Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture

Seung Won Min
UIUC
min16@illinois.edu

Kun Wu
UIUC
kunwu2@illinois.edu

Sitao Huang
UIUC
shuang91@illinois.edu

Mert Hidayetoğlu
UIUC
hidayet2@illinois.edu

Jinjun Xiong
IBM T.J. Watson Research Center
jinjun@us.ibm.com

Eiman Ebrahimi
NVIDIA
eebrahimi@nvidia.com

Deming Chen
UIUC
dchen@illinois.edu

Wen-mei Hwu
NVIDIA / UIUC
whwu@nvidia.com

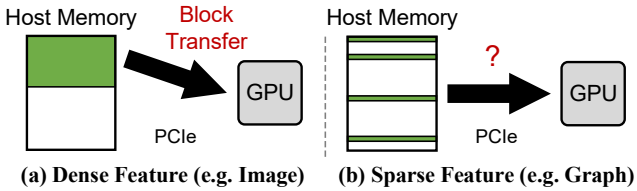
ABSTRACT

Graph Convolutional Networks (GCNs) are increasingly adopted in large-scale graph-based recommender systems. Training GCN requires the minibatch generator traversing graphs and sampling the sparsely located neighboring nodes to obtain their features. Since real-world graphs often exceed the capacity of GPU memory, current GCN training systems keep the feature table in host memory and rely on the CPU to collect sparse features before sending them to the GPUs. This approach, however, puts tremendous pressure on host memory bandwidth and the CPU. This is because the CPU needs to (1) read sparse features from memory, (2) write features into memory as a dense format, and (3) transfer the features from memory to the GPUs.

In this work, we propose a novel GPU-oriented data communication approach for GCN training, where GPU threads directly access sparse features in host memory through zero-copy accesses without much CPU help. By removing the CPU gathering stage, our method significantly reduces the consumption of the host resources and data access latency. We further present two important techniques to achieve high host memory access efficiency by the GPU: (1) automatic data access address alignment to maximize PCIe packet efficiency, and (2) asynchronous zero-copy access and kernel execution to fully overlap data transfer with training. We incorporate our method into PyTorch and evaluate its effectiveness using several graphs with sizes up to 111 million nodes and 1.6 billion edges. In a multi-GPU training setup, our method is 65-92% faster than the conventional data transfer method, and can even match the performance of all-in-GPU-memory training for some graphs that fit in GPU memory.

PVLDB Reference Format:
Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. PVLDB, 14(11): 2087 - 2100, 2021.
doi:10.14778/3476249.3476264

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.



(a) Dense Feature (e.g. Image) **(b) Sparse Feature (e.g. Graph)**

Figure 1: Challenge of GPUs accessing fine-grained sparse features in host memory.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/K-Wu/pytorch-direct_dgl.

1 INTRODUCTION

Acceleration of modern machine learning models is often severely limited by insufficient memory bandwidth [26, 49, 50]. To provide the best possible memory bandwidth, data is usually placed in memory closest to the processing units of the accelerators [45, 54]. 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. To free processing units from spending excessive amount of time accessing remote data, modern hardware systems utilize direct memory access (DMA) engines.

DMA engines are specialized in transferring large blocks of data independently. 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 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

doi:10.14778/3476249.3476264

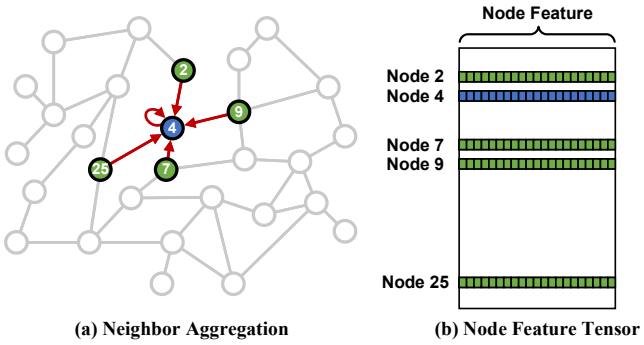


Figure 2: (a) A simple example of GCN training on single node. (b) An illustration of node features in memory. The neighboring nodes' features are scattered in memory.

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 1 (b)).

One of the most successful adaptations of deep neural network models to graph data is Graph Convolutional Network (GCN) [22]. The core idea of GCN is to create node embeddings by iteratively aggregating neighboring nodes' attributes using neural networks. Due to its neighboring node's attribute lookup, training GCN requires accessing multiple scattered locations in memory. In Figure 2 (a), we show a simple example of GCN 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 work, 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 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, recent work has 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 [28]. Therefore, in GCN training, if GPUs can make targeted fine-grain host memory accesses for sparse features while fully utilizing system interconnect (e.g., PCIe) bandwidth, the proposed approach can offer significant advantage over the DMA approach. The removal of CPU gathering stage not only shortens data access latency for GPUs, but also greatly reduces the CPU and host memory utilization (Figure 3). Offloading CPU workloads to GPUs also helps on training

