, the PCIe bandwidth utilization increased up to 11 GB/s, reaching about 90% of the peak `cudaMemcpy()` bandwidth. With the `Merged+Aligned` optimization, we add about 0.5 to 1 GB/s of additional bandwidth utilization on top of `Merge` optimization in

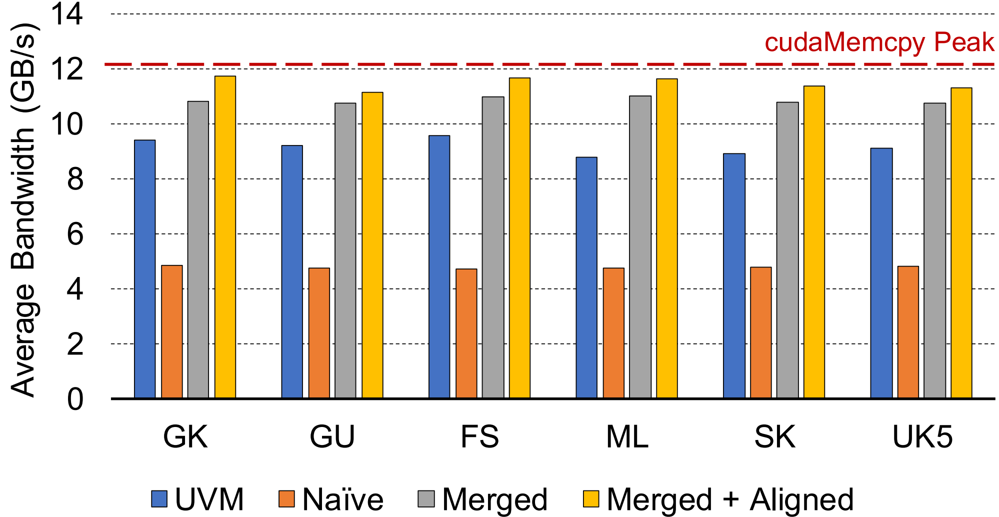


Figure 2.9: Average PCIe 3.0 x16 bandwidth utilization of the different implementations executing BFS.

all cases. The GU graph has the least improvement from the alignment optimization among all graphs. This is because BFS on the GU graph cannot send enough of 128-byte requests to saturate PCIe interconnect bandwidth. By comparing Figure 2.6 and Figure 2.9, we can clearly see the correlation between the distribution of PCIe request sizes and the achieved bandwidths in a real application, thus confirming our analysis in Section 2.3.3.

Analysis of Zero-copy Optimizations

We next evaluate the performance difference between `Naïve`, `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 Figure 2.10, the `Naïve` implementation's performance is 0.73× that of UVM on average. As discussed in Section 2.3.3, this is expected as the `Naïve` 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 `Merged` optimization's performance gain is only 1.21× over UVM. This is because the SK graph can almost fit in the 16 GB GPU memory. As a result, all the bytes trans-

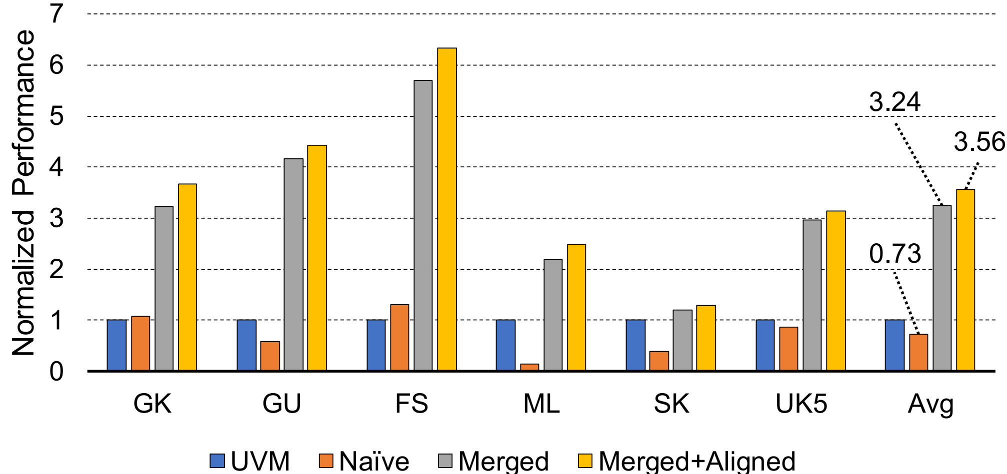


Figure 2.10: BFS performance of the `Naïve`, `Merged` and `Merged+Aligned` implementations against the UVM baseline.

ferred by page faults of the UVM approach are eventually used, minimizing the negative of the I/O amplification of the UVM approach. 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 2.8.

I/O Read Amplification

We now demonstrate the I/O read amplification benefit of EMOGI's fine-granular data accesses over the 4 kB page movement in UVM in BFS graph traversal. We chose the `Merge+Aligned` EMOGI implementation to represent EMOGI as it provides the best performance for this experiment. Figure 2.11 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 accessed during traversal are in different locations in memory, and thus there is very little spatial locality exploited for each 4 kB page moved. However, the two notable exceptions to this are the ML and SK graphs as UVM's I/O read amplification factors for them are

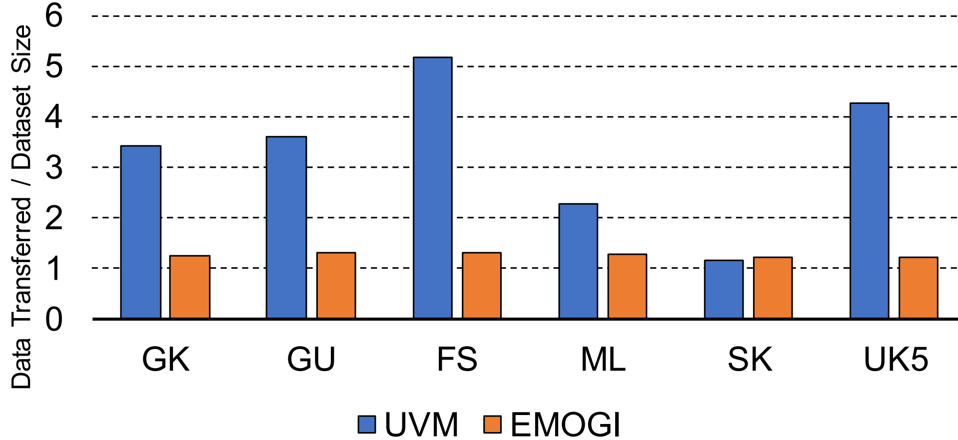


Figure 2.11: I/O read amplification of EMOGI and the UVM baseline while performing BFS.

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 does not exceed 1.31× because the fine-granular, merged, and aligned data access to zero-copy memory allows EMOGI to efficiently move only the necessary bytes over the slow PCIe interconnect.

### 2.5.4 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 2.12.

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, and EMOGI shows lower speedups over UVM than the other applications for CC and PR. In the case of SSSP and BFS, a specific vertex is selected as a root vertex and the applications start traversing the entire

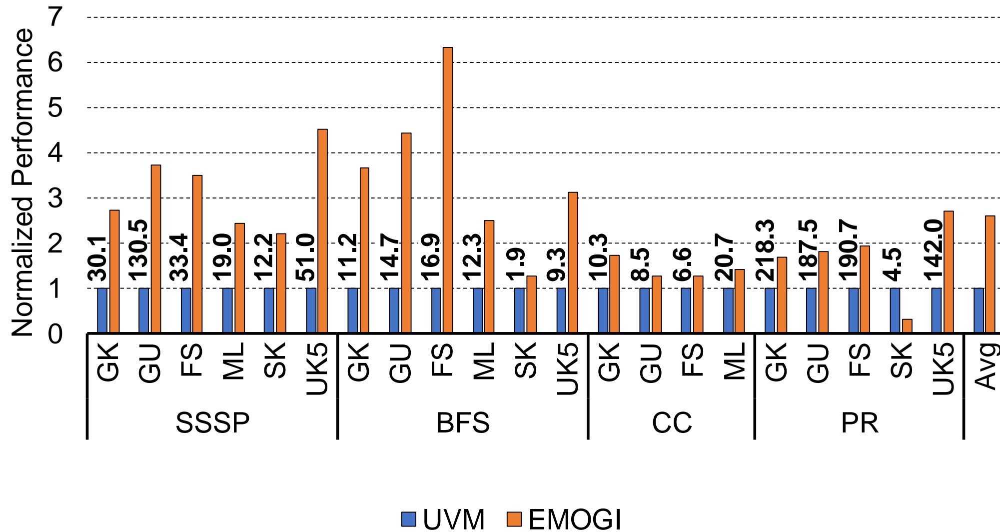


Figure 2.12: Performance comparison between UVM and EMOGI with different graph traversal applications with V100. Actual execution times of `UVM` cases are written on top of the bars (in seconds).

graph from the root vertex. However, with CC and PR, all vertices are set as root vertices instead of picking a specific vertex to start with, 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 2.13, we show the performance comparison of EMOGI when using a 4-byte edge list vs. an 8-byte edge list. On average, we observe about 1.57× of performance improvement when using 4-byte over 8-byte for EMOGI. In the case of GK and ML graphs in CC, the performance differences are nearly 2×. The performance differences in SSSP are smaller than the other applications since SSSP needs to transfer the weight values as well. Due to the higher computation to memory ratio in PR [48], PR also shows relatively smaller performance differences.

### 2.5.5 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 [3] and Subway [4]. Due to their varying runtime requirements, we also modify our EMOGI testing

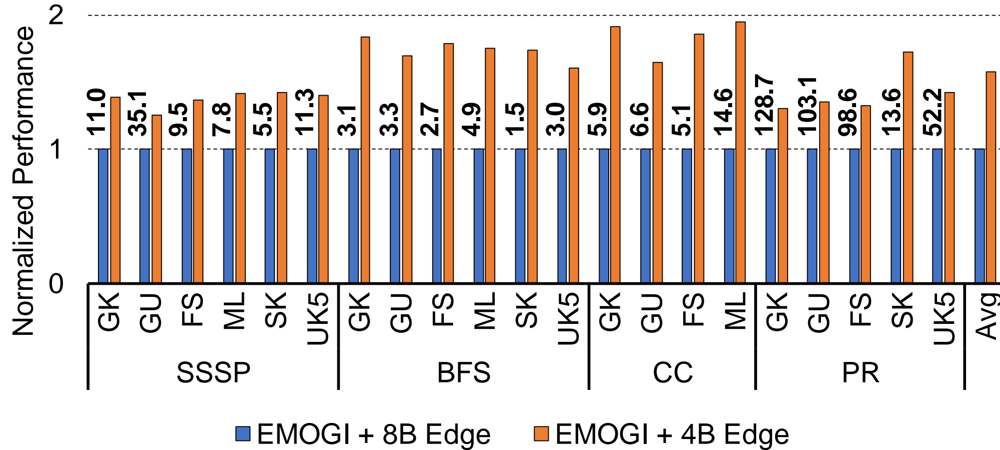


Figure 2.13: Performance comparison between using 4 B edge and 8 B edge for EMOGI with V100. Actual execution times of `EMOGI + 8B` cases are written on top of the bars (in seconds).

environment for accurate comparisons. The details of the modifications are described in the following sections.

### 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 Table 2.3. Overall, EMOGI shows 1.34× to 3.19× speedups against HALO.

### 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 small temporal 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 generate 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,

Table 2.3: Execution time comparison with HALO [3]. NVIDIA Titan Xp (12 GB) used.

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

Table 2.4: Execution time comparison with Subway [4]. NVIDIA Tesla V100 (16 GB) used. 4-byte edge used due to the Subway requirement.

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

BFS, and CC) with our platform described in Section 2.5.1. 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 Table 2.4. Overall, across all the graph datasets and graph traversal algorithms, EMOGI observes speedups of 1.57× to 4.73×.

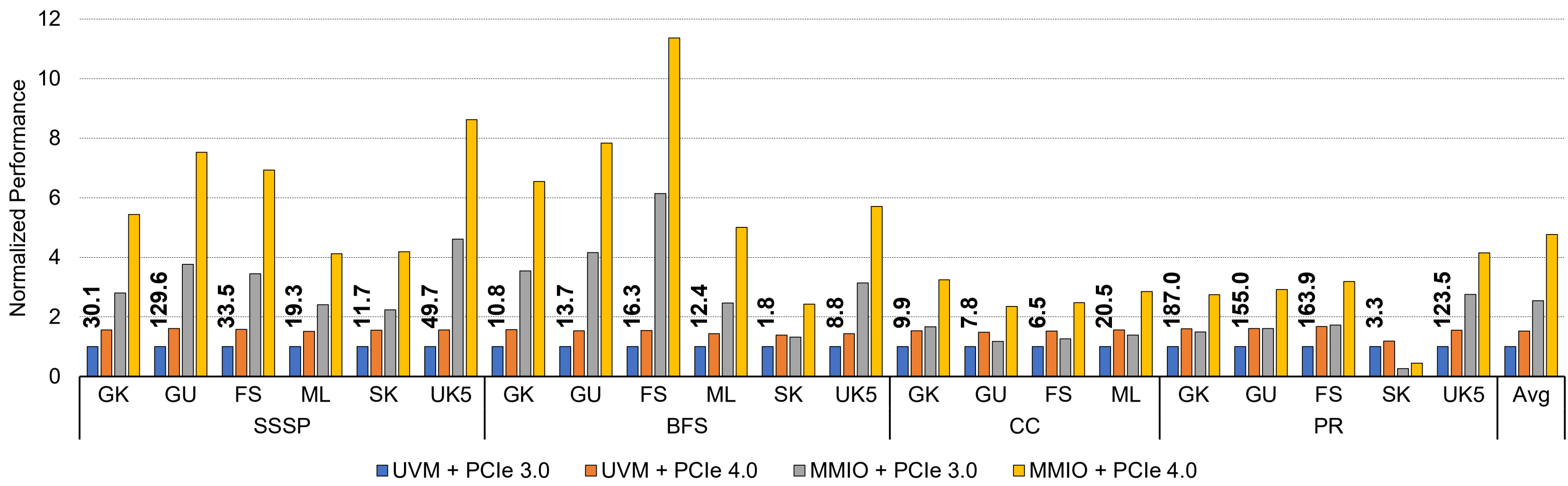


Figure 2.14: Performance comparison between UVM and EMOGI using PCIe 3.0 and PCIe 4.0. All results are measured in DGX A100. Actual execution times of `UVM + PCIe 3.0` cases are written on top of the bars (in seconds).

### 2.5.6 Performance Scaling with PCIe 4.0

As was shown in Section 2.5.3 and Section 2.5.3, EMOGI can nearly saturate the PCIe 3.0 bandwidth while outperforming 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 with `cudaMemcpy()`, approximately 24 GB/s, is twice as much as PCIe 3.0's peak measured bandwidth of approximately 12 GB/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 [49] with the A100 GPU and Dual AMD EPYC 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, and A100 memory is throttled to 16 GB.

The overall evaluation results comparing the performance of UVM and EMOGI on the DGX A100 system are shown in Figure 2.14. Here, we normalize the performance speedup 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 cannot keep up to use the higher bandwidth of the PCIe 4.0 interface. However, EMOGI does not 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.

## 2.6 Discussion

**Extending to other input formats:** In this work, EMOGI is targeting CSR, which is used by many popular graph processing frameworks [24, 46, 34, 43, 50], but the main idea of EMOGI can be extended to different formats as well. The most immediately applicable format is compressed sparse

column (CSC). The edge lists in CSR represent outgoing edges (push-based), but the edge lists in CSC represent incoming edges (pull-based). Although the directions are different between the CSR and CSC, the memory access pattern to the edge lists in both input formats is identical. Another interesting format expansion for EMOGI would be dynamic graphs [51] and compressed graphs [52]. 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 also be applied to these formats.

**Additional optimizations:** Several additional optimizations are available for EMOGI such as data compression and data caching in GPU global memory. Data compression on a graph [52, 53] can reduce the total amount of data transferred to the GPU, and we can obtain an effect similar to that of increasing the external interconnect bandwidth. As discussed in Section 2.4.3, 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 [54, 55] 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 there is currently no full hardware-based mechanism to naturally use the GPU global memory as a large cache. Therefore, a software-based caching mechanism needs to be implemented.

## 2.7 Conclusion

In this chapter, we presented EMOGI, a new method for optimizing the traversal of very large graphs with a GPU using zero-copy. We used a 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. Our experiments show that EMOGI outperforms the state-of-the-art solutions for traversing large 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 bottlenecked by the page fault handling overhead of traditional methods using UVM.

# CHAPTER 3

# FRAMEWORK INTEGRATION

## 3.1 Introduction

The use of fine-grained memory access over I/O may require very different flows of memory allocation, memory mapping, and function calls from the traditional block data transfer method. For example, enabling zero-copy access for NVIDIA GPUs requires specific types of CUDA APIs to be called in a particular order, and adding case-specific zero-copy access performance optimizations needs a capability to insert arbitrary code blocks flexibly at the framework level. While Chapter 2 has demonstrated that EMOGI can be used by an application kernel to achieve benefit over previous mechanisms, many modern applications are based on frameworks and application developers do not have the option to write/adapt data access kernels. This chapter shows how to adapt such a framework in the machine learning domain with the example of graph neural network (GNN) [56], and allow application developers to transparently benefit from the fine-grained memory access over I/O with simple adaptation.

The acceleration of modern machine learning models is often severely limited by insufficient memory bandwidth [57, 58, 59]. To provide the best possible memory bandwidth, data is usually placed in the memory closest to the processing units of the accelerators [60, 61]. However, with extremely large datasets, it is inevitable to put data farther from the processing units to take advantage of larger capacity (e.g., host memory). In this case, directly accessing remote data from the processing units can be very inefficient due to slow external interconnects. Thus, modern hardware systems utilize direct memory access (DMA) engines to free processing units from spending excessive time accessing remote data.

DMA engines are specialized to transfer large blocks of data independently.

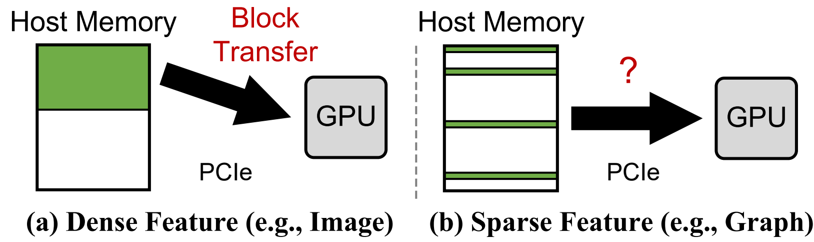


Figure 3.1: Challenge of GPUs accessing fine-grained sparse features in host memory.

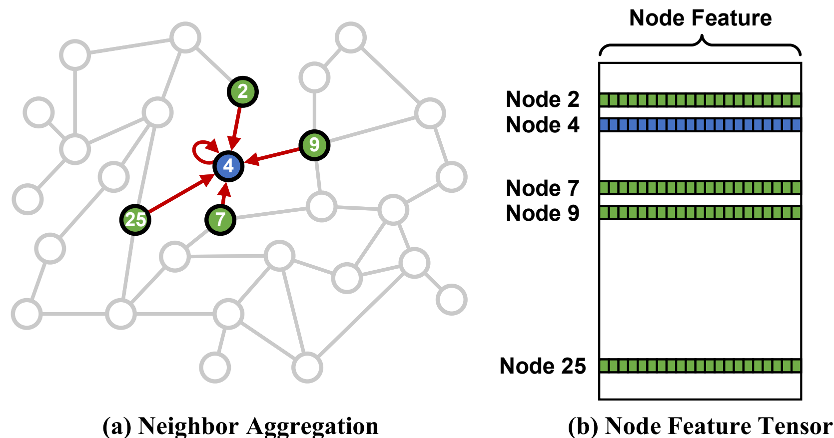


Figure 3.2: (a) A simple example of GNN training on single node. (b) An illustration of node features in memory. The neighboring nodes' features are scattered in memory.

