Exploring Page-based RDMA for Irregular GPU Workloads

A case study on NVMe-backed GNN execution

Benjamin Wagley*
bwagley@mines.edu
Colorado School of Mines
Golden, Colorado, USA

Pak Markthub
pak_markthub@hotmail.com
NVIDIA
Japan

James Crea
Bo Wu
Mehmet Belviranli
jcrea@mines.edu
bwu@mines.edu
belviranli@mines.edu
Colorado School of Mines
Golden, Colorado, USA

ABSTRACT

Paged memory systems for GPUs like NVIDIA's Unified Virtual Memory, offer a simple method for programmers to create out-of-core programs on GPUs. In the case of storage backed approaches, these systems can even handle larger than host memory systems as NVMe is used to back GPU memory through RDMA. However, paged memory systems can struggle with irregular access patterns. In this work, we analyze the limitations of paged, RDMA-backed GPU memory for out-of-core, irregular workloads, through a case study of GNN training. We highlight the key limitations of these systems that must be overcome before the true potential of RDMA backed GPU memory can be realized in a paged memory architecture.

ACM Reference Format:
Benjamin Wagley, Pak Markthub, James Crea, Bo Wu, and Mehmet Belviranli. 2024. Exploring Page-based RDMA for Irregular GPU Workloads: A case study on NVMe-backed GNN execution. In *16th Workshop on General Purpose Processing Using GPU (GPGPU '24), March 02, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649411.3649413

1 INTRODUCTION

NVIDIA's Unified Virtual Memory (UVM) encompasses memory across multiple accelerator cards, as well as host system memory, allowing for a GPU to address a memory space larger than physical vram. Recent projects like Dragon [9] and Dragon-direct [8] extend this memory system to further include NVMe storage, creating a memory framework that leverages storage across the memory hierarchy. This creates an attractive paradigm for breaking memory constraints and increasing scalability in GPU accelerated tasks, which are often bound by memory capacity when running on GPU. However, it is important to consider the effectiveness of this paradigm on irregular memory accesses. To examine this scenario we will consider the task of graph neural network (GNN) training, which often requires out-of-core training techniques that can be afforded by a unified memory system, and provides an irregular access pattern.

GPUs limited memory often restricts their effectiveness for general tasks. The growing requirement for large GPU memory is reflected by manufacturers like NVIDIA releasing cards with up to 80 GB of VRAM, tailored to data-center use [2]. Breaking memory capacity limits is important as data can still out-pace the available memory from top of the line cards, and often limits the usage of smaller, more efficient cards for general purpose tasks [7]. We examine page-based systems for breaking memory limitations due to their attractiveness to end-users, reducing the workload of a programmer optimizing out-of-core solutions.

In this work our primary contribution is to examine the limitations of page-backed RDMA memory systems for irregular GPU workloads, with a case study on GNN training applications. We highlight why these systems are attractive and valuable to programmers, and what limitations exist in current GPU page systems that restrict their effectiveness.

2 BACKGROUND AND MOTIVATION

2.1 Unified Virtual Memory

NVIDIA's UVM provides a unified memory space for their GPUs that leverages GPU page systems to extend GPU memory. At its core, UVM divides its memory space into 2 MiB `virtual address blocks (va-blocks)` that function as the primary memory system logical division. Each va-block can be further divided into memory pages equal to the host system page size. When a GPU thread accesses data that is not already resident on the faulting GPU, execution of the thread must stop and the required memory must be fetched. To accomplish this, the GPU memory manager generates a page fault, appends it to a fault buffer on the host system, and sends an interrupt to signal to the driver that a fault needs to be serviced. To handle the fault, the driver locates where a valid copy of the faulting page exists and copies it to the target GPU, then replays the fault allowing the thread to proceed.

Through this page-based system, UVM allows a programmer to break memory limits easily, while still utilizing a single address space. In practice, this means a programmer can build an out-of-core application by simply changing the allocation of their GPU data to UVM's managed memory. However, much of memory pipeline is out of the control of the programmer and can incur excessive costs

*Corresponding Author

GPGPU '24, March 02, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-1817-5/24/03
https://doi.org/10.1145/3649411.3649413

depending on workload. While the programmer can mitigate some of this overhead through careful consideration of their memory accesses, limitations incurred through the system cannot be easily rectified [1].