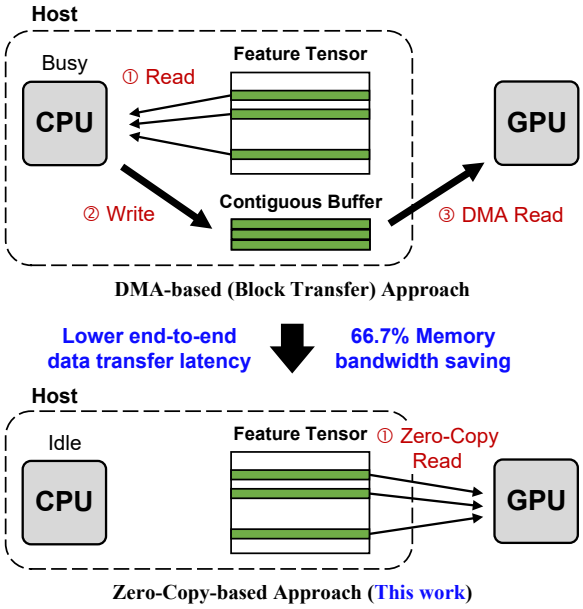


Figure 3: Workload comparison between DMA-based method and the proposed zero-copy-based method.

GCN with multiple GPUs as we can prevent the CPU becoming the bottleneck with increasing number of workers.

In order to propose the GPU-oriented data communication architecture for GCN training, we address three major questions in this work. First, can zero-copy memory access fully utilize PCIe bandwidth while training GCN 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 work, we answer all three questions. First, to maintain the best possible PCIe packet efficiency with zero-copy memory access, we propose an automatic data access alignment optimization in GPU data indexing kernel. With our optimization, zero-copy PCIe bandwidth can match up to 93% of block transfer PCIe bandwidth. Second, we propose a novel CUDA multi-process service (MPS) [37] based resource provisioning optimization to minimize GPU resource consumption of zero-copy memory accesses. Based on careful investigation of PCIe protocol and GPU architecture, we conclude that we can saturate PCIe even if only a few number of GPU cores are generating zero-copy accesses. Therefore, our optimization isolates only small portion of GPU resources for the zero-copy accesses and leaves the rest for computationally intense workloads.

Finally, we build an end-to-end zero-copy GCN 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 GCN training flow can also support zero-copy access in multi-GPU training environment.

Our modifications are seamlessly integrated with the existing PyTorch framework and therefore we can quickly apply our method on existing GCN training applications. We evaluate our design on multiple large graph datasets where the largest one has 111 millions of nodes and 1.6 billions of edges. In a single-GPU training environment, our method is 16-44% faster than the DMA-based method, but in a multi-GPU training environment, our method becomes 65-92% faster than the DMA-based method. 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 paper 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 GCN training.
- To improve the efficiency of zero-copy memory access, we propose automatic data alignment and a 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 paper is organized into the following sections. Section 2 provides the necessary background for the proposed approach. Section 3 gives a brief overview of the proposed approach. Section 4 presents an experimental evaluation of the proposed approach. Section 5 discusses potential future work. Section 6 presents related works. Section 7 offers concluding remarks.

2 BACKGROUND

2.1 Graph Convolutional Network

The idea of Graph convolutional networks (GCN) [4, 8, 14, 21, 22, 34, 57] started by an attempt to apply filters similar to convolutional neural networks (CNNs) [25] on graph structures. Bruna et al. [4] was the first to propose the GCN model, where the authors utilized Laplacian filters as hidden layers to exploit global structure of the graph. Such spectral construction is later adopted by many GCNs, including [8, 22].

GCNs are widely adopted in graph representation learning [15], where GCN 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 [13] and DeepWalk [43], are inherently shallow, transductive, and do not share parameters or utilize node attributes to encode node [15]. These limits the representation power of the model, and disables the model to infer the representation when the nodes or edges are unseen in training. GCN opens up the potential to develop algorithms to tackle these problems [14, 57].

One severe issue with the early GCN is that the Laplacian filters in each layer are matrices whose dimension increases as the number of nodes in graph increases. This effectively throttles the depth of GCN and the size of graph it can be applied due to the large memory footprint. As an example, Kipf et al. [22] presents a model for semi-supervised node classification using GCN. 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. $H^{(0)}$ is the input node feature table. Here, we can see the memory requirement of the operation is directly related to the size of N.

2.2 Neighborhood Sampling

To tackle the limitation of GCN, GraphSAGE [14] introduces neighborhood sampling and aggregating approach. By sampling 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 mini-batching 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 uniformly random selection process to sample the neighboring nodes, but other works such as FastGCN [5], VR-GCN [6] 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 $(S_1, S_2) = (10, 25)$ 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 on the number of nodes that need to be sampled. Such lower depth of network layers compared to other deeper neural networks [18, 51] makes the optimization of data transfer time critical in GCN training. After the sampling, a sub-graph which only contains 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 to cover more corner cases. The exact implementation of the sampling process is framework-dependant. In case of deep graph library (DGL) [55], this part is written in C++ with OpenMP to maximize the performance, but PyTorch-Geometric [9] simply uses a python code.

If the entire node feature table is not fitting in the GPU memory, the sampled nodes' features must be transferred after each sampling step [57]. Since the sampled nodes' features are scattered over the feature table, the current GCN 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.

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 [47].