By providing source and destination memory pointers along with the data size, DMA engines transfer data behind the scenes while keeping processing units available for other tasks. Initiating each DMA requires multiple interactions between the user application and the operating system, but these overheads can be offset by transferring large data blocks (Figure 3.1 (a)).

The recent adaptation of machine learning to a wide range of tasks has led modern deep neural networks to work on more complicated data structures such as graphs. Graphs are essential in representing real-world relational information in social networks and e-commerce. The capability to build high-quality recommender systems on graphs is indispensable to multiple businesses. In these graph data structures, the data which we need to access is often not coalesced together but scattered in memory (Figure 3.1 (b)).

One of the most successful deep neural network model adaptations to graph data is GNN. The core idea of GNN is to create node embeddings by itera-

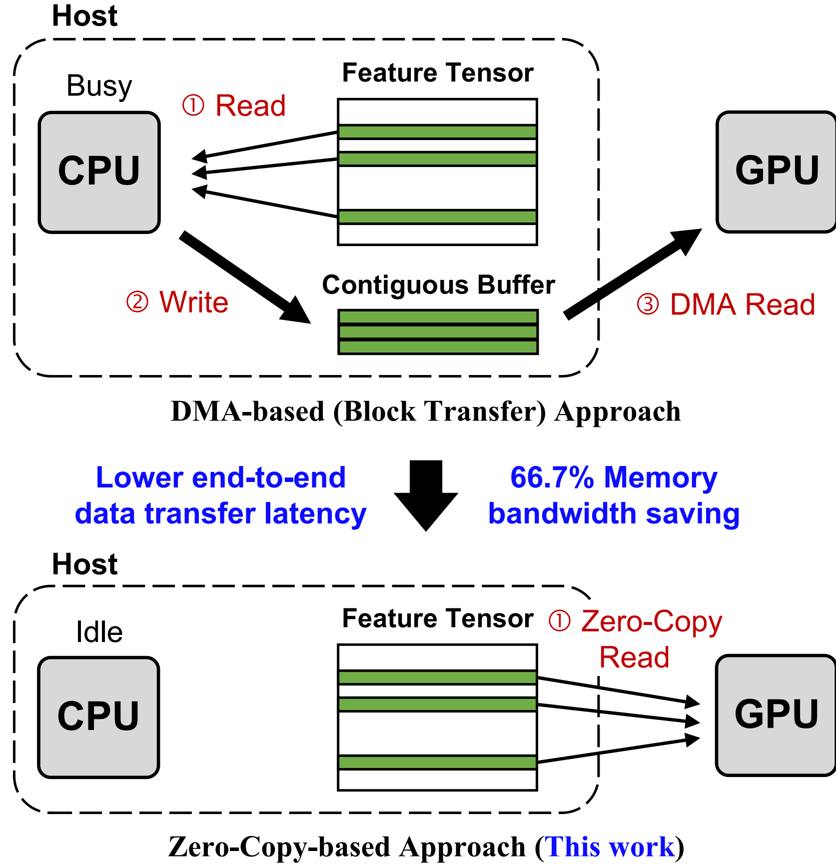


Figure 3.3: Workflow comparison between a DMA-based method and the proposed zero-copy-based method.

tively aggregating neighboring nodes' attributes using neural networks. Due to its neighboring node's attribute lookup, training GNN requires accessing multiple scattered locations in memory. In Figure 3.2 (a), we show a simple example of GNN training. To generate the embedding of node 4, we traverse the input graph and aggregate node 4's features alongside the features of all neighboring nodes in the node feature tensor. The example that we show here is only a toy example. In real-world graphs, each node can be connected to thousands of nodes. To collect relational information from those neighboring nodes, we may need to access thousands of scattered locations in memory. Without a doubt, such data access patterns make the traditional block data transfer method ineffective.

In this chapter, we propose a processor-oriented, software-defined data communication architecture. Instead of using DMA engines, we program GPU cores to directly access host memory with *zero-copy memory access*.

This approach allows the application developers to direct the GPU cores to exactly the locations that hold the data needed for computation. Conventional wisdom may still argue that since the node feature data is in host memory, CPU has a significant bandwidth advantage over GPUs, and therefore DMA should be a better option because CPU can quickly gather the sparse features on the fly. However, after the CPU gathers the sparse features, the gathered data still needs to be transferred to the GPU memory via the interconnect. Thus, the process is ultimately limited by the I/O interconnect bandwidth. In Chapter 2, we have shown that the ability to issue a massive number of concurrent memory accesses enables GPUs to tolerate latency effectively when accessing complicated data structures like graphs that reside in host memory [62]. Therefore, in GNN training, the proposed zero-copy access approach can offer a significant advantage over the DMA if GPUs can make targeted fine-grained host memory accesses for sparse features while fully utilizing the system interconnect (e.g., PCIe) bandwidth. Removing the CPU gathering stage shortens data access latency for GPUs and dramatically reduces the CPU and host memory utilization (Figure 3.3). Offloading CPU workloads to GPUs also helps with training GNN with multiple GPUs as we can prevent the CPU from becoming the bottleneck with an increasing number of workers.

In order to propose the GPU-oriented data communication architecture for GNN training, we address three major questions in this chapter. First, can zero-copy memory access fully utilize PCIe bandwidth while training GNN considering the long latency for accessing host memory? Second, what would be the price of consuming GPU cores for zero-copy memory access? Finally, after resolving the above two questions, can we show real end-to-end application performance benefit from our method?

In this chapter, we answer all three questions by proposing and applying the following optimization techniques to the GNN training framework DGL. First, we propose an automatic data access alignment optimization in the GPU data indexing kernel to maintain the best possible PCIe packet efficiency with zero-copy memory access. As we showed in Chapter 2, zero-copy accesses over PCIe can achieve up to 93% of block transfer PCIe bandwidth with our optimization. Second, we propose a novel CUDA multi-process service (MPS) [63] based resource provisioning optimization to minimize GPU resource consumption of zero-copy memory accesses. Based on careful inves-

tigation of PCIe protocol and GPU architecture, we conclude that we can saturate PCIe even by using only a few GPU cores to generate zero-copy accesses. Therefore, our optimization only requires a small portion of GPU resources for the zero-copy accesses and leaves the rest for computationally intensive workloads.

Finally, we build an end-to-end zero-copy GNN training flow in PyTorch. To enable zero-copy memory access, we devise a new class of tensor called "unified tensor". This tensor provides an address mapping of host memory for GPUs so they can directly access host memory with zero-copy accesses. By simply declaring multiple unified tensor instances for multiple GPUs, our GNN training flow can also support zero-copy access in a multi-GPU training environment. Our modifications are seamlessly integrated with the existing PyTorch framework, and therefore we can quickly apply our method on existing GNN training applications. We evaluate our design on multiple large graph datasets where the largest one has 111 million nodes and 1.6 billion edges. In a single-GPU training environment, our method is 16–44% faster than the DMA-based method, but our method becomes 65–92% faster than the DMA-based method in a multi-GPU training environment. Our method is efficient in hiding the remote sparse feature access time with the training time and can even match with the all-in-GPU-memory method for some graphs that fit in the GPU memory.

In summary, the main contributions of this chapter are as follows:

- As opposed to the traditional DMA-based data communication architecture, we propose GPU-oriented, software-defined data communication architecture with zero-copy memory accesses for efficient sparse accesses to graph node features in GNN training.

- To improve the efficiency of zero-copy memory access, we propose automatic data alignment and novel CUDA MPS-based resource provisioning optimizations.

- We seamlessly integrate our modifications with the existing PyTorch framework for easier programming and show 65–92% of end-to-end training performance gain.

The rest of the chapter is organized into the following sections. Section 3.2 provides the necessary background for the proposed approach. Section 3.3

gives a brief overview of the proposed approach. Section 3.4 presents an experimental evaluation of the proposed approach. Section 3.5 discusses potential future work. Section 3.6 offers concluding remarks.

## 3.2 Background

### 3.2.1 Graph Neural Network

The idea of graph neural networks (GNNs) [64, 65, 66, 67, 56, 68, 69] started by an attempt to apply filters similar to convolutional neural networks (CNNs) [70] on graph structures. Bruna et al. [64] were the first to propose the GNN model, where the authors utilized Laplacian filters as hidden layers to exploit the global structure of the graph. Such spectral construction was later adopted by many GNNs, including [56, 67].

GNNs are widely adopted in graph representation learning [71], where GNN is trained to produce high-quality embeddings of the given nodes. These embeddings can be used for performing several tasks such as link prediction and node classification. Traditional representation learning algorithms, including node2vec [72] and DeepWalk [73], are inherently shallow, transductive, and do not share parameters or utilize node attributes to encode nodes [71]. These characteristics limit the model's representation power and prevent it from inferring the representation when the nodes or edges are unseen in training. GNN opens up the potential to develop algorithms to tackle these problems [65, 66].

One severe issue with the early GNN is that the Laplacian filters in each layer are matrices whose dimension increases as the number of nodes in the graph increases. This effectively throttles the depth of GNN and the size of the graph it can be applied to, due to the large memory footprint. As an example, Kipf and Welling [56] present a model for semi-supervised node classification using GNN. The simplified form of its forward-propagation function can be written as:

$$H^{(l+1)} = f\left(H^{(l)}, A\right) = \text{softmax}\left(AH^{(l)}W^{(l)}\right)$$

where $A$ is an $N \times N$ adjacency matrix ($N$ is a number of nodes) representing

the node connectivity, $H$ is an embedding table, $W$ is a weight table, and $l$ is a layer number. Here, we can see the memory capacity requirement of the operation grows at least as fast as $N^2$, and the size of $N$ can go easily up to hundreds of millions in large-scale graphs.

### 3.2.2 Neighborhood Sampling

To tackle the limitation of GNN, GraphSAGE [65] introduces neighborhood sampling and aggregating approach. By sampling a fixed number of neighboring nodes instead of demanding the whole adjacency matrix, neighborhood sampling essentially reduces the computation and memory footprints and enables a fixed-size minibatching in both training and inference.

GraphSAGE models are a sequence of aggregation layers, which can be LSTM, pooling, or mean operations. The neighborhood sampling is applied to every neighboring node in every aggregation step. GraphSAGE uses a uniformly random selection process to sample the neighboring nodes, but other works such as FastGCN [74] and VR-GCN [75] use more complex algorithms to determine the neighboring nodes that need to be sampled. The commonly used hyperparameters for the neighborhood sampling size $S_{layer}$ are $(S_1, S_2) = (10, 25)$ (i.e., up to 10 samples from the immediate layer of neighboring nodes and up to 25 samples from the layer of nodes that are two layers away) and $(S_1, S_2, S_3) = (10, 10, 10)$. It is uncommon to go beyond the three layers of sampling due to the exponential growth in the number of nodes that need to be sampled. Such a lower depth of network layers than other deeper neural networks [76, 77] makes optimizing data transfer time-critical in GNN training. After the sampling, a sub-graph containing only the sampled nodes is created, so the computation kernel knows how to aggregate the node features of interest. Over different epochs of training, a new sampling is done to increase the learning entropy and cover more corner cases. The exact implementation of the sampling process is framework-dependent. In the case of deep graph library (DGL) [7], this part is written in C++ with OpenMP to maximize the performance, but PyTorch-Geometric [78] simply uses a Python code.

If the entire node feature table does not fit in the GPU memory, the sampled nodes' features must be transferred after each sampling step [66].

Since the sampled nodes' features are scattered over the feature table, the current GNN implementations in PyTorch or TensorFlow, the frameworks which use DMA as a data transfer method, require the features to be collected into a dense format prior to the data transfer.

### 3.2.3 CPU–GPU Data Communication

CUDA provides developers with three ways to transfer data between host and GPUs: (1) DMA APIs, (2) automatic page migration, and (3) zero-copy access [79].

As the first method, CUDA provides both synchronous and asynchronous APIs to copy data among host and devices. The two most commonly used APIs are `cudaMemcpy()` and `cudaMemcpyAsync()`. Both functions take a source pointer, a destination pointer, a data size, and a data transfer direction. For this method, the DMA engine is used for the data transfer. DMA engine is efficient at transferring a single large data block but suboptimal for transferring small-sized data due to the DMA request setup latency caused by user program $\Longleftrightarrow$ operating system interactions. According to Pearson et al. [80], to make the effective bandwidth of DMA to about 90% of the maximum PCIe 3.0 x16 bandwidth, the data block size should be at least 256 kB. With 64 kB of a data block transfer, the DMA efficiency drops to less than 50% of the maximum PCIe 3.0 x16 bandwidth.

Page migration is the second way. To provide convenience to programmers, NVIDIA introduced the Unified Virtual Memory (UVM) [35, 36, 41, 17, 80]. Data pointers to the memory regions managed by the UVM driver can be dereferenced by both GPU kernels and CPU functions. When a processor (either CPU or GPU) attempts to access a page that it does not own in its local memory, the accessed page needs to be migrated from a remote location. Similarly, if other processors access this page later, page migration to that processor will be triggered. The minimum migration granularity is identical to the system page size (4 kB), but it can be as large as 2 MB. UVM makes programming easier by removing the need for explicit calls of `cudaMemcpy()` by users. To allocate the UVM-backed memory region, programmers simply need to call `cudaMallocManaged()` with the desired size. However, the programmer-friendly page migration is not designed to be a performant data

transfer mechanism. Its performance is limited with irregular access patterns due to high page miss rate. This leads to an excessive amount of page faults that stall the execution and create I/O read amplification. With a larger discrepancy between the dataset size and the GPU memory size, more frequent page migrations will be incurred by severe page thrashing.

Finally, CUDA enables zero-copy access, which is also known as direct access. In zero-copy access, GPU sends a cacheline-sized memory request directly through an external interconnect (e.g., PCIe), without explicit data copy or page migration that will happen in the two methods mentioned above. The source memory region can be the host memory, peer PCIe devices, or other GPUs connected over NVLink. Zero-copy is useful in accessing fine-grained data, but it needs GPU cores to be engaged in generating memory requests.

## 3.3 Fine-Grained and GPU-Oriented Data Communication Architecture

Due to the widespread use of DMA-based data communication architecture, many system-level modifications must be established to support our GPU-oriented data communication architecture in the higher-level programming models. In this section, we first describe how we enable zero-copy accesses in PyTorch, and then we discuss some of the technical aspects of zero-copy access to identify its weaknesses and how to overcome them. Finally, we describe the end-to-end GNN training flow using zero-copy accesses.

### 3.3.1 Zero-Copy Enablement in PyTorch

For GNN training, we use PyTorch, one of the most popular Python-based ML frameworks. However, including PyTorch, there are no Python-based ML libraries which naturally support zero-copy access for GPUs. To overcome this issue, we create an extension of the existing PyTorch implementation with several modifications in its source code.

In PyTorch, data is allocated through a class called `"tensor"`. The physical location of data is determined by a context value which is passed to the class upon a declaration (Table 3.1). In the current implementation of PyTorch,

Table 3.1: PyTorch tensor class comparison.

| | Existing | | **This Work** |
|---|---|---|---|
| Context | CPU | CUDA | Unified |
| Worker | CPU | GPU | GPU |
| Data Storage | Host Memory | GPU Memory | Host Memory |

processing units can only compute data located within their own local memories. For example, a new tensor with CUDA context should be created to perform GPU-accelerated matrix multiplication on the CPU tensor. When the new CUDA tensor is created based on the old CPU tensor, PyTorch automatically calls DMA to copy the data in the host memory to the GPU memory.

In our design, we aim to aggressively avoid the implicit DMA data copy performed by PyTorch. We give GPUs direct access to tensor data in the host memory by mapping the host-memory data pointers into the GPU address space. To achieve our goal, we create a new class of tensor with a new "unified" context. A tensor with this new context can be declared from any existing CPU tensors. Upon the declaration, the tensor calls the `cudaHostReigster()` and `cudaHostGetDevicePointer()` CUDA APIs internally.

Calling `cudaHostReigster()` page-locks the given CPU tensor data and `cudaHostGetDevicePointer()` maps page-locked data into the GPU address space and returns a device pointer that can be used in GPU kernels for zero-copy accesses. There are several other ways of allocating a host memory space for zero-copy such as `cudaMallocHost()`, `cudaHostAlloc()`, or `cudaMallocManaged()` with `cudaMemAdvise()`, but these methods have some limitations for multi-GPU training, which we will explain in Section 3.3.4.

```python
import torch

#Input tensor data in host memory
input_tensor = torch.randn([100], device="cpu")

#CUDA tensor created, data copied by DMA (e.g., cudaMemcpy())
gpu_tensor = input_tensor.to(device="cuda")

#Unified tensor created, no data copy occurs
unified_tensor = input_tensor.to(device="unified")

#gpu_tensor data comes from GPU memory
#unified_tensor accessed through zero-copy access
#Computation done by GPU
output = gpu_tensor + unified_tensor
```

Listing 3.1: PyTorch Programming with CPU Tensor, GPU Tensor, and Unified Tensor

Besides the pointer manipulation, other existing PyTorch tensor mechanisms remain the same, and therefore there are no noticeable functional differences introduced to the end-users. Listing 3.1 shows a simple vector addition example in PyTorch using a unified tensor. From the code, we can see declaring the unified tensor is as simple as declaring the existing CUDA tensor. While the CUDA tensor is created by explicitly copying data from the CPU tensor, the unified tensor only creates a mapping to the host memory for the GPU. We have empirically measured that the GPU memory usage by the memory mapping is about 1/512 of the data size. Therefore, while the CUDA tensor will immediately fail on declaration if the data size is larger than the GPU memory capacity, the unified tensor can hold up to 512 times more.

### 3.3.2 Improving Zero-Copy Efficiency Over PCIe

One of the common misconceptions of zero-copy access is its low data transfer efficiency compared to the DMA-based methods [5]. The misconception is mainly coming from the fact that the users are treating the zero-copy without any specific care. However, as the zero-copy access requests are made over PCIe, it is important to understand how the zero-copy accesses interact with

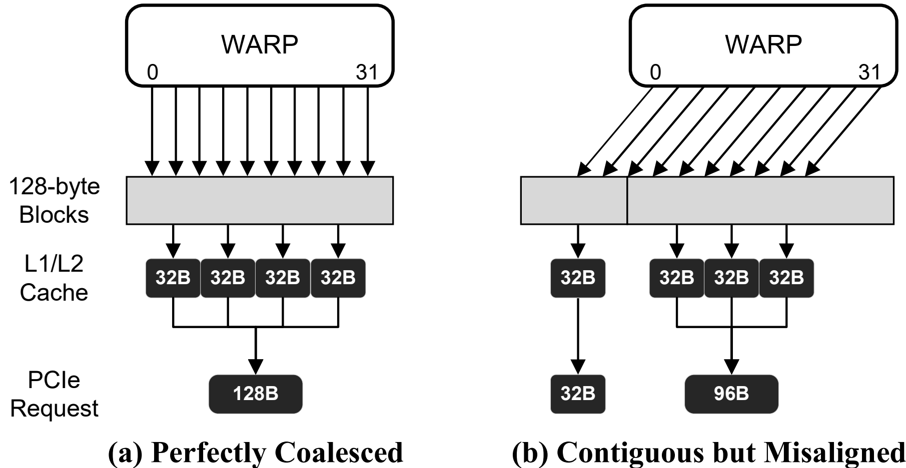


Figure 3.4: (a) A perfectly coalesced 128-byte access from a warp. (b) A warp accessing a misaligned data needs to generate multiple PCIe requests.