2.2 Expanding GPU memory with RDMA

Dragon-direct extends the UVM paradigm by allowing for page-faults to be serviced by RDMA transfers from NVMe directly to GPU memory. Dragon-direct is a built-in extension of the `nvidia-uvm` Linux kernel module, allowing for a minimal impact on user-space programming, much like UVM [8]. To a programmer, Dragon-direct functions much like Linux's `mmap`. To create a NVMe backed GPU allocation, the user calls `dragon_map` from the interfacing library, specifying the allocation size, and location of a backing file on an NVMe device. To a NVIDIA CUDA kernel, this allocation appears to be a standard UVM (managed memory) allocation. However, when a page fault is issued, Dragon-direct intercepts it and initiates a RDMA request for data from the NVMe backed allocation to be sent directly to the GPU. This system bypasses host memory entirely, in theory allowing for faster NVMe -> GPU transfers, and opens up a new paradigm for data management, similar to memory-mapped files in C++. Our work leverages Dragon-direct to evaluate RDMA performance through the UVM page system.

3 CASE STUDY: GRAPH NEURAL NETWORKS

Graph Neural Networks (GNNs) have arisen as the predominate method for analyzing graph data with deep learning. GNNs learn aggregators that collect feature data from the neighborhood around the target vertex or edge, and can be trained with supervised and unsupervised methods. In general, GNNs focus on vertex-level, edge-level, or graph-level tasks [12]. GNNs can be found in recommendation systems [15], molecular fingerprinting [3], drug discovery [4], citation network analysis and many other domains [6]. This case study focuses on the common application of vertex-level tasks, in particular vertex-level prediction.

GNNs are an attractive case study for out-of-core memory systems due to their high memory pressure and complex access patterns. They represent a problem class that is highly dependent on memory system performance, often utilizing small computational kernels resulting in memory-transfer dominated latency. A common approach to GNN training leverages mini-batches, which implicitly transfer well to the out-of-core scenario. This is because in mini-batch training, only the sampled portion of the graph, as well as the model parameters and sampled labels, are required to fit into GPU memory. Commonly, systems store the whole graph in host memory or in non-volatile storage. Then the system can sample the graph on the CPU before serving batches to the GPU for training, or in the case of pre-sampled graphs, serve batches directly for storage to the GPU. Mini-batch training for GNNs was popularized with GraphSAGE [5], which remains a benchmark model for basic GNN tasks. Distributed training methods operate similarly, leveraging multiple GPUs or compute nodes to work on separate mini-batches in parallel [13].

3.1 GraphSAGE Training Pipeline

In this paper, we examine the GraphSAGE training pipeline as a typical vertex-classification GNN training pipeline. When training in GraphSAGE the target training graph is divided into mini-batches of random k-hop samples. Each of these samples represents a random selection of training vertices and the k-hop neighborhoods around the sampled vertices. Then, during the forward pass for each mini-batch, k aggregators are used to aggregate neighboring node data down to the target nodes, with the ith aggregator functioning on the ith hop in the neighborhoods. Finally, prediction can be performed on the target training nodes[5].

By focusing on training neighborhood-based aggregators, the GraphSAGE approach utilizes small aggregators and opens the door to mini-batch training. This also leaves the memory system to play a critical role in training latency, as transferring batches to the GPU for training can dominate the small training forward-pass. Since generating batches can be seen as a graph partitioning problem, many GNN training works focus on improving batch sampling to minimize feature reuse across batches (partition uniqueness). This allows mini-batches to be more efficient in regard to knowledge learned per batch, and minimizes redundant transfers, but necessitates complex sampling methods and memory systems [16] [10] [11] [14].

4 EXPERIMENTS

For our case study we examine a simple training architecture that leverages Dragon-direct backed feature and label tensors. Since features dominate memory complexity of GNN training, we allocate topology on host-memory backed UVA tensors. Detail of the memory architecture of this experiment can be seen in Figure1. This work focuses on graphs from the Open Graph Benchmark (OGB) vertex prediction datasets `ogbn-arxiv`, `ogbn-products`, and `ogbn-papers100M`. All three graphs are for multi-class GNN classification, and represent different scales of similar problems. `ogbn-arxiv` and `ogbn-papers100M` are both paper citation networks of largely different scales. `ogbn-products` is an Amazon product co-purchasing network [6].