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, DMA engine is used for the data transfer. DMA engine is efficient in transferring

a single large data block, but sub-optimal for transferring small sized data due to the DMA request setup latency caused by user program <-> operating system interactions. According to Pearson et al. [42], to make the effective bandwidth of DMA to about 90% of maximum PCIe 3.0 x16 bandwidth, the data block size should be at least 256KB. With 64KB of 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) [17, 35, 36, 38, 42]. 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 processor accesses this page later on, page migration to that processor will be triggered. The minimum migration granularity is identical to the system page size (4KB), but it can be as large as 2MB. UVM makes programming easier by removing the need of explicit call 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 mechanism of data transfer. 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 larger discrepancy between the dataset size and the GPU memory size, there will be more frequent page migrations incurred by severe page thrashing.

Finally, CUDA enables zero-copy access, which is also known as direct access. In a zero-copy access, GPU sends a cacheline-sized memory request directly through external interconnect (e.g. PCIe), without explicit data copy or page migration that will happen in the aforementioned two methods. 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 GPU-ORIENTED DATA COMMUNICATION ARCHITECTURE

Due to the wide spread use of DMA-based data communication architecture, there are some number of system-level modifications that must to 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 GCN training flow using zero-copy accesses.

3.1 Zero-Copy Enablement in PyTorch

For GCN training, we use PyTorch which is 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 such issue, we create an extension of the existing PyTorch implementation with several modifications in its source code.

Table 1: PyTorch Tensor Class Comparison.

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

Listing 1: PyTorch Programming with Unified Tensor

```
1  import torch
2
3  # Input tensor data in host memory
4  input_tensor = torch.randn([100], device="cpu")
5
6  # CUDA tensor created, data copied by DMA (e.g. cudaMemcpy())
7  gpu_tensor = input_tensor.to(device="cuda")
8
9  # Unified tensor created, no data copy occurs
10 unified_tensor = input_tensor.to(device="unified")
11
12 # gpu_tensor data comes from GPU memory
13 # unified_tensor accessed through zero-copy access to host memory
14 # Computation done by GPU
15 output = gpu_tensor + unified_tensor
```

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 1). In the current implementation of PyTorch, processing units can only compute data located within their own local memories. For example, to perform a GPU-accelerated matrix multiplication on CPU tensor, a new tensor with CUDA context should be created. 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 which 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 limitation for multi-GPU training, which we will explain in Section 3.4.

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 1 shows a simple vector addition example in PyTorch using 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

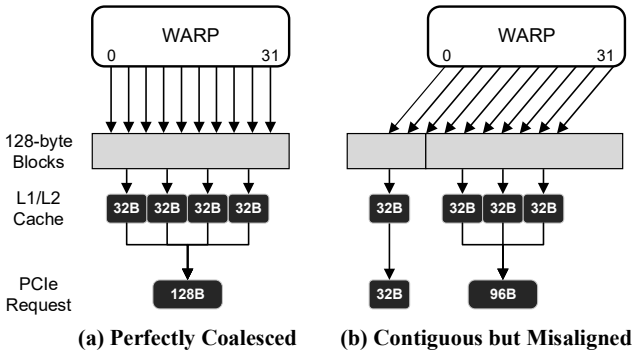


Figure 4: (a) A perfectly coalesced 128-byte access from a warp. (b) A warp accessing a misaligned data needs to generate multiple PCIe requests.

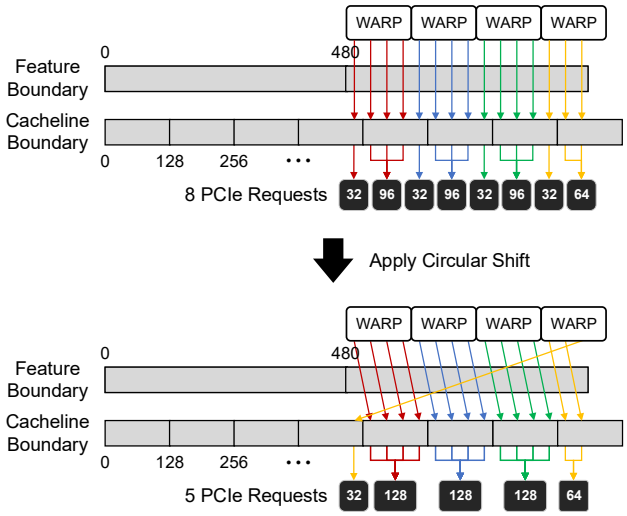


Figure 5: Circular shift optimization explained. Circular shift transforms memory requests into a GPU cacheline-friendly way.

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.

3.2 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 [11]. 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 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 GCN training.

3.2.1 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, to compensate the overhead, it is better to increase the payload size by requesting a larger memory request. 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), it is necessary to submit multiple read requests in a pipelined fashion to fully occupy the interconnect. However, if we squander the capacity by generating too many small read requests, it becomes difficult to fully tolerate the latency and utilize the PCIe bandwidth. The numbers of maximum outstanding read requests for PCIe 3.0 and PCIe 4.0 are 256 and 768, respectively.

Now, with all that in mind, how do we generate coarser-grained PCIe requests? According to Min et al. [28], to make PCIe read requests more efficient, the same technique used for the GPU memory coalescing [16] can be used. In Figure 4, we explain 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 case of (a), the accesses from the threads in a warp are perfectly coalesced and the coalesced requests becomes a single 128B PCIe read request. In 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 32B, 64B, 96B, and 128B, while 32B is a single sector size of GPU cacheline [40]. Each GPU cacheline is composed of four sectors.