PCIe. In this section, we take a deep-dive into the technical aspect of PCIe protocol and its interaction with GPUs. We then present two important techniques for maximizing the zero-copy efficiency during GNN training.

Aligned Memory Access

Even though our purpose of using zero-copy is to make fine-grained memory accesses to the host memory, it is still desirable to make coarser-grained PCIe memory requests whenever possible for a couple of reasons. First, each PCIe packet has 12–16 bytes of header overhead. Therefore, it is better to increase the payload size by requesting a larger memory request to compensate for the overhead. Second, PCIe devices have a hard limit on the number of outstanding requests they can create. Since the PCIe round trip time (RTT) is very long (1–5us, variable), submitting multiple read requests in a pipelined fashion is necessary to occupy the interconnect fully. However, if we squander the capacity of the allowed number of outstanding requests by generating too many small read requests, it becomes difficult to tolerate the latency and utilize the PCIe bandwidth fully. The numbers of maximum outstanding read requests for PCIe 3.0 and PCIe 4.0 are 256 and 768, respectively.

Now, how do we generate coarser-grained PCIe requests with all that in mind? According to Min et al. [62] and also presented in Chapter 2, to make PCIe read requests more efficient, the same technique used for the

GPU memory coalescing [81] can be used. Figure 3.4 illustrates two cases where (a) memory accesses from a warp are contiguous and aligned with the GPU cacheline, and (b) memory accesses from a warp are contiguous but misaligned with the GPU cacheline. In the case of (a), the accesses from the threads in a warp are perfectly coalesced, and the coalesced requests become a single 128 B PCIe read request. In the case of (b), the accesses from a warp are scattered over two GPU cachelines, and they result in generating two separate PCIe read requests. The possible memory access granularities are 32 B, 64 B, 96 B, and 128 B, while 32 B is a single sector size of GPU cacheline [82]. Each GPU cacheline is composed of four sectors.

```
#define WARP_SIZE 32
__global__ void index(float* dst, float* src,
                      int* idx_list, int feat_size,
                      int numElem) {
  int linearIdx = blockDim.x * blockIdx.x
                  + threadIdx.x;

  for (int i = linearIdx; i < numElem;
       i += blockDim.x * gridDim.x) {
    int dstIdx = i / feat_size;
    int offset = i % feat_size;

    // src: host memory, dst: GPU memory
    int dstStart = dstIdx * feat_size;
    int srcStart = idx_list[dstIdx] * feat_size;

    int dstOffset = offset + dstStart;
    int srcOffset = offset + srcStart;

    // Cacheline-size-aware circular shift stage added
    if (feat_size > WARP_SIZE && feat_size % WARP_SIZE) {
      int diff = (dstStart - srcStart) % WARP_SIZE;
      diff = diff < 0 ?  diff + WARP_SIZE : diff;

      dstOffset += diff;
      srcOffset += diff;

      if (srcOffset >= srcStart + feat_size) {
        dstOffset -= feat_size;
        srcOffset -= feat_size;
      }
    }

    dst[dstOffset] = src[srcOffset];
  }
}
```

Listing 3.2: GPU Indexing Kernel and Automatic Alignment

Of course, we would not need to worry about the misaligned accesses if the node feature objects always start at 128 B boundaries and the sizes of node features are always multiples of 128 B, but it is very unlikely to be so in reality. For example, if a certain dataset's node feature size is 480 B,

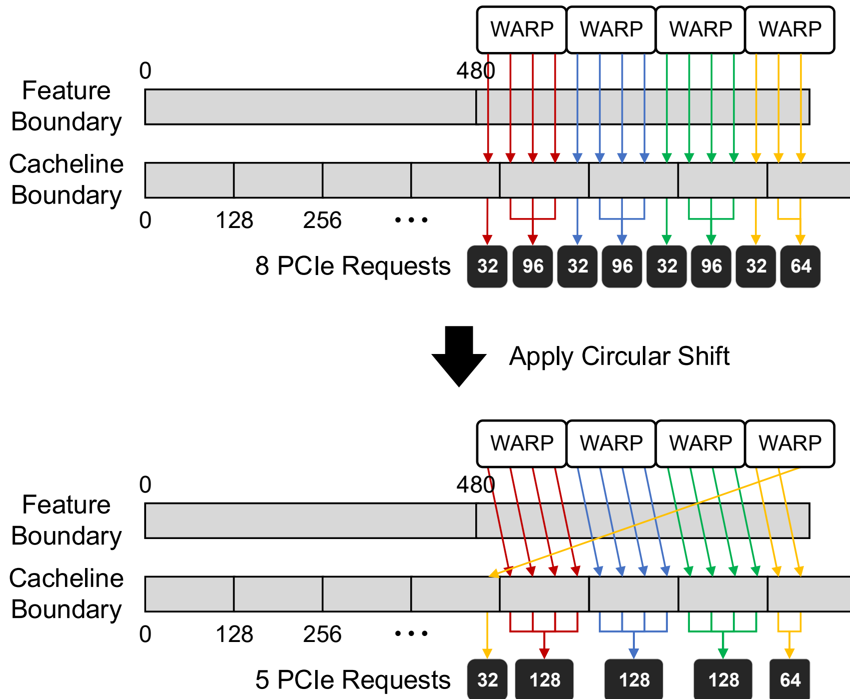


Figure 3.5: Circular shift optimization explained. Circular shift transforms memory requests into a GPU cacheline-friendly way.

accessing the second node feature will start from accessing the 480th byte in a memory address. In this case, we are off by 32 B from the closest 128 B boundary (512 B). To automatically resolve this issue, we add a circular shift stage in the PyTorch indexing CUDA kernel. In Listing 3.2, we show the circular shift stage code we added, but in a simplified manner. The shifting stage is aware of the GPU cacheline size and shifts the memory access indices by calculating the offset between the nearest 128 B aligned location and the current indexing location. The visualization of our circular shift mechanism is shown in Figure 3.5. In this example, we want to access the second node feature with zero-copy access, where each node feature size is 480 B. Without the optimization, each warp starts reading from misaligned locations and ends up generating 8 PCIe requests. However, once our optimization is applied, the warps adjust their indexing locations and try to generate aligned memory accesses as much as possible. In this example, the total number of PCIe read requests is reduced to 5.

We do not apply the circular shift stage if the node feature size is less than the GPU cacheline size or if it is already a multiple of the GPU cacheline

size. All these adjustments are transparent to the high-level programmers due to our modifications to PyTorch source code.

### Asynchronous Operations and Resource Provisioning

One important distinction of our design is that zero-copy accesses are done by GPU kernels. In other words, the other following GPU kernels need to wait until the zero-copy kernel is finished, even if it is simply reading the host memory. However, like in many other ML algorithms, GNN can also greatly benefit from overlapping data communication time and training time, which naturally happens in DMA-based methods. To achieve the best training performance, we must devise a way to overlap our design's training GPU kernels and zero-copy GPU kernels.

Normally, concurrency and overlapping activities can be accomplished by using CUDA streams. CUDA streams allow GPU kernels and API service activities in different streams to execute in arbitrary order so as to enable overlapped operations. Unfortunately, there are several situations where achieving concurrency is impossible. First, there are several blocking CUDA APIs such as `cudaMalloc()`, `cudaFree()`, and `cudaEventQuery()` that serialize the GPU operations. In the current implementation of PyTorch, some of the listed APIs are called in the background implicitly, such as by the memory allocation manager. If one of the CUDA APIs is called in between the zero-copy GPU kernel and the training GPU kernel, the latter GPU kernel must wait until the earlier GPU kernel's operation is finished. Second, when a current GPU kernel completely consumes the GPU resources, the following kernel must wait until the resources are released. In general, most of the GPU kernels try to occupy as much as of GPU resources as they can, and the serialization situation is very likely to occur.

However, in fact, we have missed a fundamental question here. Before we think about concurrency, how much GPU resource do we need for the zero-copy GPU kernels? If we need the entire GPU resource to utilize the PCIe bandwidth fully, then there is no point in attempting to achieve concurrency in the first place. This is the core question that needs to be answered to verify the validity of the idea of overlapping zero-copy and training GPU kernels.

To answer to this question, we explore the architecture details of NVIDIA

Table 3.2: NVIDIA RTX 3090 specifications.

| Category | Specification |
|---|---|
| PCIe Generation | 4.0 |
| Max # of Outstanding PCIe 4.0 Read Requests | 768 |
| # of Multiprocessors | 82 |
| # of Threads per Multiprocessor | 1536 |
| # of Threads per Warp | 32 |

GPUs. In NVIDIA GPUs, to better utilize computation units and to hide long GPU memory access latency, every single physical core may have multiple active warps to issue instructions from [83]. In this way, the physical core will not be stalled when some of the warps are waiting for the completion of their memory requests. Therefore, the number of physical GPU cores we need to reserve is much smaller than the number of memory requests we want to generate.

As we discussed in Section 3.3.2, there is a hard limit on the number of outstanding PCIe read requests that PCIe devices can generate at a given moment. Therefore, if we can prove that we only need a small amount of GPU resources to saturate the limit fully, it is worthwhile to seek a way to achieve concurrency. In Table 3.2, we list the specifications of NVIDIA RTX 3090 GPU which we use for our evaluations. At any given moment, the PCIe interface would not allow the GPU to generate more than 768 outstanding PCIe read requests. To identify the portion of GPU resource we need to generate 768 outstanding PCIe read requests, we perform the following calculation. First, assume each warp's memory requests are coalesced to a single PCIe read request, and ignore the payload size for now. In this case, we need 768 warps available to the scheduler to reach the PCIe 4.0 limit. Since each streaming multiprocessor (physical processor) can hold up to 1,536 threads at a given moment, each multiprocessor can sustain up to 1,536 / 32 = 48 outstanding PCIe read requests. Now, we have 82 multiprocessors in RTX 3090, so the amount of GPU resource that we need to reserve for the zero-copy GPU kernel is about 16 / 82 = 19.5%. However, this is the upper bound for the extreme case. If we assume we can always generate 128B PCIe read requests, we can saturate the PCIe 4.0 bandwidth

with far fewer outstanding requests. For example, the measured maximum PCIe bandwidth with `cudaMemcpy()` in RTX 3090 is 25.8 GB/s and if we assume RTT (Round-Trip-Time) of PCIe is 1.5 $\mu$s [84], the number of outstanding requests that we need to sustain is (25.8 GB/s) / (128 B) $\times$ 1.5 $\mu$s = 324.6. That is, assuming all PCIe requests are 128 B in size, we need to reserve only 8.2% of the total GPU resource for the zero-copy GPU kernel. In reality, since some of the requests will be smaller, this number is a lower bound and the actual number will be somewhat higher. In short, even if we try to maximize the zero-copy GPU kernel efficiency, there are at least 80% and up to 91.8% of the GPU resources available for other workloads.

Now, finally, since we realize how much GPU resource should be allocated for the zero-copy kernel, we explore the method to enforce the limitation in practice. Fortunately, NVIDIA GPUs already provide support for limited execution resource provisioning through CUDA multiprocessing service (MPS) [63]. MPS is originally designed to improve the quality of service (QoS) between different clients' workloads, but we utilize this service to control the resource utilization of the zero-copy GPU kernel. To assign different resource limitations to different kernels, the kernels must be running in different contexts/processes. Since PyTorch already supports a multiprocessing programming model, it is simple to launch the zero-copy GPU kernel and the training GPU kernel in two separate processes.

To manually partition the GPU resource, we use the `cuCtxCreate_v3()` API from CUDA. With this API, we can control the amount of GPU resources that we want to reserve at the user-programming level. For example, if we want to only use two SMs of GPU for a given context/process, we can set the `CUexecAffinitySmCount` argument to 2 and pass it to the `cuCtxCreate_v3()` API. Now, later in the program, if we want to launch a CUDA kernel with only up to two SMs, we just need to switch to the context that we created with the API. Another side benefit of the multiprocessing approach is that the different GPU kernels running in different processes are not affected by the other processes' blocking CUDA API calls. With our approach, zero-copy accesses can take full advantage of the PCIe bandwidth while leaving the majority of GPU resources available for other computationally intensive workloads.

Figure 3.6 summarizes the benefit of our approach. When the zero-copy kernel is not throttled, it blocks the follow-up work (Figure 3.6 (a)), while the

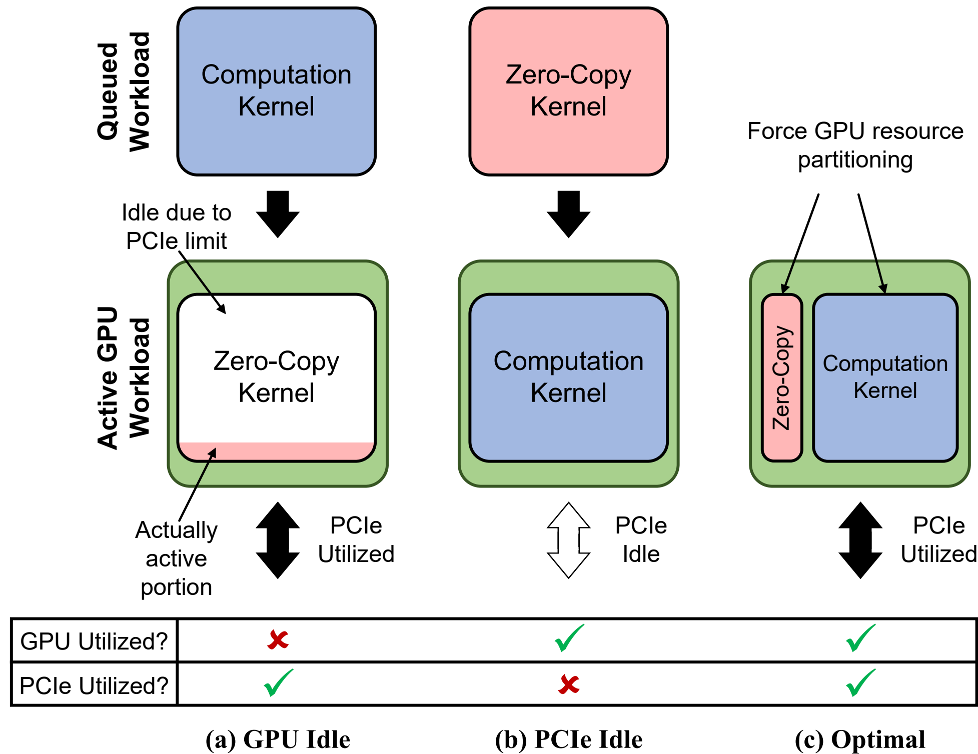


Figure 3.6: GPU and PCIe utilization comparison on different scenarios of workload executions and GPU resource partitioning.

GPU resource is mostly idling. When the other kernels (e.g., computation) are not throttled, the GPU can focus on the computation, but the PCIe is not utilized at all (Figure 3.6 (b)). When we enable the GPU resource partitioning, now we can properly utilize both the GPU and the PCIe at the same time (Figure 3.6 (c)).

With this optimization, we can transform the GPU cores into an intelligent DMA engine that can asynchronously perform complex data accesses such as data dependent index calculations and fine-grained host memory accesses. This optimization can also be useful for some workloads that utilize peer-to-peer GPU memory accesses with zero-copy accesses.

### 3.3.3 Workload Scheduling

In this section, we combine all the implementation details we discussed in the previous sections, and explain the overall flow of our GNN training with zero-copy accesses enabled. In Figure 3.7 (a), we show the initial tensor

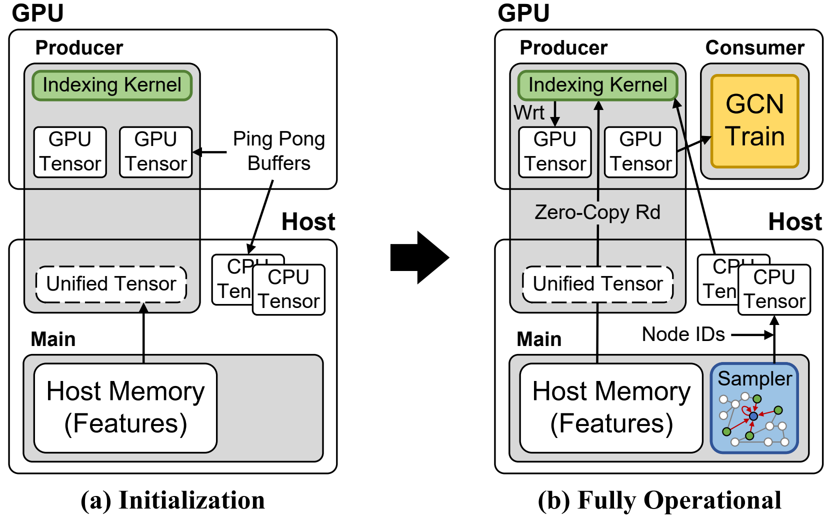


Figure 3.7: GNN training flow with zero-copy accesses. Only the operations related to data accesses are shown. (a) We set up unified tensor and the returned pointer is passed to GPU for zero-copy accesses. (b) The sampler generates node IDs used by the producer and the producer gathers scattered node features in the host memory. The consumer uses the gathered node features for training.

allocations during the initialization step. First, we map the whole node feature tensor into the GPU address space using the unified tensor. This unified tensor holds a memory pointer that the GPU can use in its kernel to generate zero-copy accesses to the node feature tensor. Next, we create two sets of ping pong buffers for interprocess communications. The goal of using ping pong buffers is to remove the usage of locking mechanisms between two different processes sharing data and allow them to start working for the next minibatch immediately after finishing their current works. In our design, each process needs to be synchronized just once per minibatch.

After the initialization, the training pipeline begins from the sampler process, randomly selecting nodes and collecting their neighbors' node indices (Figure 3.7 (b)). Once all the node indices are identified, the combined list is transferred to the producer process running on GPU for the zero-copy accesses. The list of node indices is transferred over DMA as it is contiguous and small. Once the node features are all gathered into one of the ping pong buffers, the producer notifies the consumer to train on the new minibatch data as soon as it is ready. Since the GPU ping pong buffers are located in the same GPU memory, it naturally makes sense for the consumer to di-

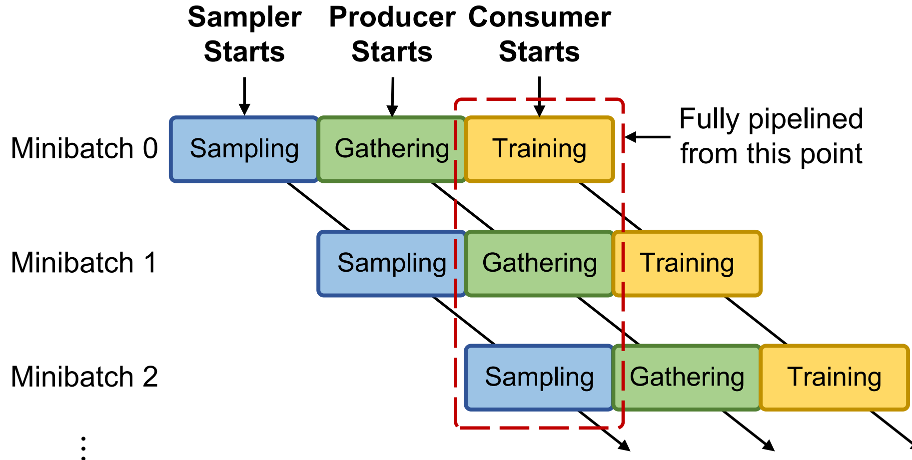


Figure 3.8: A visualization of GNN processing pipeline.

rectly access the buffer owned by the producer instead of copying it to its own space. To achieve this, we utilize CUDA interprocess communication (IPC) APIs. With the CUDA IPC APIs, two different GPU kernels running on different processes can share the same GPU memory space without data copies in between. This specific GPU pointer sharing procedure is implemented in the PyTorch `Queue` class and we utilize it for our application. The ping pong buffers are statically located for the entire training process, and therefore the pointer sharing needs to be done only once at the beginning of the producer process.

From the user's point of view, the training process is pipelined in a sampler $\rightarrow$ producer $\rightarrow$ consumer order (Figure 3.8). Except for the unified tensor declaration, all of our end-to-end GNN training implementations are developed with the existing PyTorch functionalities, making our method more accessible for the existing users. Another benefit of using PyTorch is the access to multiple fault-tolerant mechanisms in PyTorch, such as checkpointing and TorchElastic [85] framework, which allow users to recover from failure or to train even with faulty hardware steadily. Our implementation does not alter those mechanisms, and they can be used at the same time. Our modifications are isolated into the data transfer portion of the GNN training, and the training algorithms are unaffected.

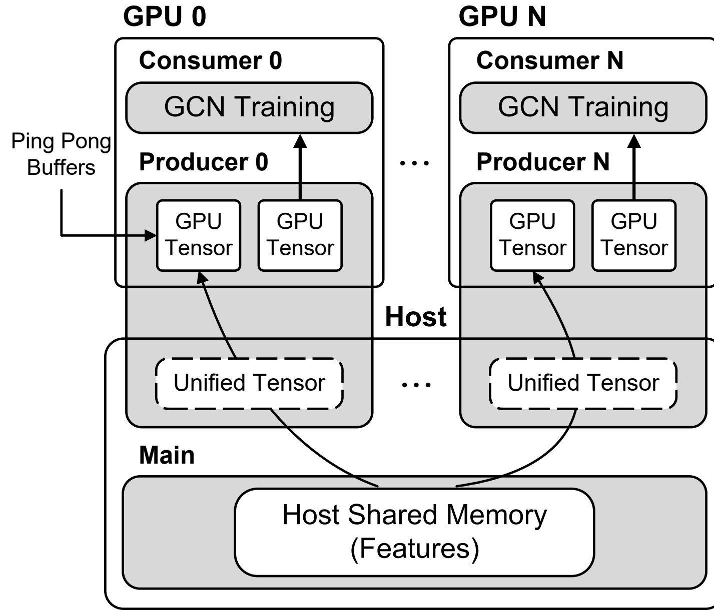


Figure 3.9: Simplified view of Multi-GPU GNN training flow. All unified tensors provide identical mappings. Sampling processes and indexing kernels are omitted in this diagram.

### 3.3.4 Multi-GPU Training

The final challenge of our method is supporting a multi-GPU training environment. Multi-GPU training is one of the keystones of modern ML for reducing the training time, and the existing DMA-based method already supports the multi-GPU training. Therefore, it is not useful or practical to propose a method that can only support a single-GPU training environment.

For multi-GPU training, we take the data parallelism approach used in DGL [7]. In the original DGL implementation, the multi-GPU training is done by increasing the number of sampler-consumer pairs and assigning one GPU to each pair. On top of the DGL implementation, we add the GPU-based producer process into each pair. The DGL implementation does not have a dedicated producer process as it assumes the node feature data is collected by the sampler and transferred into each GPU's memory. The simplified diagram of our multi-GPU training design is shown in Figure 3.9.

The main difficulty of multi-GPU training with zero-copy accesses lies in sharing the same host memory space across different GPUs running in different processes. In general, sharing the host memory space across different GPUs is simple when the kernels are launched by a single process.

In this case, programmers simply need to allocate a memory space by either calling `cudaHostAlloc()` with a `cudaHostAllocPortable` flag or calling `cudaMallocManaged()`. However, with this method, the host memory space allocated is bound to the process which called the memory allocators. Currently, due to how the CUDA memory allocators work, there is no way for the users to make the space allocated by them to be shareable with the other processes. It is possible to create multiple copies of node feature tensors for each training process, but this leads to extremely inefficient usage of host memory capacity.

```
import torch
import torch.multiprocessing as mp

def producer(features, process_id, ...):
    #Specify target GPU ID
    torch.cuda.set_device(process_id)
    #Map host shared memory to GPU address space
    features = features.to(device="unified")
    ...

if __name__ == '__main__':
    features = torch.randn([100], device="cpu")
    # Allocate shared memory space
    features = features.share_memory_()
    ...
    #Pass feature tensor alloctead in shared memory space
    #and call producers for multiple GPUs
    producer1 = ctx.Process(target=producer,
                            args=(features, 0, ...))
    producer2 = ctx.Process(target=producer,
                            args=(features, 1, ...))
    ...
```

Listing 3.3: Unified Tensor Declaration in Multiprocessing Environment

Therefore, we take the opposite direction of memory allocation in our implementation. Instead of sharing a memory space after allocating with CUDA APIs, we first allocate a shareable memory space and then call CUDA APIs to allow GPUs to access the space. In Linux, to allow multiple processes to share the same memory space, *shared memory* can be used. Here, this *shared memory* refers to a specific Linux implementation to allow interprocess communication, and it should not be confused with other similar terminologies,

such as the GPU shared memory. We utilize the `cudaHostRegister()` API because it can be used on top of the Linux shared memory. Therefore, by letting different processes call `cudaHostRegister()` individually on the same shared memory space which has been already allocated, each GPU can get identical address mapping to the same host memory space. The specific code that implements this approach is shown in Listing 3.3. Line 14 shows the declaration of a shared memory tensor in the main process. Shared memory allocation is already supported in PyTorch code by simply adding `.share_memory_()` command after the CPU tensor instance. To map this shared memory space for different GPUs, we pass the shared CPU tensor to the producer processes running on the GPUs (Lines 18 and 21). Each producer process simply calls the unified tensor declaration (Line 8) to effectively convert the shared CPU tensor into a unified tensor and maps it into the GPU's address space for zero-copy access.

Inside the producer code (Lines 4-9), the first thing that we must do is select the correct CUDA device (e.g., producer0 $\rightarrow$ GPU0, producer1 $\rightarrow$ GPU1, and so on). Without this step, all unified tensor declarations in different producer processes will create a mapping for the default CUDA device defined by the system (e.g., GPU0).

## 3.4 Evaluation

This section presents an evaluation of the impact of our proposed design on GNN training time. We first take a closer look at the improvements made by our optimizations one by one, and then show the overall training time reduction achieved.

### 3.4.1 Methodology

#### Evaluation System

For our evaluation, we use the system described in Table 3.3. Our host system can hold two RTX 3090 GPUs and both are operating in PCIe 4.0 mode. With PCIe 4.0 interconnects, both GPUs can achieve about 25.8 GB/s of host to GPU DMA bandwidth in our microbenchmark. The measured

aggregated bandwidth of the two GPUs performing DMA on host memory at the same time is about 51.7 GB/s.

### Application

Our unified tensor implementation and the indexing kernel modification are based on PyTorch 1.8.0-nightly version. For the GNN training, we use the GraphSAGE [65] implementation of DGL [7]. We only modify the data communication portion of the implementation. The sampling mechanism and the training algorithm remain unmodified.

**(a) CPU-Only** implementation only uses CPU for training GNN. In this case, there is no need for data transfer over PCIe since GPUs are not involved in the training.

**(b) DMA-based** implementation uses CPU to gather node features into a contiguous buffer. The gathering process in CPU is multithreaded by default in PyTorch and the data transfer time is overlapped with the training time by using asynchronous DMA.

**(c) Naïve Zero-Copy** uses zero-copy as a main data transfer method, but does not include any optimizations we discussed in this chapter. Unified tensors are used to allow GPUs to perform zero-copy accesses on host memory.

**(c) Zero-Copy** implementation enables zero-copy accesses and additionally includes all optimizations we discussed in this chapter. Unified tensors are used to allow GPUs to perform zero-copy accesses on host memory.

**(d) All-in-GPU** implementation allocates the entire node feature array into each GPU memory before the training begins. This implementation is used to show the rough upper bound of the performance improvement we can achieve through the data transfer optimization. Due to the limited GPU memory capacity, we do not evaluate all datasets with this implementation. We explicitly denote such cases as "out-of-memory (OOM)".

### Dataset

In Table 3.4, we show the datasets we used for the evaluation. `wikipedia` [86] network consists of the wikilinks of Wikipedia in the English language. Nodes are Wikipedia articles, and directed edges are wikilinks. `amazon` [87] dataset

Table 3.3: Evaluation system configuration.

| Category | Specification |
|---|---|
| CPU | AMD Ryzen Threadripper 3960x 24C/48T |
| Memory | DDR4 3200 MHz 256 GB in Quad Channel |
| GPU | 2x NVIDIA Ampere RTX 3090 24 GB |
| OS | Ubuntu 20.04.1 & Linux Kernel 5.8.0 |
| S/W | CUDA 11.2 & PyTorch 1.8.0-nightly |

Table 3.4: Evaluation dataset.

| Name | #Feature | #Node | #Edge | Size |
|---|---|---|---|---|
| ogbn-products | 128 - 4096 | 2.4M | 61.9M | - |
| wikipedia | 315 | 13.6M | 437.2M | 17.1 GB |
| amazon | 578 | 14.7M | 64.0M | 34.0 GB |
| ogbn-papers100M | 128 | 111.1M | 1.6B | 56.9 GB |

is based on Amazon product network connected by "also viewed" and "also bought" links. `ogbn-papers100M` dataset is a directed citation graph of 111 million papers indexed by MAG [88]. The above datasets are used for basic performance evaluations. `ogbn-products` [89] dataset is based on Amazon co-purchasing network [90] where nodes represent products sold in Amazon, and edges between two products indicate that the products are purchased together. `ogbn-products` is only used for the training time vs. node feature size sensitivity analysis on Section 3.4.4.

### 3.4.2 Bandwidth Analysis

In Figure 3.10, we show the comparison of the effective bandwidths we measured during the wikipedia dataset training. To observe the impact of the misaligned node feature access on the PCIe bandwidth, we sweep the node feature size from 1024 B to 1044 B in this experiment. Zero-copy naïve approach does not implement the circular shift optimization we discussed in Section 3.3.2. Throughout the experiment, the effective bandwidth of the DMA-based approach is only about half of the zero-copy approaches as it requires a long CPU gathering process.

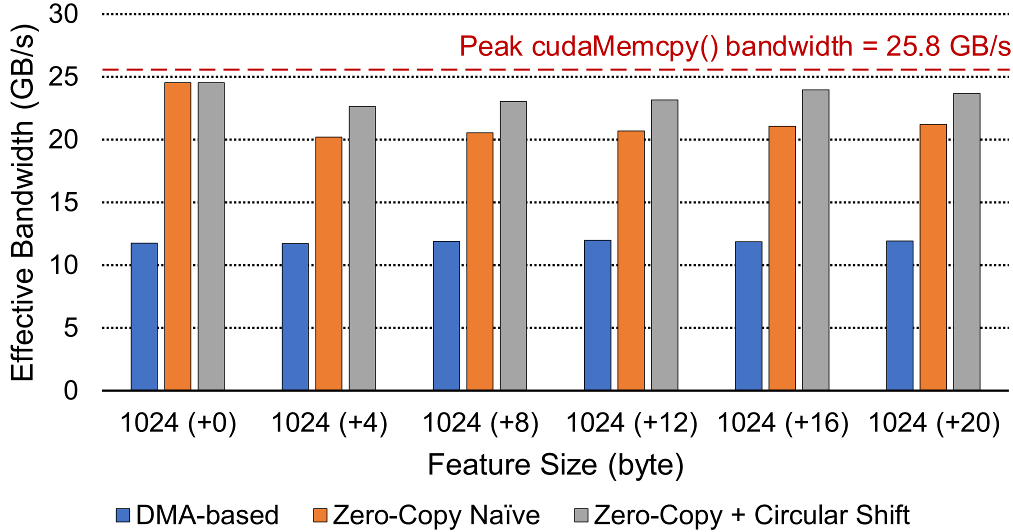


Figure 3.10: Effective data transfer bandwidth measured during the wikipedia dataset training. We sweep feature size to observe the impact of misaligned zero-copy accesses over PCIe.

When the node feature size is 1024 B, regardless of the circular shift optimization existence, the zero-copy implementations show the best effective bandwidth numbers. Because the GPU cacheline size is 128 B, in this case accessing any node features results in generating perfectly coalesced accesses. Considering that the best `cudaMemcpy()` bandwidth we achieved is about 25.8 GB/s, we can roughly estimate the upper bound efficiency of zero-copy access is about 95.1%. With more misaligned accesses, the efficiency of the naïve zero-copy implementation drops to 78–82% while the optimized zero-copy implementation can achieve 88–93% of efficiency.

In general, the results re-emphasize the importance of making cacheline-aligned accesses whenever using zero-copy accesses. For savvy programmers, we expect them to understand the underlying hardware mechanism and to consider padding the input data if the overhead is not too big. However, even if they fail to do so, our optimizations would still reduce the performance penalty for them.

### 3.4.3 Concurrency Analysis

The best way to check if our MPS-based resource provisioning is helping concurrency is to profile the workload and visually inspect the GPU kernel

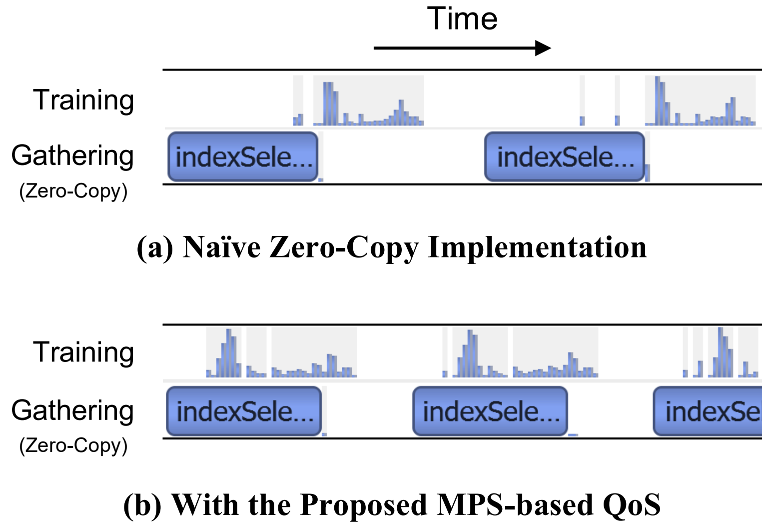


Figure 3.11: Snapshots of NVIDIA Nsight Systems Profiler during GNN training. CPU workloads not shown here.

timeline. In Figure 3.11, we show two profiling results of GNN training where (a) we do not apply any resource restriction and (b) we allocate 10% of GPU resource for the zero-copy kernels and 90% for the training kernels. Without any MPS running, there is almost no concurrency occurring since each kernel is trying to consume the whole GPU resource. In this specific case, the indexing (zero-copy) kernel is blocking other training kernels using GPU resources. The training kernels are already scheduled into the queue, but most of them cannot be actually executed until the zero-copy kernel is finished. Only a few kernels which require a small amount of GPU resource can be executed along the zero-copy kernel. In the NVIDIA tools, the GPU is considered to be 100% utilized at this point, but as we discussed in Section 3.3.2, in fact only a limited number of cores can actually submit memory requests over PCIe due to the protocol limitation (i.e., up to 768 pending PCIe requests over PCIe). Most of the cores are simply stalled, waiting for their turns to submit memory requests.

On the other hand, when we enable the MPS and limit the GPU resource usage for the zero-copy kernel to 10%, it does not block the following training kernels anymore. Furthermore, even though the zero-copy kernel can now use only up to 10% of the GPU resource, there is no significant bandwidth drop. In Figure 3.12, we show the zero-copy PCIe bandwidth change over allocating different amount of GPU resource to the zero-copy kernel. With

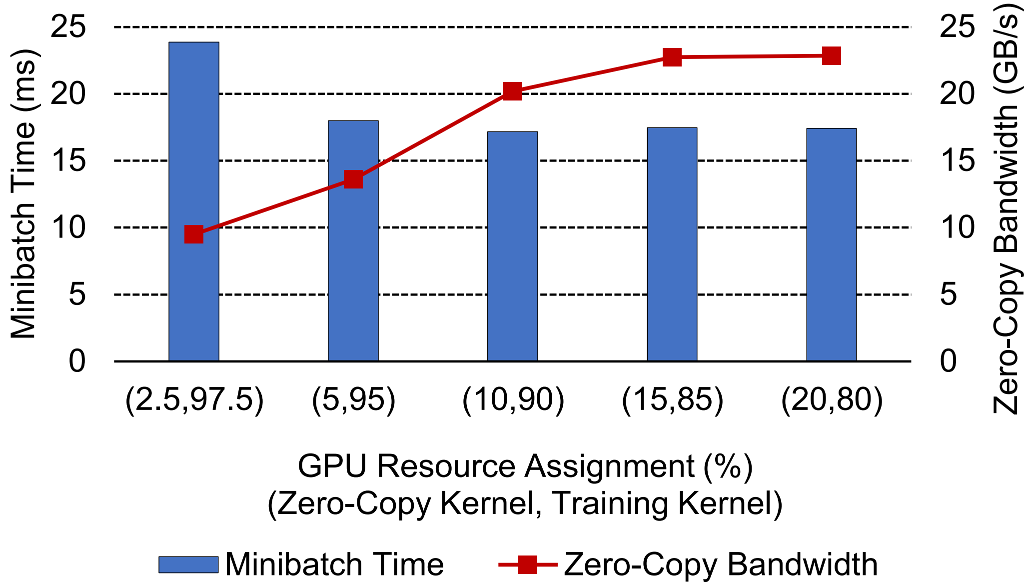


Figure 3.12: MPS resource partitioning ratio sensitivity analysis.

2.5–10% of resource allocations, the zero-copy kernel cannot generate enough PCIe read requests and therefore the measured PCIe bandwidth is limited to 9.5–20.2 GB/s. Further increasing the GPU resource allocation can make the zero-copy bandwidth reach around 23.0 GB/s, but we do not observe any significant improvement after 15% of allocation. At this point, the amount of GPU resource allocated is excessive and we have already reached the maximum number of PCIe read requests that we can generate. The results roughly fall into the estimation we made in Section 3.3.2. If the other users want to apply the same optimization technique on different types of GPUs, the same methodology we used to make the estimation could be useful.

For the training time, 2.5–5% of resource allocation is not enough to overlap (hide) the zero-copy kernel time with other processes and therefore the minibatch time is longer than the optimal case. We achieve the best minibatch training time when the resource allocation is 10%. With more resource allocation on the zero-copy kernel, the computation kernels start to starve from lack of GPU resource. If one wants to apply the same technique for different types of workloads, it might be worthwhile to fine-tune the ratio. However, still, one must be aware of the PCIe bandwidth limit. For the rest of our evaluations, we simply use an allocation ratio of 10:90 since the minibatch time is quite stable with small variations in the allocation ratio.