Of course, we would not need to worry about the misaligned accesses if the node feature objects always start at 128B boundaries and the sizes of node features are always multiples of 128B, but it is very unlikely to be so in reality. For example, if a certain dataset's node feature size is 480B, accessing the second node feature will start from accessing 480th byte in memory address. In this case, we are off by 32B from the closest 128B boundary (512B). To automatically resolve such issue, we add a circular shift stage in the PyTorch indexing CUDA kernel. In Listing 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 128B aligned location and the current indexing location. The visualization of our circular shift mechanism is shown in Figure 5. In this example, we want to access the second node feature with zero-copy access where each node feature size is 480B. Without the optimization, each warp start reading from misaligned locations and end 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 as a result of our modifications to PyTorch source code.

Listing 2: GPU Indexing Kernel and Automatic Alignement

```
#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];
  }
}
```

3.2.2 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 all it's doing is simply reading the host memory. However, like in many other ML algorithms, GCN 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 the training GPU kernels and the zero-copy GPU kernels in our design.

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 the concurrency is impossible. First, there are several blocking CUDA APIs such as `cudaMalloc()`, `cudaFree()`, and `cudaEventQuery()` that serialize the GPU operations. In current implementation of PyTorch, some of the listed APIs are called in background implicitly, such as by memory allocation manager. If one of the CUDA APIs are called in between the zero-copy GPU kernel and the training GPU kernel, the latter GPU kernel must wait until the entire operation of the earlier GPU kernel to be finished. Second, when the GPU resources are completely consumed by a current GPU kernel, 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 they can and the serialization situation is very likely to occur.

Table 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

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

To answer to this question, we explore the architecture details of NVIDIA GPUs. In NVIDIA GPUs, to better utilize computation units and to hide long GPU memory access latency, each single physical core may have multiple active warps to issue instructions from [52]. In this way, the physical core won't 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 the previous Section 3.2.1, 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 small amount of GPU resources to fully saturate the limit, it is worthwhile to seek for a way to achieve the concurrency. In Table 2, we list the specifications of NVIDIA RTX 3090 GPU which we use for our evaluations. At any given moment, the GPU cannot 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, lets assume each warp's memory requests are coalesced to a single PCIe read request, and lets 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 much fewer outstanding requests. For example, the measured maximum PCIe bandwidth with `cudaMemcpy()` in RTX 3090 is 25.8GB/s and if we assume RTT (Round-Trip-Time) of PCIe is 1.5us [32], the number of outstanding requests that we need to sustain is (25.8GB/s) / (128B) $\times$ 1.5us = 324.6. That is, assuming all PCIe requests are 128B in

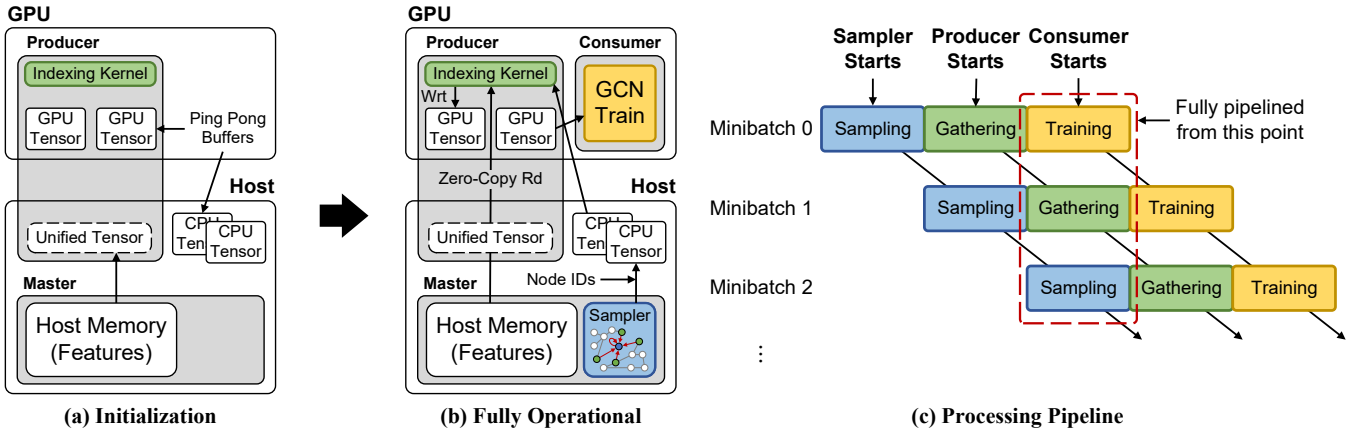


Figure 6: GCN training flow with zero-copy accesses. Only the operations related to data accesses are shown. (a) We setup unified tensor and the returned pointer is passed to GPU for zero-copy accesses. (b) The sampler generates node IDs used by the producer and the producer gathers scattered node features in the host memory. The consumer uses the gathered node features for training. (c) A visualization of processing pipeline.

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 is at least 80% and up to 91.8% of the GPU resources available for other workloads.

Now, finally, since we realized how much of 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) [37]. MPS is originally designed to improve 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 processes. Since PyTorch already supports multiprocessing programming model, it is simple to launch the zero-copy GPU kernel and the training GPU kernel in two separate processes. Before we launch the zero-copy GPU kernel, we modify the GPU resource limitation to $X\%$ with the `nvidia-cuda-mps-control` utility. Next, before we launch the training GPU kernel, we also modify the resource limitation to $(100 - X)\%$. In our PyTorch code, the whole process is scripted for an easier use. It would be more elegant if the resource limitation can be configured in the user CUDA code instead of the MPS utility, but currently CUDA does not support such functionality. 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 saturate the PCIe bandwidth while leaving majority of GPU resources opened for other computationally intensive workloads.