Additionally, we test how the usage of vector datatypes can further reduce

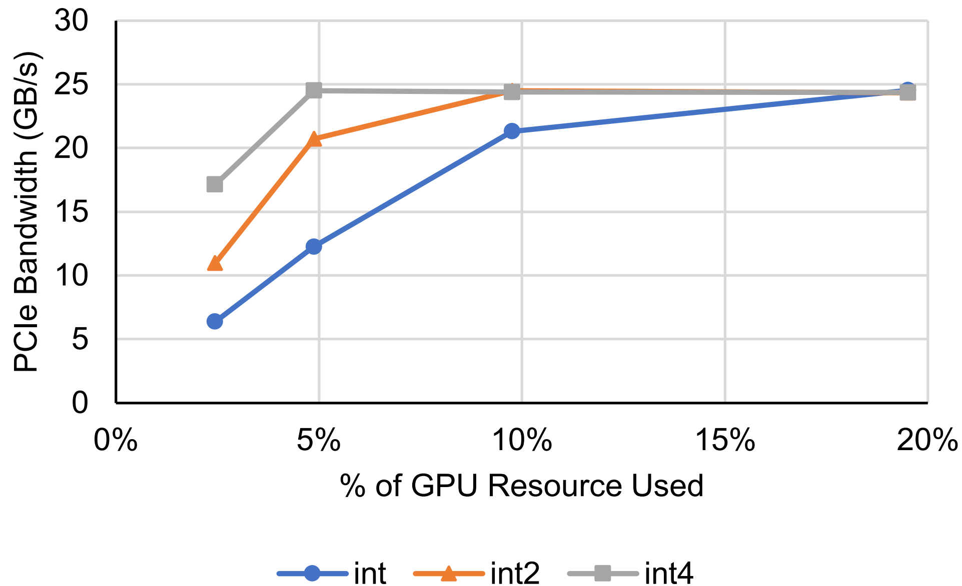


Figure 3.13: PCIe bandwidth comparison with different vector datatypes (`int`: 4 B, `int2`: 8 B, `int4`: 16 B) used to make the zero-copy access. With a larger vector datatype, we can use fewer GPU cores to saturate the PCIe bandwidth.

the amount of GPU resources used by the zero-copy kernel. Figure 3.13 shows the GPU resource requirement to saturate the PCIe bandwidth while using different datatypes for the zero-copy access. CUDA supports custom `int2` and `int4` vector datatypes, which are 8 B and 16 B, respectively. By using the `int4` datatype, we can theoretically reduce the number of threads required to saturate the PCIe bandwidth to $1/4$ compared to the `int` datatype. For example, Figure 3.13 shows that while we need to use about 20% of GPU resources to saturate the PCIe 4.0 x16 with the `int` datatype, it takes only about 5% with the `int4` datatype. If the computation power demand of the training model is critically high, then additionally applying those vector-type-based zero-copy access optimization can be considered.

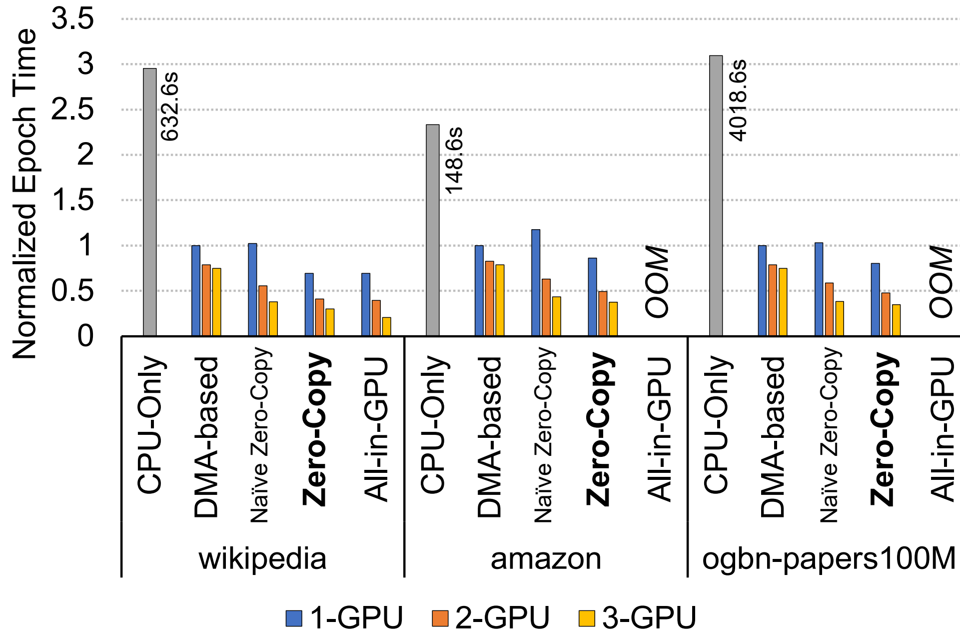


Figure 3.14: GNN training time comparison. OOM denotes out-of-memory.

### 3.4.4 Training Performance

#### Overall Comparison

In Figure 3.14, we show the overall training performance comparison. Throughout the entire comparison, the CPU-only case shows the worst performance. By limiting the computation unit to CPU, there is no need to worry about efficient data transfers over PCIe, but at the same time the computation power is severely limited. In general, the CPU-only method is 32–43% as fast as the DMA-based method. This performance difference is an important motivation for moving the training part into the GPUs.

For the DMA-based method, doubling the number of GPUs does help reduce the overall training time, but an additional GPU results in only 21-27% performance improvement across different datasets. Because the DMA-based method amplifies the usage of CPU resource and host memory bandwidth, increasing the number of GPUs throttles the entire training process. At this point, we observe that the host memory is 100% utilized. The CPU gathering process is a massive memory copy operation and we often observe that any other overlapping processes slow down in comparison to the usual cases.

For the naïve zero-copy method, we observe a 2–17% performance degra-

dation compared to the DMA-based method in a single-GPU setup. This degradation is consistent with the conventional wisdom that naïve zero-copy is inferior to DMA-based methods. Without our proposed optimizations, the zero-copy method suffers from the low bandwidth and the serialization issues described in Section 3.3.2 and Section 3.3.2. This result also gives us an idea how the programmers can make a premature conclusion to not further investigate the usage of zero-copy accesses.

With two GPUs, the naïve zero-copy method shows much better performance as well as performance scalability than the DMA-based method. In a dual-GPU setup, the naïve zero-copy method becomes 30–41% faster than the DMA-based method. This is because, even without the optimizations, the zero-copy method by default much more efficiently uses the CPU resource and host memory bandwidth than the DMA-based method. However, this benefit is not visible until the number of GPUs increases.

Finally, with our zero-copy optimizations, we can now clearly see the benefit of zero-copy in all cases. In a single-GPU setup, the optimized zero-copy method is 16–44% faster than the DMA-based method, and in a dual-GPU setup it is 65–92% faster. More surprisingly, with all optimizations included, the performance of the zero-copy method matches with the all-in-GPU method for the `wikipedia` dataset training. Since the data communication time is completely hidden by the training process in this case, there is no disadvantage compared to the all-in-GPU method. Overall, we observe a very significant benefit of using zero-copy accesses for GNN training. Thanks to the design flow optimizations we discussed in Section 3.3.3, we do not observe any performance impact from the interprocess communications.

#### Node Feature Size Sensitivity Analysis

Even though we use multiple different graphs to evaluate GNN training performance, some of other real-world datasets can have very different node feature dimensions. For example, the node feature dimension of Pinterest dataset [66] is about 4096, which is far larger than the node feature dimensions in our datasets. However, many of those real-world datasets are proprietary and difficult to obtain for academic purposes. Therefore, in this section, we artificially sweep the node feature dimension of `ogbn-products` dataset and compare the performances of zero-copy method and DMA-based

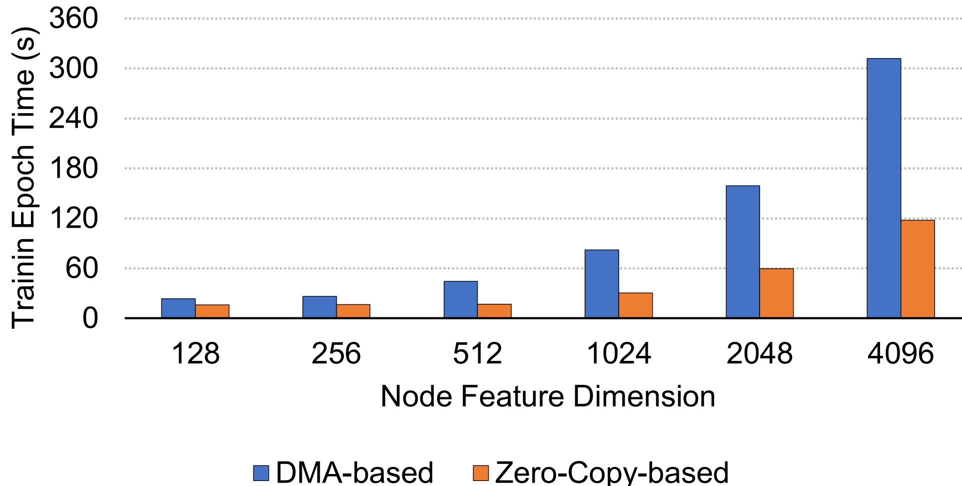


Figure 3.15: Node feature size sensitivity analysis.

method.

In Figure 3.15, we show the node feature size sensitivity analysis results. In this experiment, we use two GPUs. With small node feature dimensions, the zero-copy method is only 1.5–1.6× faster than the DMA-based method. However, when the node feature dimension is 4096, the zero-copy method is about 2.7× faster than the DMA-based method. This is an expected behavior as with small node feature dimensions, other overheads in the GNN training processes take a sizable portion of training time and therefore the data transfer time is relatively less important. This experiment result stresses the necessity of using efficient data communication architecture when training GNN with large node feature dimensions.

## 3.5 Discussion

**Physical GPU Resource Partitioning.** Even though the CUDA MPS is already providing workload partitioning service, it is still a logical partitioning rather than a physical partitioning. To guarantee a stable and high zero-copy PCIe bandwidth, it is better to physically isolate the GPU resources used by the training kernels from those used by the zero-copy kernels. With recent introduction of Ampere architecture GPUs, NVIDIA started to support partitioning of a single GPU into multiple GPU instances [91]. Each

GPU instance has dedicated GPU memory resources which limit both the available capacity and bandwidth, and provide memory QoS. Currently it is impossible to share the same piece of data between different GPU instances, but such advances in supporting hardware partitioning capability can potentially help isolate the zero-copy kernels with better QoS in the future.

**DMA vs. Zero-Copy Trade-Off Point.** By increasing the length of feature dimension, the DMA method may become efficient enough to send each node feature to GPU without the need for CPU gathering. To understand this trade-off, we conduct a microbenchmark measuring the setup time of asynchronous DMA. In this microbenchmark, we send a series of DMA requests back-to-back. The microbenchmark result shows that the average DMA setup time is about 3.16 $\mu$s in our setup, which is identical to the time of sending 85.5 kB of data over PCIe 4.0 x16. Therefore, if the size of individual item is larger than 85.5 kB, using a series of DMA operations can be an alternative to the zero-copy method. For the GNN training, our current largest node feature size is 16 kB, as shown in Section 3.4.4, which is only about 18.7% of the 85.5 kB requirement.

**Applications Beyond GNNs.** Our work benefits machine learning models other than GNNs as well, as embedding is a widely adopted technique to represent entities, especially when the data scale is large. For example, Facebook's large-scale recommendation systems model DLRM [92] involves a sparse embedding lookup process, thus benefiting from our work [93, 94]. Aside from models for large-scale recommendation systems, our work also benefits some traditional machine learning operators. For instance, the Hummingbird compiler [95] converts many supported machine learning operators to tensor operations. Tensor operations of tree models after conversion show the same feature gathering challenge in batch inference. Therefore, such scenarios would identically benefit from our work when the input data size is large.

**Comparison with Remote DMA (RDMA).** The main performance benefit of our work comes from eliminating the intermediate data buffering for the DMA data transfer. A similar technique was previously introduced in the networking field, where usually the Ethernet packets were first buffered in the kernel space. The kernel space packet buffering was causing a huge host memory bandwidth waste, and now many high-speed network interface cards (NICs) support remote DMA (RDMA) [96, 6], which directly delivers data

to the userspace without going through the kernel space. This elimination of the "intermediate data copy" is the same way we improve the application performance in our work. Yet, one main difference of RDMA compared to our work is that in our work, the data transfer is initiated by accelerators while the RDMA is still set up by CPU. Therefore, the RDMA operation still needs the CPU-device synchronization which may add some more latency in data transfers.

## 3.6 Conclusion

In this chapter, we introduced a GPU-oriented, software defined data transfer architecture for efficient GNN training on large graphs. In large-scale GNN training, one of the most challenging tasks is to efficiently transfer node features scattered in the host memory to GPUs. As opposed to the traditional DMA-based method, we directly utilize GPU cores as a data moving agent to access sparse features in the host memory over zero-copy accesses. Our evaluations show that together with zero-copy accesses and our optimizations, the GNN training performance can be improved by 65–92%. Furthermore, the benefit of our proposed approach is significantly larger for 2-GPU training than 1-GPU training. By implementing the end-to-end zero-copy based GNN training flow in PyTorch, we also show that our modifications can be seamlessly integrated with the existing high-level DNN programming models.

# CHAPTER 4

# DATA ACCESS OPTIMIZATION IN MULTI-LEVEL MEMORY HIERARCHY

## 4.1 Introduction

A modern system's memory hierarchy may contain several types of memory-attached devices such as host, local, and peer devices. Often those memory devices are connected to each other using different grades of interconnects, and optimizing data placement among those memory devices can have a significant impact on application performances [8]. While partitioning dense datasets over multiple memory devices is straightforward, it is less clear how to partition sparse datasets over multiple memory devices due to the irregular memory access pattern. However, with the use of fine-grained memory access over I/O, this challenge can be mitigated.

This chapter proposes unifying all memory access types with fine-grained memory access over I/O, such as zero-copy access, to simplify memory access to different kinds of memory devices. The use of fine-grained memory access over I/O eliminates the need for runtime sparse dataset transformation for the data transfers and significantly reduces the programming complexity. With the reduced programming complexity, programmers can attempt more advanced data placement strategies for different memory devices to better utilize high-bandwidth interconnects in the system.

In this chapter, we apply the idea of *Data Tiering* with the GNN training, that we have shown in Chapter 3 to demonstrate the effectiveness of the fine-grained memory access over I/O in multi-level memory hierarchy systems. Our data tiering technique improves GNN training by providing a statistical method to effectively predict the access frequency of each node feature in the input graphs and identify which nodes should be located in the GPU memory. The less frequently accessed node features can be located in the peer-GPU's memory or the CPU memory and we can access them in runtime

*without incurring additional control sequences* because now with the fine-grained memory access over I/O, we do not need the peer-GPUs and CPU to gather those node features at runtime.

We evaluate our work using public frameworks PyTorch and DGL. The demonstration of our work on realistic GNN training shows that our approach eliminates PCIe traffic by 87–95% in various datasets by loading only 10% of them into GPU memory. With the data transfer time optimization alone from our data tiering strategy, we find the training speeds of the existing GNN implementations can be improved by 1.6–2.1×. To demonstrate the scalability of our work, we also train a dataset with 350 GB of size in a system with four NVIDIA V100 32 GB GPUs.

## 4.2 Data Tiering

### 4.2.1 Scoring Function

By definition, the neighborhood sampling process is random, and it is difficult to predict exactly which nodes will be sampled during training. Thus, we must statistically approach the problem of identifying and exploiting locality. The first metric we can use is the out-**degree** of each node in the input graph. With a high out-degree, even if the node is not selected in a specific run of neighborhood sampling, the cumulative chance of the node being selected during the entire training process is higher than that of the less connected nodes. Considering that we perform quite a significant number of samplings per training epoch for the large graphs, this prediction is statistically reasonable, as we empirically prove in Section 4.4.2. In the case of ogbn-papers100M, we sample about 130 million nodes per training epoch.

The second option is a reverse pagerank (**R-Pagerank**) [97]. In the original pagerank, the score of each node is higher if the in-degree is higher and the out-degree is lower. For the reverse pagerank, it is the opposite. In Figure 4.1, we depict the difference between the original pagerank and the reverse pagerank further. For simplicity, we only show a case of node $A$ with single iteration, but this is done for all nodes until the score values converge in the real implementation. In the original pagerank, to calculate the score of the source node, we sum the scores of the nodes which are targeting the

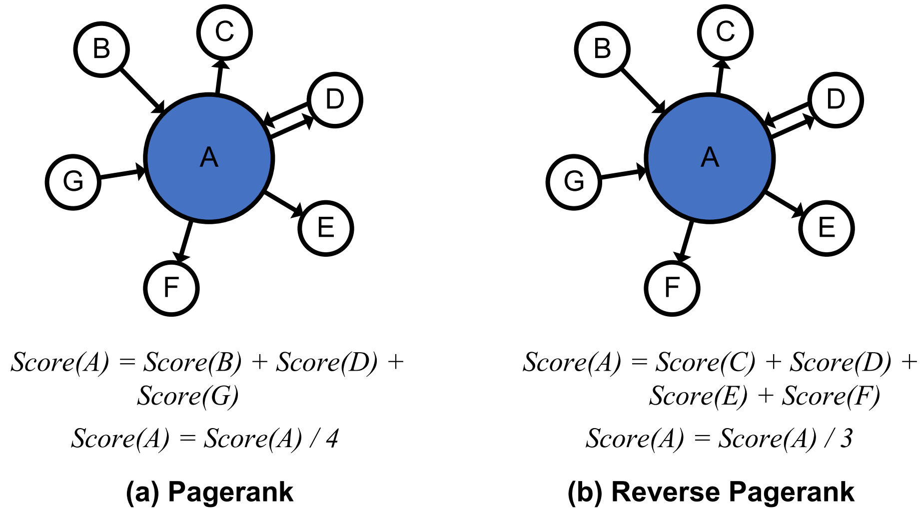


Figure 4.1: Snapshot of Pagerank vs. Reverse Pagerank. Only a single iteration of algorithms shown. In case of regular pagerank, the score is divided by the **out-degree**, but in case of reverse pagerank, the score is divided by the **in-degree**.

source node and divide the summed score by the out-degree of the source node. With the reverse pagerank, we sum the scores of the nodes targeted by the source node and divide the summed score by the in-degree of the source node. The idea behind this mechanism is that if a certain node $A$ has many outgoing edges, it can potentially get a higher score by summing many nodes' scores. Therefore, if another node $B$ targets node $A$, node $B$ also gets a higher score by adding the score of node $A$.