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

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 GCN training with zero-copy accesses enabled. In Figure 6 (a), we show the initial tensor allocations during the initialization step. First, we map the whole node feature tensor into the GPU address space by using the unified tensor. This unified tensor holds a memory pointer which 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 to 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 6 (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 directly 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

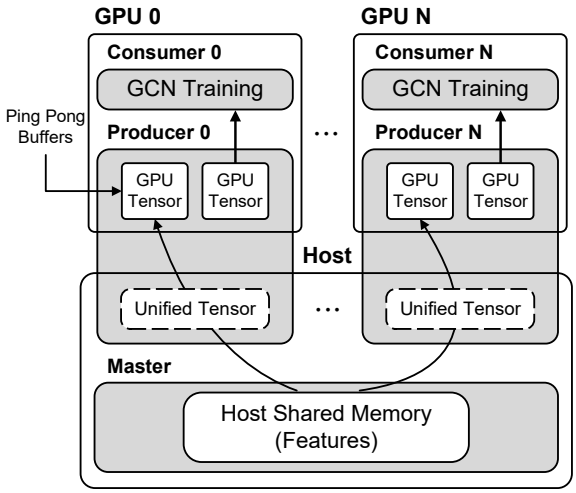


Figure 7: Simplified view of Multi-GPU GCN training flow. All unified tensors provide identical mappings. Sampling processes and indexing kernels are omitted in this diagram.

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 → producer → consumer order (Figure 6 (c)). Except the unified tensor declaration, the rest of our end-to-end GCN training implementations is developed with the existing PyTorch functionalities, and this makes 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 [44] framework, which allow users to recover from failure or to steadily train even with faulty hardware. 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 GCN training and the training algorithms are unaffected.

3.4 Multi-GPU Training

The final challenge of our method is supporting 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 infeasible to propose a method which can only support a single-GPU training environment.

For multi-GPU training, we take the data parallelism approach used in DGL [55]. 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 7.

Listing 3: Unified Tensor Declaration in Multiprocessing Environment

```
1  import torch
2  import torch.multiprocessing as mp
3
4  def producer(features, process_id, ...):
5      # Specify target GPU ID
6      torch.cuda.set_device(process_id)
7      # Map host shared memory to GPU address space
8      features = features.to(device="unified")
9      ...
10
11 if __name__ == '__main__':
12     features = torch.randn([100], device="cpu")
13     # Allocate shared memory space
14     features = features.share_memory_()
15     ...
16     # Pass feature tensor alloctead in shared memory space
17     # and call producers for multiple GPUs
18     producer1 = ctx.Process(target=producer,
19                             args=(features, 0, ...))
20     producer2 = ctx.Process(target=producer,
21                             args=(features, 1, ...))
22     ...
```

The main difficulty of multi-GPU training with zero-copy accesses lies on 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 the way 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 an extremely inefficient usage of host memory capacity.

Therefore, in our implementation, we take an opposite direction of memory allocation. Instead of attempting to share a memory space after allocating with CUDA APIs, we first allocate a shareable memory space and then call CUDA APIs to allow GPUs access the space. In Linux, to allow multiple processes to share a 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 becasue it can be used on top of the Linux shared memory. Therefore, by letting different processes to 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. 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 selecting the correct CUDA device (e.g. producer0 → GPU0, producer1 → 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).

4 EVALUATION

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

4.1 Methodology

4.1.1 Evaluation System. For our evaluation, we use the system described in Table 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 25.8GB/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 51.7GB/s.

4.1.2 Application. Our unified tensor implementation and the indexing kernel modification are based on PyTorch 1.8.0-nightly version. For the GCN training, we use the GraphSAGE [14] implementation of DGL [55]. 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 GCN. In this case, there is no need of 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 do not include any optimizations we discussed in this paper. 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 paper. 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 as "out-of-memory (OOM)" for such cases.

4.1.3 Dataset. In Table 4, we show the datasets we used for the evaluation. `wikipedia` [24] network consists of the wikilinks of the Wikipedia in the English language. Nodes are Wikipedia articles, and directed edges are wikilinks. `amazon` [33] dataset 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 [53]. The above datasets are used for basic performance evaluations. `ogbn-products` [19] dataset is based on Amazon co-purchasing network [3] 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 4.4.2.

Table 3: Evaluation system configuration.

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

Table 4: Evaluation Dataset.