In the context of neighbor sampling, the scoring mechanism of the reverse pagerank can be understood by referring to Figure 4.2. The green nodes are sampled while generating the embedding for the red node, referred to as node A, because they have an outgoing edge to A. The blue nodes are sampled because they have an outgoing edge to the green nodes. Therefore, if a node can reach many nodes directly or indirectly through its outgoing edges, it will likely be picked during the sampling process. Since the probability of node $A$ being picked is high, the other nodes targeting this node also have a higher likelihood of being picked when we are sampling multiple layers. However, if the node $A$ also has a high in-degree, because now there are so many nodes that can be sampled from node $A$, the other nodes have lower likelihood of being sampled when the node $A$ is selected. Thus, in this case, we divide the score of node $A$ by the in-degree before propagating it to the other nodes,

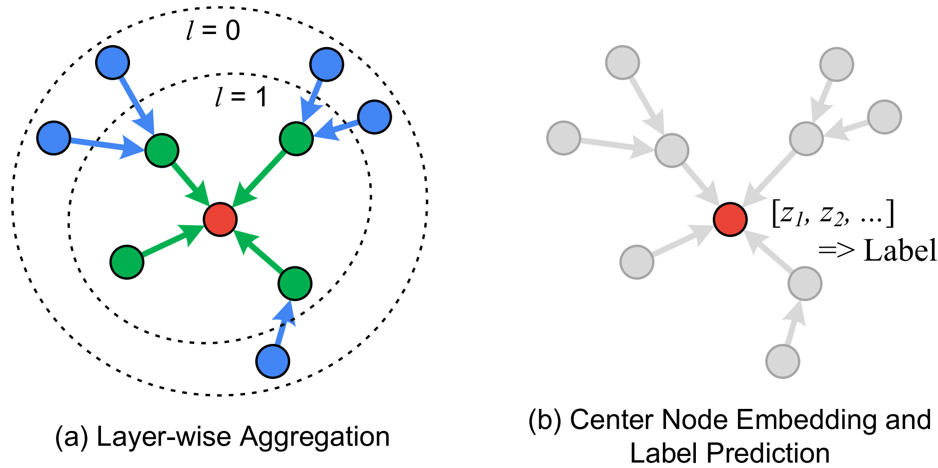


Figure 4.2: Node feature aggregation and label prediction.

so these nodes receive less increase to their estimated probability of being picked during sampling.

The potential advantage of using the reverse pagerank over the simple degree method is to capture further multi-layer sampling patterns. For the simple degree method, the information we can capture is limited to a single hop of relationship, while the actual neighbor sampling can extend to multiple hops. On the other hand, in reverse pagerank, the score value of each node is propagated to multiple layers of nodes away until the score converges to a certain limit. Therefore, by using the reverse pagerank, it is possible to capture the subtle pattern of multi-layer sampling in the neighbor sampling in a better way. To the best of our knowledge, this is the first work to utilize the reverse pagerank in GNN training, and the past works [98, 99] did not seek the opportunity to use a smarter scoring function than the degree-based method.

The third option is to further incorporate the labeling status of the nodes into the reverse pagerank method. As we explained in Chapter 3 Section 3.2.1, the goal of GNN training is to create a model which can predict the labels for the unlabeled nodes. To train such models, we must be able to compare the predicted labels with the ground-truth labels. Therefore, during training, the nodes we can pick to start the neighbor sampling are reduced to the nodes that come with labels. This means that if we can create a method to statistically put further emphasis on those nodes and their surrounding nodes, we can compress the search space.

**Algorithm 2** Weighted Reverse Pagerank

```
1:  Input: graph g, iteration iter, damp d, train_id tid
2:  num_node = num_nodes(g)
3:  num_train = length(tid)
4:  for i = 0 to num_node − 1 do
5:      score[i] = 1/num_node
6:      in_degree[i] = num_in_degree(g, i)
7:  end for
8:  weight = num_node/num_train
9:  for i = 0 to num_train − 1 do
10:     score[tid[i]] = score[tid[i]] ∗ weight
11: end for
12: for i = 0 to iter − 1 do
13:     for j = 0 to num_node − 1 do
14:         score[j] = score[j]/in_degree[j]
15:     end for
16:     pull_from_neighbor(g, score)
17:     for j = 0 to num_node − 1 do
18:         score[j] = (1 − d)/num_node + d ∗ score[j]
19:     end for
20: end for
```

To do this, we add some tweaks on top of the reverse pagerank algorithm by uniformly applying a weight value to the labeled nodes. The detailed implementation of the weighting process is described in Algorithm 2. First, before we decide how to weight the labeled nodes, we need to decide how much we want to weight them. The assumption behind the weighting is that by knowing the exact starting locations of the neighbor sampling, we can focus more on those nodes and their surroundings. This means that if there are few starting nodes available in the graph, the sampling tendency will be more biased toward them and their surrounding nodes. Inversely, if every node can be selected as a starting node, there is no starting bias and simply the nodes with high out-degree are likely to be selected during the sampling. As a result, the weighting intensity should be high if there are few labeled nodes, and the weighting intensity should be low if there are many labeled nodes. In our algorithm, we reflect this by defining weight = (# of all nodes) / (# of labeled nodes).

Next, the actual weighting is done by multiplying the initial scores of the labeled nodes with the weight value we calculated (Algorithm 2, Line #10).

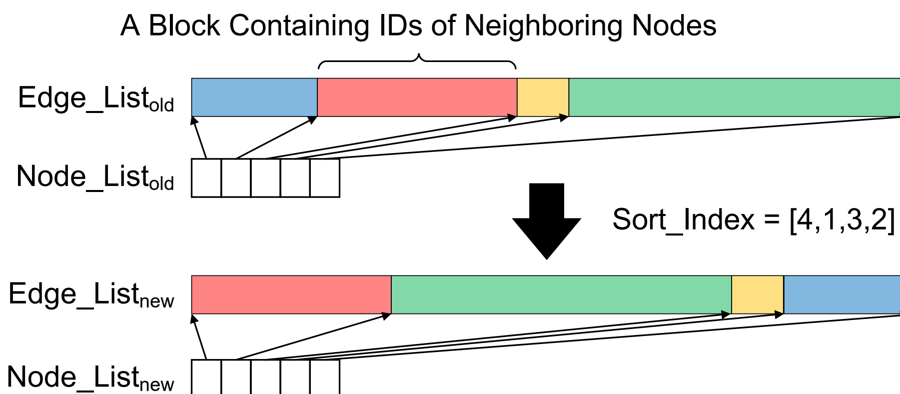


Figure 4.3: Adjacency Matrix Reordering Overview.

In the original pagerank algorithm, the default initial score of all nodes is one divided by the total number of nodes in the graph. In general, by running pagerank long enough, the initial impact of the initial scores wears down, and the scores converge to certain values. To avoid this, we do not run our weighted reverse pagerank until the scores converge, but only five iterations. The rest of the algorithm is identical to the reverse pagerank algorithm. We call this algorithm a weighted reverse pagerank (**Weighted R-Pagerank**).

### 4.2.2 Adjacency Matrix Reordering

With the score values, now we want to split the node features into a high score group and a low score group. The simplest way to achieve this is to sort the node features in the descending order of the score values and divide them into top X% of hot portion and bottom 100-X% of cold portion. To maintain the correctness of the GNN function, reordering the node feature tensor $H$ in Chapter 3 Section 3.2.1 also requires reordering the adjacency matrix $\mathcal{A}$.

However, unfortunately, the process of reordering the adjacency matrix is less intuitive than reordering the node feature tensor because the adjacency matrix is often represented by a sparse matrix format. Due to its counter-intuitive nature, the current implementation of adjacency matrix reordering in existing frameworks like DGL has a simple sequential implementation, but the sequential approach may consume a significant amount of time when the input graphs have hundreds of millions of nodes. Therefore, in this chapter, we implement our own parallel version of algorithm to accelerate

the reordering process.

To convey a better understanding of our implementation, we first briefly explain the adjacency matrix reordering problem in general (Figure 4.3). In compressed sparse representation (CSR), the adjacency matrix is divided into an edge list and a node list. The edge list is a collection of many neighbor lists where each contains the IDs of the connected nodes. To reorder an adjacency matrix, we need to perform the following three tasks: First, we need to create a new node list containing the neighbor lists' new starting indices. Second, we need to reorder the neighbor lists based on the new starting indices. Third, we need to relabel all node ID values inside all neighbor lists.

The key to parallelizing the workloads is to generate a full mapping of old to new IDs in advance, so the ID translations in the later processes become simple table lookup processes. The old to new ID mapping table is created based on the score values we generated from Section 4.2.1. The indices in this mapping table indicate the new placing order of the old neighbor lists. For example, if the table has [4, 1, 3, 2], that means the 1st neighbor list (blue in Figure 4.3) now should be placed in the 4th place, the 2nd neighbor list (red in Figure 4.3) now should be placed in 1st place, the 3rd neighbor list (yellow in Figure 4.3) now should be placed in 3rd place, and so on. Because the sparse matrix $\mathcal{A}$ has varying row sizes, unlike the simple 2-D matrix $H$, the new starting index of each neighbor list should be calculated using cumulative sum.

Once the cumulative sum is done, we simply need to copy & paste the old neighbor lists into the new edge list based on the new starting indices. This process can be easily done in parallel. After relocating the neighbor lists, we now need to update the ID values in each neighbor list. The update can be done by simply looking up the previously set mapping table, and this process can also be easily done in parallel. When we define $n$ as a number of nodes and $e$ as a number edges, the time complexity of this algorithm is either $O(nlogn)$ due to the score sorting during the mapping table creation, or $O(e)$ when the number of edges is very large. However, thanks to our parallelizable approach, we find the end-to-end adjacency matrix reordering takes only about 31 seconds with ogbn-papers100M dataset (Table 4.1), which has 111 million nodes and 3.2 billion edges.

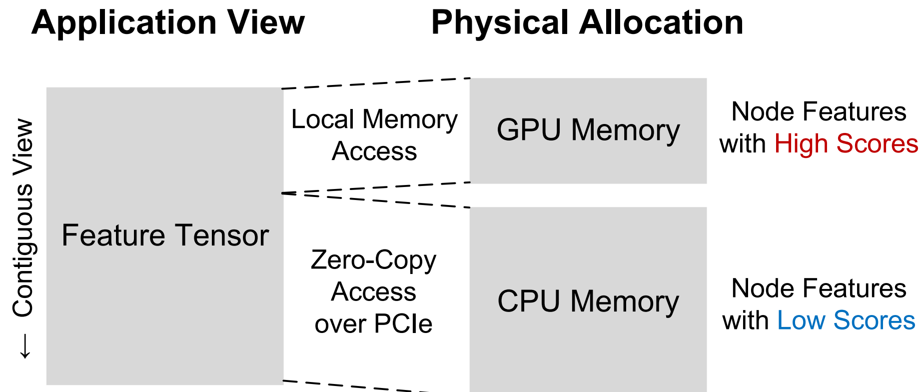


Figure 4.4: Simple data placement and access method overview.

## 4.3 Data Placement and Access

### 4.3.1 Memory Allocation and Indexing Scheme

The overall data placement and access strategy is shown in Figure 4.4. With the sorted node feature tensor, the *hot data* portion with a high score is placed in the GPU memory, and the rest of the *cold data* portion with a low score is placed in the CPU memory. From the user application perspective, we provide a single monolithic and contiguous fake view of the two tensors so the user application can use the existing array indexing scheme.

For the cold data access, it is important to maintain a low end-to-end data transfer overhead since crossing over PCIe is already a huge burden. One of the most common mistakes made by programmers during CPU-GPU data communications is that the programs often spend too much time coordinating the data transfer. Using cross-device data copy engines like DMA is widespread, but it is only effective when the size of data that we want to transfer is large enough. In the case of node feature sampling and aggregation, the data transfer size is typically between 512 B and 4 kB, much smaller than the size required to make DMA transfers efficient.

Therefore, in this chapter, we take a GPU-centric approach to accessing data instead of depending on DMA or CPU. At the hardware level, the GPU-centric method is enabled by using the zero-copy access capability of NVIDIA GPUs. The zero-copy accessible CPU memory space is mapped into the GPU page table, and it allows us to directly access the CPU memory space from the GPU kernel code. At the user (application) level, we utilize the DGL's

UnifiedTensor [100] implementation to enable this data access mechanism.

With our data placement and access strategy, the overall flow of node feature access in GNN training is as follows. First, the neighbor sampling function traverses the input graph and generates a list of sampled node IDs. Next, the node IDs are sent to GPU(s), and GPU threads start accessing the node features with the node IDs. While reading the node IDs, the GPU threads check if the ID values are within the hot data boundary set by the users. If the ID values are within the boundary, then the threads use a pre-stored GPU memory pointer and take advantage of fast local memory access. If the node ID is outside the boundary, then the threads use a pre-stored CPU memory pointer and perform the zero-copy access.

One major benefit of our approach is that we do not inflict any changes in the original programming structure. Our implementation of the fake tensor can provide the same experience to the users as if they are accessing a single large tensor as they did previously. The hardware-level details can be hidden behind the framework and the user-level understanding requirement is minimal. Unlike the other large GNN training solutions such as Roc [101] and NeuGraph [102] which mandate a new holistic pipeline restructuring for the data transfer optimization, only changing about 2-3 lines of code would be sufficient for our approach.

### 4.3.2 Tensor Distribution over Multiple GPUs

GNN training also extensively utilizes multiple GPUs to accelerate the training process, similar to the other neural network training methods. With fast GPU-to-GPU interconnects like NVLink, we can create a larger pool of collective GPU memory space (Figure 4.5) from multi-GPU systems. In Figure 4.6, we show the complete view of our data tiering strategy in a multi-GPU system. To load the hot data into this collective memory space, we use an interleaved data loading method instead of a naïve blocked partition method. Since the node feature tensor is sorted in descending order of the score, a simple block partitioning scheme can result in unbalanced memory and interconnect bandwidth consumption across GPUs. With the combined GPU memory space, we can hold a larger portion of hot data in a faster tier of memory space.

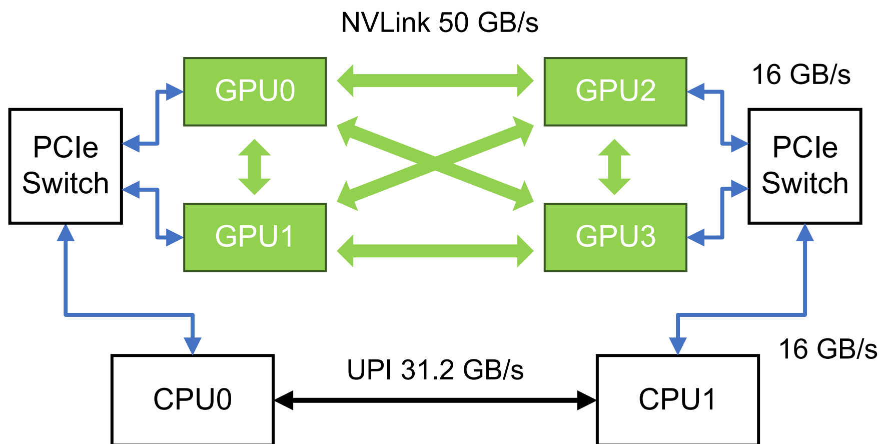


Figure 4.5: An example system with four NVIDIA V100 GPUs connected over NVLink. Unidirectional bandwidths shown.

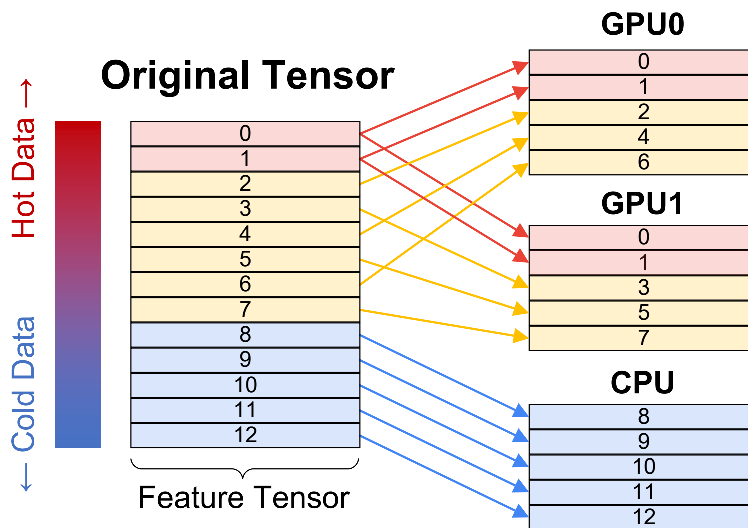


Figure 4.6: Available permutation of data placement in multi-GPU environment. Red: replicated in multiple GPUs. Yellow: evenly distributed over multiple GPUs. Blue: located in CPU.

From the implementation perspective, the combined tensor is basically a table which contains multiple pointers pointing to different memory locations in different devices (e.g., CPU and peer GPUs). Depending on the index value from the user space, we pick the corresponding pointer and let the GPU kernel access the node feature. The brief mechanism of this process is explained in Listing 4.1. The real CUDA kernel implementation of the indexing function is more complicated than the code provided here for several optimization purposes.

```
#define WARP_SIZE 32
void index_feature(int row_id, int **table, int *threshold,
    int feat_len, int gpu_num, int *output_feat) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int *row_ptr;
  local_boundary = threshold[0];
  multi_boundary = threshold[1];

  // Data is in local GPU tensor
  if (row_ids[i] < local_boundary)
    row_ptr = table[0] + feat_len * row_ids[i];
  // Data is in multi-GPU tensor
  else if (row_ids[i] < multi_boundary) {
    int offset = row_ids[i] - local_boundary;
    row_ptr = table[offset % gpu_num + 1] + feat_len * offset
    / gpu_num;
  }
  // Data is in CPU tensor
  else {
    int offset = row_ids[i] - multi_boundary;
    row_ptr = table[gpu_num + 1] + feat_len * offset;
  }

  for (int i = tid; i < feat_len; i+=WARP_SIZE)
    output_feat[i] = row_ptr[i];
}
```

Listing 4.1: Indexing the Combined Tensor

For the GPU memory sharing, we use the following APIs: `cudaMalloc()`, `cudaIpcGetMemhandle()`, and `cudaIpcOpenMemhandle()`. `cudaMalloc()` allocates a memory space in GPU, and `cudaIpcGetMemhandle()` creates a memory handle that can be shared with other processes to create a virtual

mapping of the originally allocated GPU space. In short, this *memory handle* can be understood as a medium to share the virtual mapping between two different processes. In action, `cudaIpcOpenMemhandle()` takes the memory handle created by `cudaIpcGetMemhandle()` and maps the other GPU's memory space into the GPU device that belongs to the current process. The overview of this process is shown in Figure 4.7.

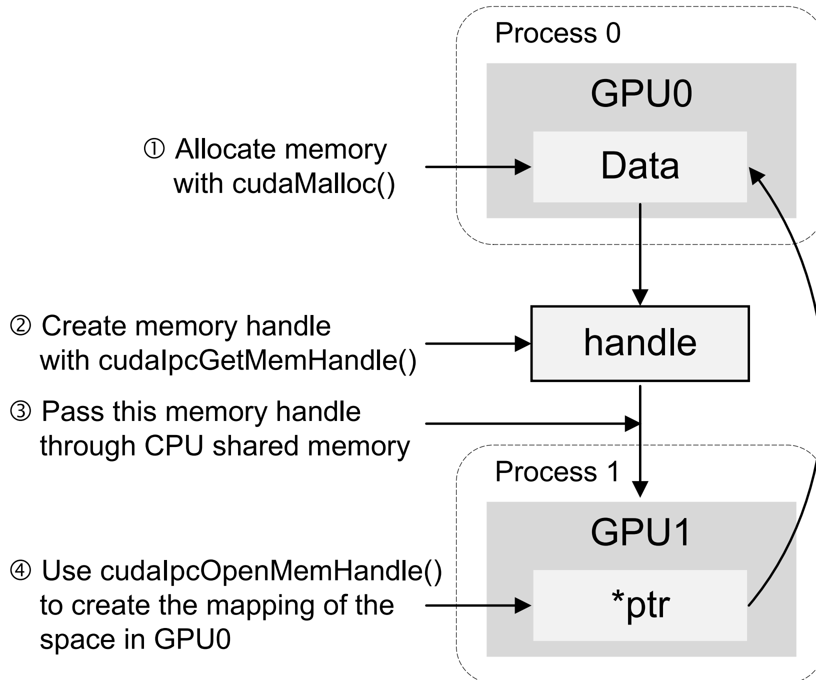


Figure 4.7: Peer-to-peer GPU memory sharing mechanism in multiprocessing setup.

In our experience, the NVLink bandwidth is fast enough to hide most of the data transfer time of GNN training, but in case the data transfer time is still an issue, we can additionally replicate some hot data over multiple GPUs. In this case, the most frequently accessed data will come from the local GPU memory, the next most frequently accessed data from the peer GPU memory, and the least frequently accessed data from the CPU memory. The generation of the combined tensors is fully automated in our implementation. To generate this combined tensor, users simply need to provide the sizes of local GPU tensor and multi-GPU tensor, and the number of GPUs connected over NVLink. If there are no high-speed links between GPUs, the mapping simply falls back to Figure 4.4. The optimal distribution factor of each GPU's memory capacity between the replicated hot data and the interleaved

hot data in the collective memory space may vary depending on the dataset. For our experiments, we simply maximize the multi-GPU tensor and do not utilize the replicated local GPU tensor.

## 4.4 Evaluation

### 4.4.1 Methodology

#### Application & Dataset

For the evaluation, we implement data tiering on PyTorch and DGL. Since neither of the frameworks supports kernel-level direct peer GPU memory access, we modify their tensor implementations to enable it. Currently, the frameworks can only perform peer-to-peer DMAs. We use GraphSAGE implementation of DGL to explore various neighbor sampling strategies. For the dataset, we use the following three from Open Graph Benchmark (OGB) [89]: ogbn-papers100M, MAG240M, and WikiKG90M. WikiKG90M is from a different task domain and does not come with the labels needed for node classification, but due to the lack of public large datasets we repurpose it as a node classification task dataset with synthetic label values. Further details of the datasets are shown in Table 4.1. To make our experiment realistic, we use the carefully tuned hyperparameters for different datasets which are taken from previous GNN training works with high-accuracy models [103, 104, 105]. Based on the previous works, we use (12, 12, 12) as neighbor sampling fanout parameter for ogbn-papers100M and (25, 15) for MAG240M. For WikiKG90M, we use the identical parameters used in ogbn-papers100M training.

#### Hardware

Throughout the evaluation, we use a machine with two Intel Xeon Gold 6230 CPUs and four NVIDIA V100 32 GB GPUs (Figure 4.5). All NVIDIA V100 GPUs are connected over NVLink with 50 GB/s unidirectional bandwidth per connection. Because each GPU is connected to three other peer GPUs, the aggregated NVLink bandwidth is $3\times50$ GB/s = 150 GB/s for each GPU.

Table 4.1: Evaluation datasets.

| Name | #Nodes | #Edges | Node Feature Total Size |
|---|---|---|---|
| ogbn-papers100M | 111.1M | 3.2B | 53 GB |
| MAG240M | 244.2M | 3.5B | 350 GB |
| WikiKG90M | 87.1M | 1.0B | 125 GB |

### 4.4.2 Scoring Function vs. Measured Data Reuse

In this experiment, we try to determine if the scoring functions discussed in Section 4.2.1 can correctly predict the data reusability in real GNN training. In Figure 4.8, we list the nodes of graphs in the X-axis in descending order of scores with the three different scoring functions: node degree, reverse pagerank, and weighted reverse pagerank. In the Y-axis, we show the measured access frequency of each node during GNN training in a cumulative fashion. In general, we find all functions can provide some benefits when we perform data tiering in GNN training. For example, based on the scores calculated, when we keep the top 10% of nodes, we can expect at least 35% hit ratio during the training regardless of the dataset. By keeping the top 25% of nodes, the minimum hit ratio further increases to 56%.

Also, in general, we find it becomes easier to predict which nodes would have high data access counts if a graph has a more extreme power-law distribution. For example, WikiKG90M graph has extremely unbalanced edge connectivity, and 80% of edges in the entire graph are connected to only 1% of nodes. With such extremely concentrated connections, we can observe that simply choosing a few nodes with the highest degrees automatically guarantees a very high hit ratio during the neighbor sampling. In cases of ogbn-papers100M and MAG240M, the edge connectivities of the datasets are more balanced, and the ratios are 32% and 46%, respectively. However, even though the simple degree method can be effective for certain graphs, we find the weighted reverse pagerank is preferable in general because it consistently gives the best prediction result as it best represents the multiple layers of neighboring node embedding accesses in the sampling process.

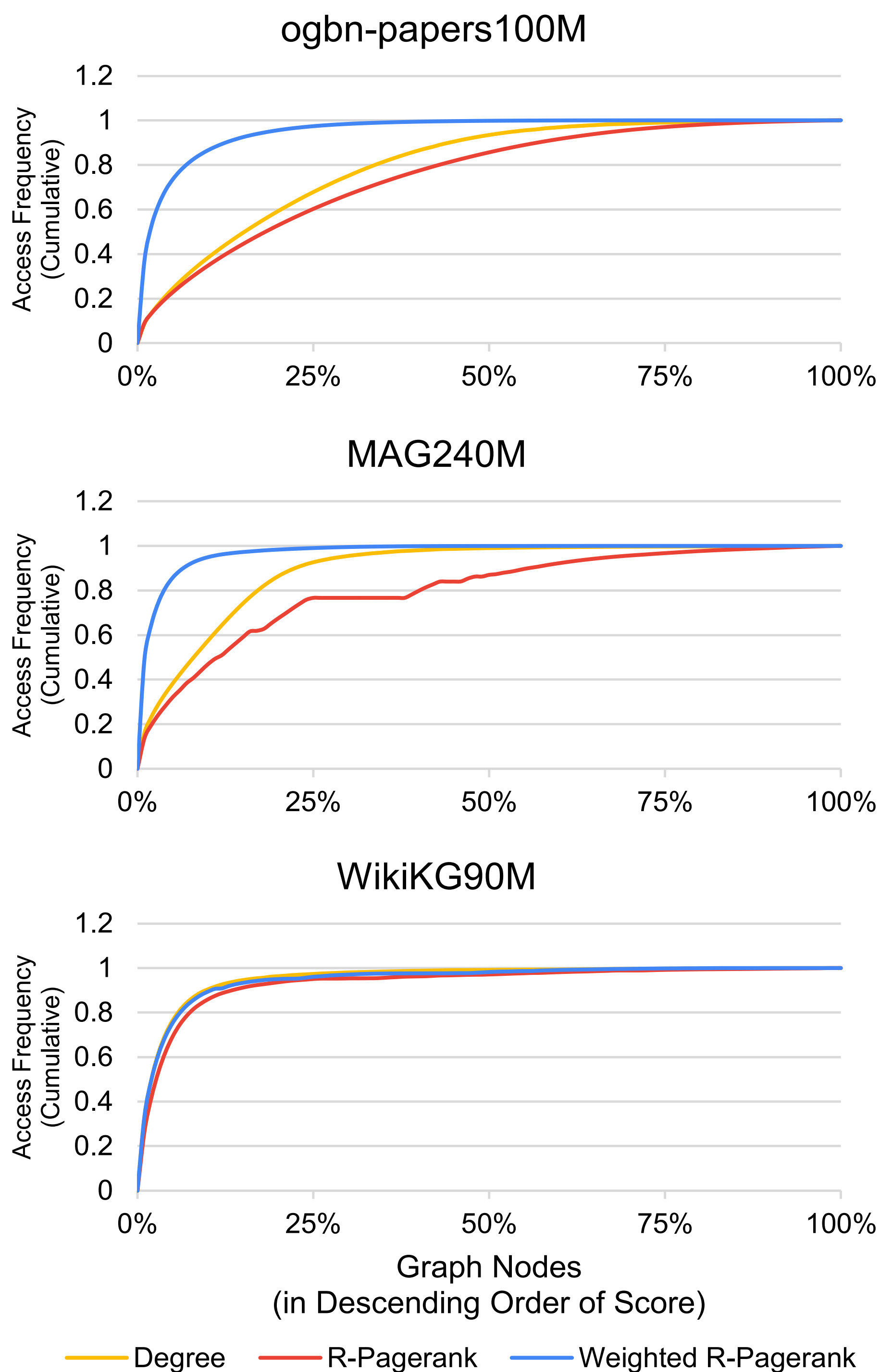


Figure 4.8: Access frequency distribution comparison on different datasets with different scoring functions. It is easy to cluster frequently accessed nodes with Weighted R-Pagerank function.

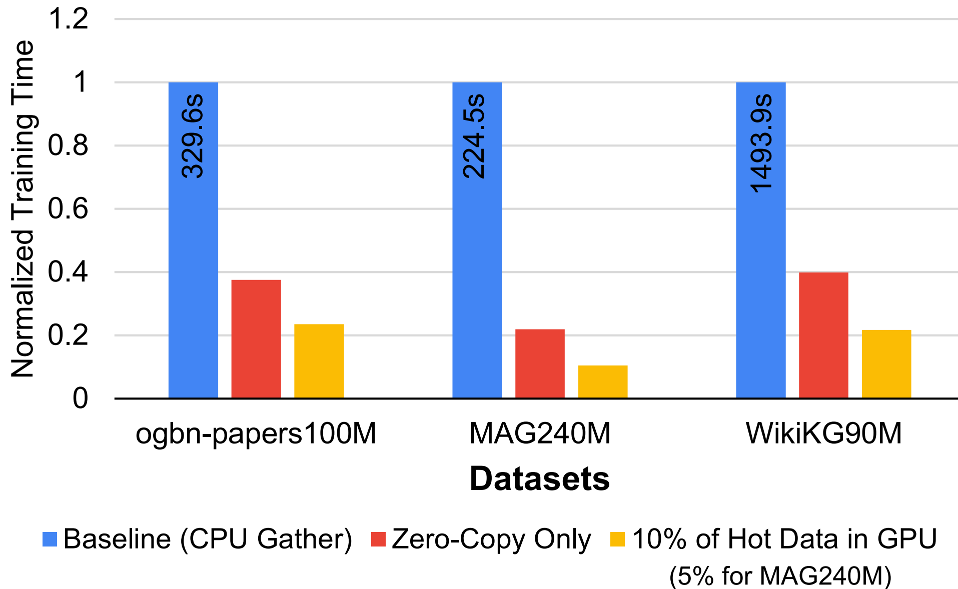


Figure 4.9: Single epoch training time comparison.

### 4.4.3 GNN Training Time (Single GPU)

In this section, we evaluate the actual benefit of our work in GNN training. We compare the performance of our work against the following two existing methods: (1) CPU gathering and (2) zero-copy access. The CPU gathering method relies on CPU to gather node features and then utilize GPU DMA engine to copy the gathered node features into GPU memory. Due to the additional data gathering process, this method wastes the CPU memory bandwidth and adds a non-negligible amount of data transfer latency to GPU. This is the only way currently available in PyTorch to transfer scattered data from CPU memory to GPU memory.

The zero-copy access method is a method recently introduced in Chapter 2 and adopted into DGL to overcome the data gathering overhead in the CPU gathering method. With this method, GPU kernels can directly dereference CPU memory pointers, and thus we do not need to rely on CPU to gather data for the GPUs. Recall that to enable the zero-copy access capability, DGL implements a new class of tensor called UnifiedTensor, which transforms the CPU tensor of PyTorch into a zero-copy accessible tensor. In UnifiedTensor, the specific task is done by utilizing `cudaHostRegister()` API from CUDA on top of existing CPU memory allocation. The further technical detail is identical to the process explained in Chapter 3, Section 3.3.1.

In our approach, we first score the nodes with the weighted reverse pagerank function like in Figure 4.8, and then reorder the node feature tensor and the graph nodes in the datasets. For this experiment, we load 10% of hot data for ogbn-papers100M and WikiKG90M, but only 5% for MAG240M due to the GPU memory limitation.

In Figure 4.9, we show the overall comparison. From this comparison, we can first observe that relying on CPU to gather data results in seriously increasing the overall training time. In this case, we find that the GPU is only about 10–30% utilized and mostly idling. By adopting the zero-copy access method, the training performance is significantly improved (2.5–4.6×). The zero-copy only method does not leverage any temporal data locality strategies, but simply removing CPU from the data access path significantly increases the overall performance. Finally, with our method, the training performance is further improved by 1.6–2.1× over the zero-copy only method. Considering that we have run the entire experiment on top of PyTorch and DGL with Python, the benefits that we observe are immediately deliverable to the regular users as well.

### 4.4.4 Case Study: ogbn-papers100M

Now, we take ogbn-papers100M as an example for more detailed analysis. First, even though we know that most existing GNN works use two to three layers of sampling depths, we still want to know how much increasing the sampling depth can affect the data tiering efficiency. To understand the impact, we use different sampling depths during the GNN training and observe how the node access frequency distribution varies. In Figure 4.10, we show two access frequency charts similar to Figure 4.8, but now with the varying sampling parameters of (10, 10), (10, 10, 10), (10, 10, 10, 10), and (10, 10, 10, 10, 10). For the scoring functions, we use the weighted reverse pagerank and the degree count. For both cases, we observe that the accesses are now more spread out with the deeper sampling parameters. This is expected behavior because with a deeper sampling depth, the graph coverage of each minibatch becomes larger, and we start to access secluded nodes more frequently.

However, even with the spreading out of the deeper sampling cases, we can still identify a significant portion of accesses that are made to a few

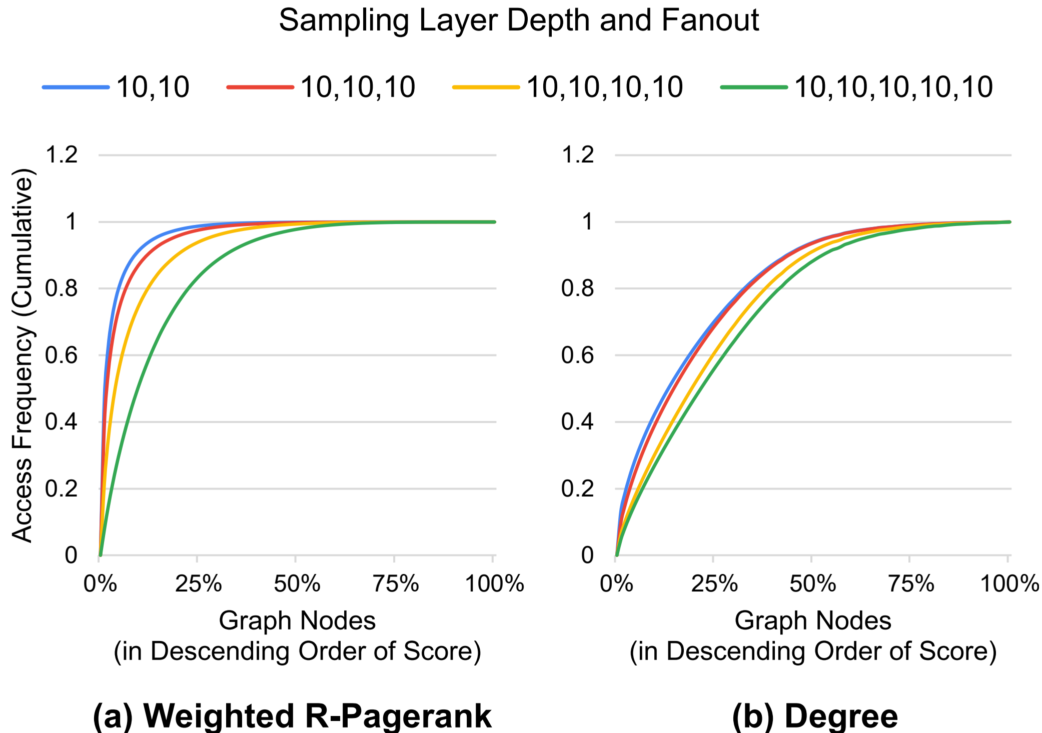


Figure 4.10: Access frequency distribution comparison of using different neighbor sampling depths in ogbn-papers100M.

selected nodes. For example, with the (10,10,10,10,10) sampling parameter, the top 10% of the highest score nodes of the weighted reverse pagerank and the degree count functions account for 52% and 28% of the entire accesses, respectively. This experiment result shows that the benefit of data tiering is not immediately nullified with a growing sampling layer depth. The result opens the way for future GNN models that may attempt to sample deeper.

For the second analysis, we would like to more closely: (1) verify the hardware-level benefit of data tiering and (2) observe the impact of controlling the portion of data loaded to GPU. To achieve these, we sweep the portion of data loaded in GPU during GNN training and measure the volume of PCIe traffic and the training time (Figure 4.11). For this experiment, we perform data tiering with the weighted reverse pagerank function on ogbn-papers100M. To measure the PCIe traffic, we use the NVIDIA profiling tool *nvprof*. As we can see, our data placement and access strategy effectively reduce the PCIe traffic with more hot data loaded into the GPU memory. When we compare the cases with no data loading and 25% of data loading,

we can achieve about 97% of PCIe traffic reduction. At this point, most of the node feature accesses are resolved within GPU, and only very few data accesses need to be directed to the CPU memory over slow PCIe.

The performance gains in GNN training show a similar trend to the PCIe traffic reduction. With the 5% of data loaded, we can already reduce the training time by 33%, and with the 25% of data loaded, we can further reduce the training time by 42%. In general, the GPU memory consumed by the training process itself is proportional to the minibatch size, and the minibatch size is exponentially proportional to the sampling depth [106]. Therefore, hypothetically, if the sampling depth is very deep, the GPU memory available for hot data can be stolen by the training process memory requirement. However, because the default memory space complexity of GNN training is relatively low, the rapidly growing minibatch size is unlikely to have a realistic impact. For example, in the case of ogbn-papers100M training with 3-layer sampling, we consume only about 400 MB of GPU memory, and the rest of the space is left unused. Additionally, considering the trend of increasing the capacity of GPU memory (e.g., NVIDIA A100 80 GB) and the distributed Multi-GPU tensor solution we discussed in Section 4.3.2, we believe the actual impact is negligible.

### 4.4.5 Multi-GPU Training

In this section, we show the performance benefit of the multi-GPU implementation of our work described in Section 4.3.2. In this experiment, we use four V100 32 GB, and therefore we can have a total of 128 GB of collective GPU memory space. For the training dataset, we use MAG240M which has 350 GB of node feature tensor. For data placement, we divide the node feature tensor into two tensors, a multi-GPU tensor and a CPU tensor. We do not allocate any space for the replicated GPU tensor.