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

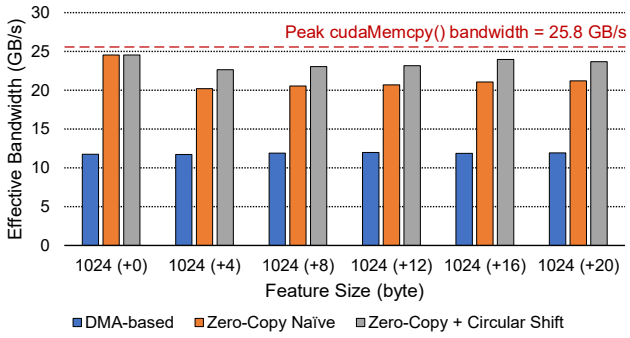
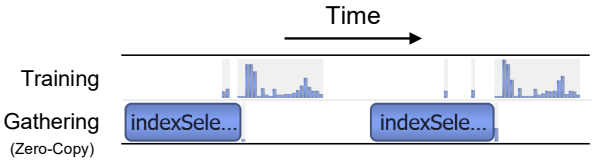


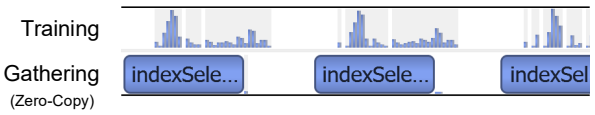
Figure 8: Effective data transfer bandwidth measured during the wikipedia dataset training. We sweep feature size to observe the impact of misaligned zero-copy accesses over PCIe.

4.2 Bandwidth Analysis

In Figure 8, 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 1024B to 1044B in this experiment. Zero-copy naïve approach does not implement the circular shift optimization we discussed in Section 3.2.1. Throughout the experiment, the effective bandwidth of the DMA-based approach is



(a) Naïve Zero-Copy Implementation



(b) With the Proposed MPS-based QoS

Figure 9: Snapshots of NVIDIA Nsight Systems Profiler during GCN training. CPU workloads not shown here.

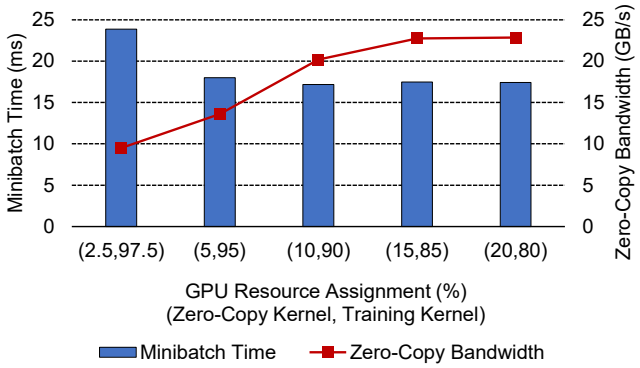


Figure 10: MPS resource partitioning ratio sensitivity analysis.

only about half of the zero-copy approaches as it requires a long CPU gathering process.

When the node feature size is 1024B, regardless of the circular shift optimization existence, the zero-copy implementations show the best effective bandwidth numbers. Because the GPU cacheline size is 128B, in this case accessing any node features results in generating perfectly coalesced accesses. Considering that the best `cudaMemcpy()` bandwidth we achieved is about 25.8GB/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.

4.3 Concurrency Analysis

The best way to check if our MPS-based resource provisioning is helping the concurrency is profiling the workload and visually inspecting the GPU kernel timeline. In Figure 9, we show two profiling results of GCN 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 3.2.2, in fact only a limited number of cores can actually submit memory requests over PCIe due to the protocol limitation. 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 caused from it. In Figure 10, we show the zero-copy PCIe bandwidth change over allocating different amount of GPU resource to the zero-copy kernel. With 2.5–10% of resource allocations, the zero-copy kernel cannot generate enough number of PCIe read requests and therefore the measured PCIe bandwidth is limited to 9.5–20.2GB/s. Further increasing the GPU resource allocation can make the zero-copy bandwidth to reach around 23.0GB/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 in to the estimation we made in Section 3.2.2. 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 worth 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.

4.4 Training Performance

4.4.1 Overall Comparison. In Figure 11, 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

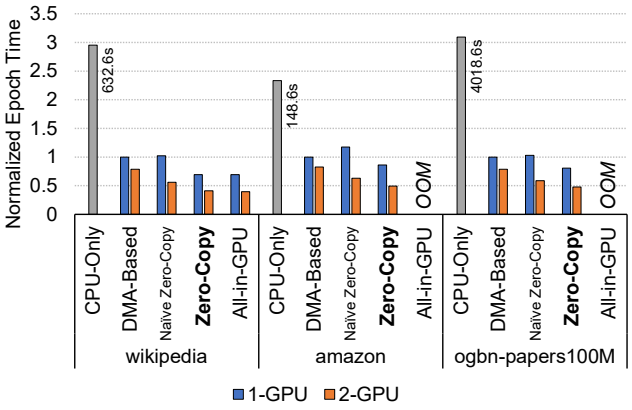


Figure 11: GCN training time comparison. OOM denotes out-of-memory.

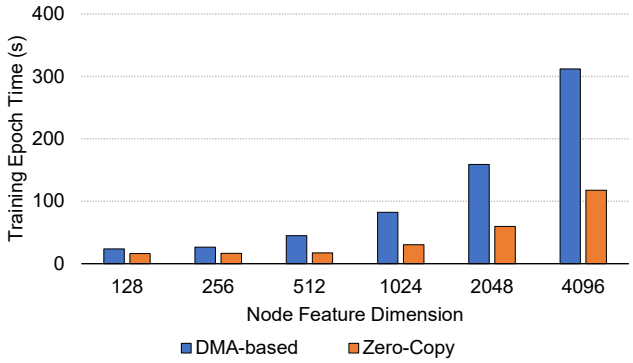


Figure 12: Node feature size sensitivity analysis.

data transfers over PCIe but at the same time the computation power is severely limited. In general, the CPU-only method is 2.3–3.1× slower than 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 reducing 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 GPU quickly makes the entire training process to be throttled. In fact, the host memory overutilization exists even with a single GPU. In the DMA-based method, each DMA transfer is preceded by two memory accesses as shown in Figure 3. This means, to sustain a single PCIe 4.0 x16 bandwidth (25.8GB/s), additional 51.6GB/s of host memory bandwidth is wasted by the CPU gathering process. Based on our measurement, we find our machine can provide about 59GB/s of host memory bandwidth with real workloads. Therefore, the sum of bandwidth requirement (77.4GB/s) well goes beyond the available host memory bandwidth.