In Figure 4.12, we show the training time evaluation of MAG240M with increasing sizes of hot data loaded in the multi-GPU tensor. Before we go into further detail of the GPU sampling results, we first focus on the CPU sampling results. Similar to the results from Figure 4.11, we observe a sharp drop of training time with 5% of node features loaded into GPU memory. Beyond that, we observe only marginal performance improvements. The

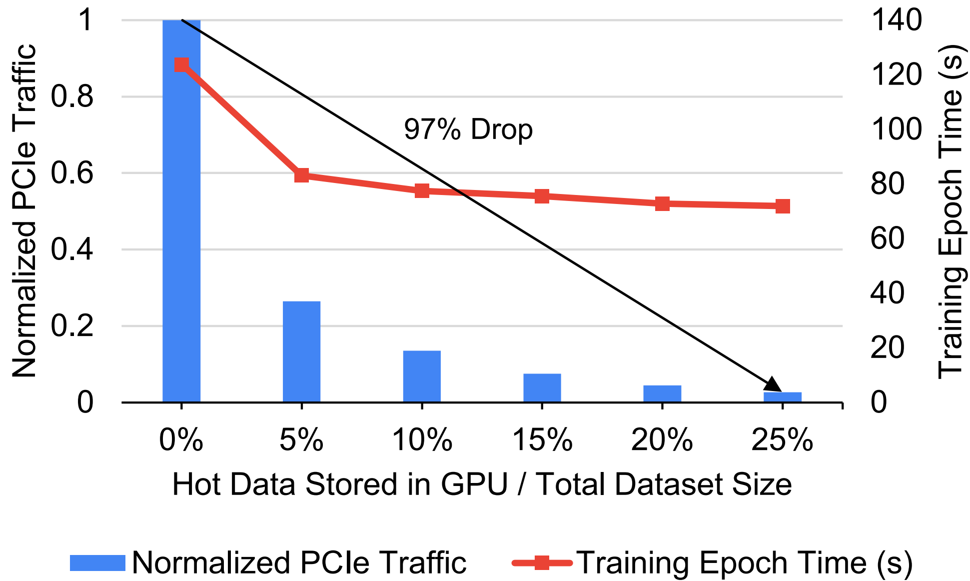


Figure 4.11: Single GPU PCIe traffic and training time comparison in actual GNN training with increasing hot data portion loaded in GPU. ogbn-papers100M dataset used.

overall performance improvements in multi-GPU training are underwhelming because the single GPU training of MAG240M in Figure 4.9 can reach 23.5 seconds already. This means, with four GPUs, we can reduce only about 3 seconds of training time further.

After performing several profilings, we find that in the multi-GPU training, the neighbor sampling process itself starts to throttle the whole training process and gives us poor scalability. As we described in Chapter 3 Section 3.2.2, the sampling step traverses the graph structure and generates the node IDs for a minibatch in preparation for the node feature aggregation step. We have been using the CPU for the neighbor sampling since the graph structure is stored in the CPU memory. In a single GPU training, the CPU could sample neighbors fast enough and provide their IDs to GPU in a reasonable amount of time. However, now in a multiple-GPU training, the number of minibatches that we need to generate is multiplied by the number of GPUs, which starts to affect the overall training time. Just to clarify, the implementation of the CPU sampling process is already done in a parallel fashion. In short, the amount of parallelism available from the CPU is not enough to quickly traverse the graph structure and sample neighbors for multiple GPUs.

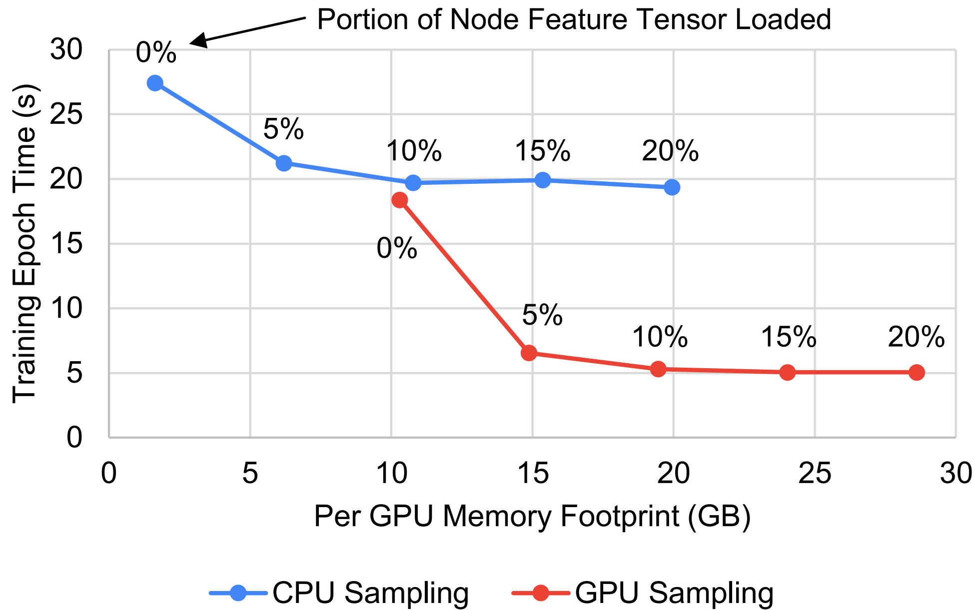


Figure 4.12: Multi-GPU MAG240M training time comparison while loading different amounts (in %) of node feature tensor into GPU memory. GPU sampling has a larger footprint since it has the adjacency matrix loaded in GPU memory.

This problem can be overcome with the GPU-based sampling method, but only with our multi-GPU data placement strategy. This is because to perform the GPU-based sampling, we now need to consider how to let GPUs access the graph structure as well. Of course, the simplest way of achieving this is to load the entire graph structure to each GPUs' memory, but the size of the graph structure of MAG240M alone is 30 GB, and it is too wasteful to load it into every GPU. To resolve this issue, we expand the idea of a multi-GPU node feature data placement strategy to the graph structure as well and distribute the graph structure over multiple GPUs. The benefit of the combination of our data placement strategy and the GPU sampling is shown in Figure 4.12. When we compare the memory footprints of the CPU sampling method and the GPU sampling method, we find that in general the GPU sampling chart has been shifted to the right because now the graph structure is consuming some GPU memory space. However, in terms of overall performance, the GPU sampling method removes the CPU sampling bottleneck and notably increases the training speed in the multi-GPU setup.

## 4.5 Conclusion

In this chapter, we presented a data tiering technique for GNN training. In general, we found the training time of GNN can be easily improved with well-defined data placement and rearrangement optimizations. Our data tiering strategy is a novel solution that does not affect the algorithm of GNN at all but still maximizes the benefit of the multi-tier memory subsystem of modern hardware. We empirically showed that our work can effectively reduce the data transfer time over the slow interconnect and improve the overall training time. We further demonstrated that our approach improves the scalability of multi-GPU training by eliminating the CPU bottleneck in both the sparse data access operation and the neighborhood sampling operation. We demonstrated our work by using existing libraries such as PyTorch and DGL, and showed that the end-users can immediately adopt our data tiering implementation in their programs.

# CHAPTER 5

# CONCLUSION

The increasing use of large sparse datasets with challenging memory access behaviors limits accelerators' effectiveness from multiple directions. Such memory access behaviors are often deeply associated with the application characteristics, and it is challenging to tackle the problem with software-only approaches. In this dissertation, we started solving the problem at the I/O-level and proposed using fine-grained memory access over I/O.

Chapter 2 takes graph analytics, such as breadth-first-search and PageRank, as examples and analyzes the impact of using the fine-grained memory access over I/O. The I/O-level analysis using FPGA shows that overcoming the limited number of outstanding requests over I/O is the key to achieving high effective bandwidth. This chapter demonstrates two optimization techniques, opportunistic packet merging and address alignment, in GPU implementation to overcome such limitations. We also show that the use of fine-grained memory access over I/O has a massive advantage over the previous algorithm-only solutions, which still incur the penalty of frequent CPU-GPU synchronization for small data accesses.

Chapter 3 discusses how the new data access method can be integrated into the existing machine learning frameworks such as PyTorch. The existing frameworks solely focus on using the traditional block data transfer method, and the modification is exceptionally rigid. In our work, we propose a new data class called Unified Tensor to accommodate newer data access methods and provide a flexible interface to both the users and the framework developers. The use of Unified Tensor simplifies the adoption of the fine-grained memory access over I/O at the framework level and requires minimum algorithm changes at the user program level.

Finally, in Chapter 4, we show other data transfer optimization techniques in multi-level memory hierarchy systems using the fine-grained memory access over I/O. The multi-level memory hierarchy system provides different

tiers of memory devices, and it is difficult to efficiently access sparse datasets spread over different memory devices due to the irregular memory access pattern. In our implementation, by allowing the finer grain of memory access to different tiers of memory directly, we can avoid the memory access complexity existing in the previous work and significantly increase the data access efficiency.

# REFERENCES


[1] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[2] J. Webber, “A programmatic introduction to Neo4j,” in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2384716.2384777 p. 217–218.

[3] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, “Traversing large graphs on GPUs with unified memory,” *Proceedings of the VLDB Endowment*, vol. 13, no. 7, p. 1119–1133, Mar. 2020.

[4] A. H. N. Sabet, Z. Zhao, and R. Gupta, “Subway: Minimizing data transfer during out-of-GPU-memory graph processing,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020.

[5] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, “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)*, 2020.

[6] A. Kalia, M. Kaminsky, and D. G. Andersen, “Design guidelines for high performance RDMA systems,” in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia pp. 437–450.

[7] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, “Deep graph library: A graph-centric, highly-performant package for graph neural networks,” *arXiv preprint arXiv:1909.01315*, 2019.

[8] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl, "Pump up the volume: Processing large data on gpus with fast interconnects," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3318464.3389705 p. 1633–1649.

[9] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, p. 420–431, Dec. 2017.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[11] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[12] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[13] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012. [Online]. Available: http://arxiv.org/abs/1205.6233

[14] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[15] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[16] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th International Conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[17] M. Harris, "Unified memory for CUDA beginners," https://devblogs.nvidia.com/unified-memory-cuda-beginners/, 6 2017.

[18] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "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*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 49–63.

[19] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "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*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1357–1370.

[20] V. S. Mailthody, K. Date, Z. Qureshi, C. Pearson, R. Nagi, J. Xiong, and W.-m. Hwu, "Collaborative (CPU+GPU) algorithms for triangle counting and truss decomposition," in *2018 IEEE High Performance Extreme Computing Conference (HPEC'18)*, Boston, USA, 2018.

[21] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page placement strategies for gpus within heterogeneous memory systems," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, p. 607–618, Mar. 2015.

[22] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "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*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 136–150.

[23] J. Kehne, J. Metter, and F. Bellosa, "GPUswap: Enabling oversubscription of GPU memory through transparent swapping," *SIGPLAN Not.*, vol. 50, no. 7, p. 65–77, Mar. 2015.

[24] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "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*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016.

[25] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan, "Graphreduce: Processing large-scale graphs on accelerator-based systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[26] W. Han, D. Mawhirter, B. Wu, and M. Buland, "Graphie: Large-scale asynchronous graph traversals on just a GPU," in *Proceedings of 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 233–245.

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

[28] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 197–208.

[29] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with GPUs and CUDA," *Parallel Comput.*, vol. 36, no. 12, pp. 655–678, Dec. 2010. [Online]. Available: https://doi.org/10.1016/j.parco.2010.07.002

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

[31] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 267–276.

[32] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 479–490.

[33] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Transactions on Parallel and Distribution Systems*, vol. 25, no. 6, p. 1543–1552, June 2014.

[34] "nvGRAPH," https://developer.nvidia.com/nvgraph.

[35] "Nvidia Tesla P100 architecture whitepaper," https://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[36] "Nvidia Tesla V100 GPU architecture whitepaper," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[37] C. Pearson, M. Almasri, O. Anjum, V. S. Mailthody, Z. Qureshi, R. Nagi, J. Xiong, and W.-m. Hwu, "Update on triangle counting on GPU," in *2019 IEEE High Performance extreme Computing Conference (HPEC'19)*, Boston, USA, 2019.

[38] "CUDA C++ best practices guide," https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, 2020.

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

[40] "PCIe 3.0 specification," https://members.pcisig.com/wg/PCI-SIG/document/download/8257, 2020.

[41] "Nvidia A100 GPU architecture whitepaper," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[42] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[43] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015.

[44] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 267–276.

[45] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang, "GARDENIA: A domain-specific benchmark suite for next-generation accelerators," *CoRR*, vol. abs/1708.04567, 2017. [Online]. Available: http://arxiv.org/abs/1708.04567

[46] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2517349.2522739 p. 456–471.

[47] J. Sybrandt, M. Shtutman, and I. Safro, "Moliere: Automatic biomedical hypothesis generation system," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1633–1642.

[48] T. K. Aasawat, T. Reza, and M. Ripeanu, "How well do CPU, GPU and hybrid graph processing frameworks perform?" in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 458–466.

[49] "Nvidia DGX A100 datasheet," https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf, 2020.

[50] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, p. 135–146, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2517327.2442530

[51] M. Sha, Y. Li, B. He, and K.-L. Tan, "Accelerating dynamic graph analytics on GPUs," *Proc. VLDB Endow.*, vol. 11, no. 1, p. 107–120, Sep. 2017. [Online]. Available: https://doi.org/10.14778/3151113.3151122

[52] M. Sha, Y. Li, and K.-L. Tan, "GPU-based graph traversal on compressed graphs," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3299869.3319871 p. 775–792.

[53] K. Kaczmarski, P. Przymus, and P. Rzążewski, "Improving high-performance GPU graph traversal with compression," in *New Trends in Database and Information Systems II*. Springer, 2015, pp. 201–214.

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

[55] K. Lakhotia, S. Singapura, R. Kannan, and V. Prasanna, "Recall: Reordered cache aware locality based graph processing," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 273–282.

[56] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[57] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[58] N. Strom, "Scalable distributed DNN training using commodity GPU cloud computing," in *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[59] Y. Lin, S. Han, H. Mao, Y. Wang, and B. Dally, "Deep gradient compression: Reducing the communication bandwidth for distributed training," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=SkhQHMW0W

[60] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic GPU memory management for training deep neural networks," *SIGPLAN Not.*, vol. 53, no. 1, p. 41–53, Feb. 2018. [Online]. Available: https://doi.org/10.1145/3200691.3178491

[61] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[62] S. W. Min, V. S. Mailthody, Z. Qureshi, J. Xiong, E. Ebrahimi, and W.-m. Hwu, "EMOGI: Efficient memory-access for out-of-memory graph-traversal in GPUs," *arXiv preprint arXiv:2006.06890*, 2020.

[63] NVIDIA, "Multi-process service," 2020. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[64] J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun, "Spectral networks and locally connected networks on graphs," in *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*, 2014.

[65] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[66] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3219819.3219890 p. 974–983.

[67] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Red Hook, NY, USA: Curran Associates Inc., 2016, p. 3844–3852.

[68] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *NIPS Workshop on Bayesian Deep Learning*, 2016.

[69] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016. [Online]. Available: http://proceedings.mlr.press/v48/niepert16.html pp. 2014–2023.

[70] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[71] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *IEEE Data Eng. Bull.*, vol. 40, no. 3, pp. 52–74, 2017. [Online]. Available: http://sites.computer.org/debull/A17sept/p52.pdf

[72] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2939672.2939754 p. 855–864.

[73] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2623330.2623732 p. 701–710.

[74] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rytstxWAW

[75] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. Stockholmsmässan, Stockholm Sweden: PMLR, 10–15 Jul 2018. [Online]. Available: http://proceedings.mlr.press/v80/chen18p.html pp. 942–950.

[76] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 630–645.

[77] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[78] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[79] T. Schroeder, "Peer-to-peer & unified virtual addressing," 2011. [Online]. Available: https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf

[80] C. Pearson, A. Dakkak, S. Hashash, C. Li, I.-H. Chung, J. Xiong, and W.-M. Hwu, "Evaluating characteristics of CUDA communication primitives on high-bandwidth interconnects," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3297663.3310299 p. 209–218.

[81] M. Harris, "How to access global memory efficiently in CUDA C/C++ kernels," 2013. [Online]. Available: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[82] NVIDIA, "Kernel profiling guide," 2021. [Online]. Available: https://docs.nvidia.com/nsight-compute/pdf/ProfilingGuide.pdf

[83] V. Volkov, "Understanding latency hiding on GPUs," Ph.D. dissertation, UC Berkeley, 2016.

[84] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, "Understanding PCIe performance for end host networking," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3230543.3230560 p. 327–341.

[85] PyTorch, "TorchElastic," 2021. [Online]. Available: https://pytorch.org/elastic/0.2.2/index.html

[86] J. Kunegis, "KONECT: The Koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*, ser. WWW '13 Companion. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2487788.2488173 p. 1343–1350.

[87] J. Ni, J. Li, and J. McAuley, "Justifying recommendations using distantly-labeled reviews and fine-grained aspects," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019. [Online]. Available: https://www.aclweb.org/anthology/D19-1018 pp. 188–197.

[88] K. Wang, I. Shen, C. Huang, C.-H. Wu, Y. Dong, and A. Kanakia, "Microsoft academic graph: When experts are not enough," *Quantitative Science Studies*, vol. 1, no. 1, pp. 396–413, February 2020, https://doi.org/10.1162/qss_a_00021.

[89] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[90] K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma, "The extreme classification repository: Multi-label datasets and code," 2016. [Online]. Available: http://manikvarma.org/downloads/XC/XMLRepository.html

[91] NVIDIA, "Nvidia multi-instance GPU userguide," 2020. [Online]. Available: https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf

[92] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," *CoRR*, vol. abs/1906.00091, 2019. [Online]. Available: https://arxiv.org/abs/1906.00091

[93] M. Smelyanskiy, "Zion: Facebook next-generation large memory training platform," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–22.

[94] S. Krishna and R. Krishna, "Accelerating recommender systems via hardware "scale-in"," *CoRR*, vol. abs/2009.05230, 2020. [Online]. Available: https://arxiv.org/abs/2009.05230

[95] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi, "A tensor compiler for unified machine learning prediction serving," in *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, November 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-tensor-compiler-for-unified-machine-learning-prediction-serving/ pp. 899–917.

[96] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating spark with RDMA for big data processing: Early experiences," in *2014 IEEE 22nd Annual Symposium on High-Performance Interconnects*, 2014, pp. 9–16.

[97] Z. Bar-Yossef and L.-T. Mashiach, "Local approximation of pagerank and reverse pagerank," in *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, ser. CIKM '08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: https://doi.org/10.1145/1458082.1458122 p. 279–288.

[98] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling GNN training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3419111.3421281 p. 401–415.

[99] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "AliGraph: A comprehensive graph neural network platform," *arXiv preprint arXiv:1902.08730*, 2019.

[100] S. W. Min, K. Wu, S. Huang, M. Hidayetoglu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, "Large graph convolutional network training with GPU-oriented data communication architecture," *Proc. VLDB Endow.*, vol. 14, no. 11, p. 2087–2100, Oct. 2021.

[101] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with Roc," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf pp. 187–198.

[102] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "NeuGraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/ma pp. 443–458.

[103] Y. Shi, Z. Huang, W. Wang, H. Zhong, S. Feng, and Y. Sun, "Masked label prediction: Unified message passing model for semi-supervised classification," *CoRR*, vol. abs/2009.03509, 2020. [Online]. Available: https://arxiv.org/abs/2009.03509

[104] M. Niu, "ogbn-papers100m-sage," https://github.com/mengyangniu/ogbn-papers100m-sage, 2021.

[105] Y. Shi, Z. Huang, W. Li, W. Su, and S. Feng, "R-UNIMP: Solution for KDDCup 2021 MAG240M-LSC," 2021.

[106] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019. [Online]. Available: http://arxiv.org/abs/1901.00596