For the naïve zero-copy method, we observe 2–17% of performance degrade compared to the DMA-based method in a single-GPU setup. This degradation is consistent with some of the conventional wisdoms 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 3.2.1 and Section 3.2.2. 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 GCN training. Thanks to the design flow optimizations we discussed in Section 3.3, we do not observe any noticeable performance impact from the interprocess communications.

4.4.2 Node Feature Size Sensitivity Analysis. Even though we use multiple different graphs to evaluate GCN training performance, some of other real-world datasets can have very different node feature dimensions. For example, the node feature dimension of Pinterest dataset [57] 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 it is 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 method.

In Figure 12, 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 GCN training processes take a sizeable 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 GCN with large node feature dimensions.

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 [39]. 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 a same piece of data between different GPU instances, but such advances in supporting hardware partitioning capability can potentially help isolating 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 of 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 3.16μs in our setup, which is identical to the time of sending 85.5KB of data over PCIe 4.0 x16. Therefore, if the size of individual item is larger than 85.5KB, using a series of DMA operations can be an alternative of zero-copy method. For the GCN training, our current largest node feature size is 16KB, as shown in Section 4.4.2, which is only about 18.7% of the 85.5KB requirement.

Applications Beyond GCNs. Our work benefits other machine learning models than GCNs 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 [31] involves sparse embedding lookup process, thus benefiting from our work[23, 48]. Aside from models for large-scale recommendation systems, our work also benefits some traditional machine learning operators. For instance, the Hummingbird compiler [30] 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.

6 RELATED WORKS

GCN Training on Very Large Graph. One of the most notable works of extreme-scale GCN training is PinSage [57] which used a graph with 3 billions of nodes and 18 billions of edges. In this work, multiple GPUs were used to accelerate the training process. Similar to the DMA-based method that we described in our paper, PinSage also utilizes CPU to gather node features from the host side and then DMA to GPUs. There are other works which desert GPUs and train only with CPUs due to the extreme memory capacity requirement. SIGN [10] uses GPUs in training, but the whole neighboring node aggregation steps are actually done by CPU. DistDGL [59] uses CPU-only distribute system to parallelize graph neural network training. To manage the distributed storage, DistDGL requires complex data management processes to provide data in a timely fashion. Results in our work show that having GPUs to perform zero-copy access to the host memory can offer major performance advantage over using only the CPUs.

Alternative GCN Algorithms. The out-of-GPU-memory issue can be also circumvented by modifying the application itself. Several works [7, 20, 27, 58] attempt to partition input graphs into smaller clusters prior to the training phase so each cluster can be fit into the GPU memory. In this case, each training processor does not have view to the entire graph but only to the assigned clusters. The immediate issue of this method is that partitioning graphs creates bias in the training result as it clusters similar nodes together [7]. Furthermore, partitioning graphs results in losing multiple edges which represent relational information [56]. Especially with larger graphs, we need to create more number of partitions, and thus we also need to remove more number of edges during the partition. It is empirically shown that the partitioning methods result in lower accuracy in several GCN training workloads [1].

Other Graph Workload Accelerations on GPU. GCN training is not the only workload where the GPUs suffer from inefficient irregular host memory accesses. There are several works which try to utilize GPUs in graph traversal workloads like PageRank [41] with large datasets. Due to the usage of sparse matrix format for the representation of graph structure, traversing graphs results in generating very irregular memory accesses. Considering that large graphs such as WDC14 [2] has about 64 billions edges, the graphs cannot be placed in GPU memory and therefore the graph traversal workloads face the similar issue in this paper. EMOGI [28] utilizes zero-copy accesses to enable fine-grained host memory access during several graph traversal workloads. Halo [12] tries to ensure the spatial locality of graph nodes in the memory as well through extensive pre-processing. However, the effectiveness of this method is completely random depending on the shape of the input graph. Subway [46] uses a method very similar to the DMA-based method used in this work, which tries to utilize CPU as much as possible to gather scattered data for more efficient DMA. Marius [29] performs locality aware graph partitioning for a large scale knowledge-graph training.

7 CONCLUSION

In this work, we introduced a GPU-oriented, software defined data transfer architecture for efficient GCN training on large graphs. In large-scale GCN training, one of the most challenging tasks is that how 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 GCN 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 GCN training flow in PyTorch, we also show that our modifications can be seamlessly integrated with the existing high-level DNN programming models.

ACKNOWLEDGMENTS

This work was partially supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR). The authors would also like to thank the anonymous reviewers for their constructive feedback during the reviewing process.

REFERENCES

[1] [n.d.]. *Leaderboards for Node Property Prediction.* Retrieved Feb 21, 2021 from https://ogb.stanford.edu/docs/leader_nodeprop/

[2] [n.d.]. *Web Data Commons - Hyperlink Graphs.* Retrieved Feb 21, 2021 from http://webdatacommons.org/hyperlinkgraph/index.html

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

[4] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. 2014. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014.*

[5] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *International Conference on Learning Representations.* https://openreview.net/forum?id=rytstxWAW

[6] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 942–950. http://proceedings.mlr.press/v80/chen18p.html

[7] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Anchorage, AK, USA) *(KDD '19)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/3292500.3330925

[8] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3844–3852.

[9] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds.*

[10] Fabrizio Frasca, Emanuele Rossi, Davide Eynard, Benjamin Chamberlain, Michael Bronstein, and Federico Monti. 2020. SIGN: Scalable Inception Graph Neural Networks. In *ICML 2020 Workshop on Graph Representation Learning and Beyond.*

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

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

[13] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[15] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 52–74. http://sites.computer.org/debull/A17sept/p52.pdf

[16] Mark Harris. 2013. *How to Access Global Memory Efficiently in CUDA C/C++ Kernels.* https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/

[17] Mark Harris. 2017. *Unified Memory for CUDA Beginners.* https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 630–645.

[19] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[20] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with Roc. *Proceedings of Machine Learning and Systems (MLSys)* (2020), 187–198.

[21] Thomas N Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *NIPS Workshop on Bayesian Deep Learning* (2016).

[22] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR).*

[23] Suresh Krishna and Ravi Krishna. 2020. Accelerating Recommender Systems via Hardware "scale-in". *CoRR* abs/2009.05230 (2020). arXiv:2009.05230 https://arxiv.org/abs/2009.05230

[24] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488173

[25] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1, 4 (1989), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

[26] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *International Conference on Learning Representations.* https://openreview.net/forum?id=SkhQHMW0W

[27] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 443–457.

[28] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. 2020. EMOGI: Efficient Memory-access for Out-of-memory Graph-traversal In GPUs. *arXiv preprint arXiv:2006.06890* (2020).

[29] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. 2021. Marius: Learning Massive Graph Embeddings on a Single Machine. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 533–549. https://www.usenix.org/conference/osdi21/presentation/mohoney

[30] Supun Nakandala, Karla Saur, Gyeong-In Yu, Konstantinos Karanasos, Carlo Curino, Markus Weimer, and Matteo Interlandi. 2020. A Tensor Compiler for Unified Machine Learning Prediction Serving. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 899–917. https://www.microsoft.com/en-us/research/publication/a-tensor-compiler-for-unified-machine-learning-prediction-serving/

[31] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[32] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[33] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. 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)*. Association for Computational Linguistics, Hong Kong, China, 188–197. https://doi.org/10.18653/v1/D19-1018

[34] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. 2016. Learning Convolutional Neural Networks for Graphs. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. PMLR, New York, New York, USA, 2014–2023. http://proceedings.mlr.press/v48/niepert16.html

[35] Nvidia. 2016. *Nvidia Tesla P100 Whitepaper.* https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf

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

[37] NVIDIA. 2020. MULTI-PROCESS SERVICE. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

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

[39] NVIDIA. 2020. NVIDIA MULTI-INSTANCE GPU USERGUIDE. https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf

[40] NVIDIA. 2021. KERNEL PROFILING GUIDE. https://docs.nvidia.com/nsight-compute/pdf/ProfilingGuide.pdf

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

[42] Carl Pearson, Abdul Dakkak, Sarah Hashash, Cheng Li, I-Hsin Chung, Jinjun Xiong, and Wen-Mei Hwu. 2019. Evaluating Characteristics of CUDA Communication Primitives on High-Bandwidth Interconnects. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering* (Mumbai, India) *(ICPE '19)*. Association for Computing Machinery, New York, NY, USA, 209–218.

https://doi.org/10.1145/3297663.3310299

[43] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[44] PyTorch. 2021. TorchElastic. https://pytorch.org/elastic/0.2.2/index.html

[45] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783721

[46] Amir Hossein Nodehi Sabet, Zhijia Zhao, and Rajiv Gupta. 2020. Subway: Minimizing Data Transfer during out-of-GPU-Memory Graph Processing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 16 pages.

[47] Tim Schroeder. 2011. *Peer-to-Peer & Unified Virtual Addressing*. https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf

[48] Misha Smelyanskiy. 2019. Zion: Facebook Next- Generation Large Memory Training Platform. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–22. https://doi.org/10.1109/HOTCHIPS.2019.8875650

[49] Nikko Strom. 2015. Scalable distributed DNN training using commodity GPU cloud computing. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[50] V. Sze, Y. Chen, T. Yang, and J. S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (2017), 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740

[51] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[52] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. Ph.D. Dissertation. UC Berkeley.

[53] Kuansan Wang, Iris Shen, Charles Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft Academic Graph: when experts are not enough. *Quantitative Science Studies* 1, 1 (February 2020), 396–413. https://doi.org/10.1162/qss_a_00021.

[54] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. *SIGPLAN Not.* 53, 1 (Feb. 2018), 41–53. https://doi.org/10.1145/3200691.3178491

[55] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[56] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

[57] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 974–983. https://doi.org/10.1145/3219819.3219890

[58] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJe8pkHFwS

[59] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. arXiv:2010.05337 [cs.LG]