4.1 Theoretical Improvements

An interesting side-effect of this naive implementation is that inter-batch feature data reuse does not necessitate a memory transfer. This is because, by leveraging a paged-memory system like Dragon-direct, pages can remain resident on the training GPU between batches (if GPU memory allows). We utilize a pre-sampled training pipeline, allowing us to focus on memory transfers during training. Note that pre-sampled training pipelines are sometimes leveraged in production training for efficiency and data stability [16].

Table 1: Mean Probability for Feature Data Reuse for OGB Node Prediction Datasets, with Expected Subgraph Size, 2-hop samples.

Graph	5%		10%		20%	
arxiv	0.127	10.5 MiB	0.242	20.0 MiB	0.440	36.4 MiB
products	0.141	132 MiB	0.268	250 MiB	0.482	450 MiB
papers100M	0.106	5.61 GiB	0.203	10.8 GiB	0.376	19.9 GiB

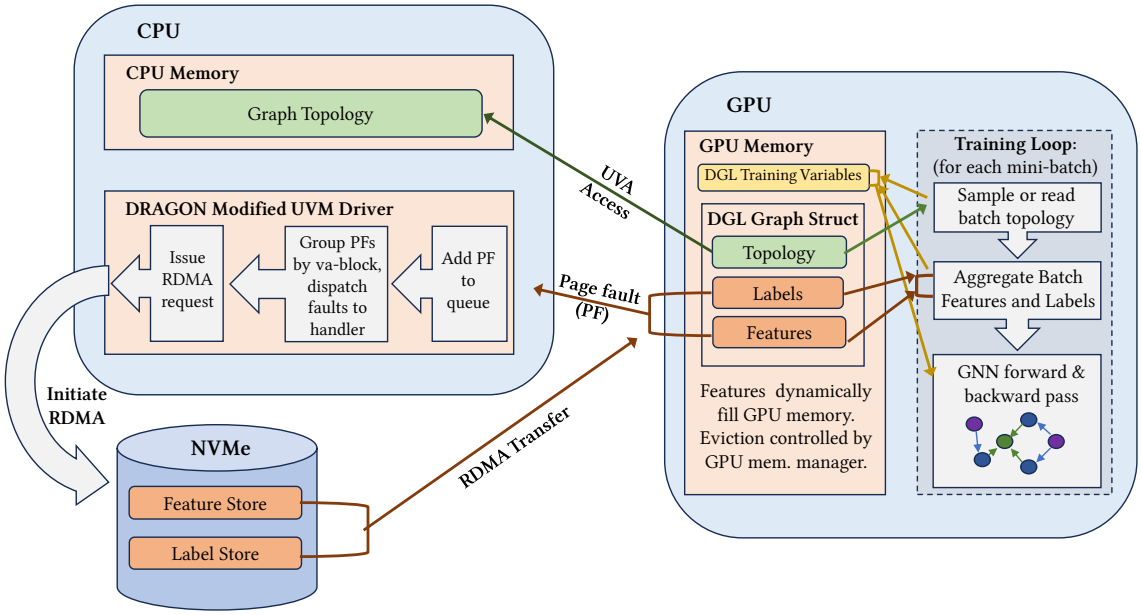

Figure 1: Experimental Setup with UVA-backed topology, and Dragon-direct backed features and labels.

By using a page-backed unified memory system we can expect training latency to decrease due to the feature data reuse between batches. Consider the probability of a feature being sampled in a given mini-batch. Since batch samples are independent, this is equivalent to the probability of a feature being included in two consecutive batches. We can model this probability numerically to understand the potential overhead in memory copies that a paged memory system eliminates. Consider $\hat{P}^0$ to be the probability vector that any given vertex is selected for sampling. This is the ratio of batch size to graph size. Given an adjacency matrix A for the given graph, we can compute the probability that a feature is sampled in the kth hop of the neighborhood around a sampled node with $\hat{P}^k = P^0 A^k$. This allows us to derive a numerical model for feature reuse probability. For example, the 2-hop neighborhood sample can be modeled with,

$$\mathcal{P}(v \in S) = \hat{P}_v^0 + \hat{P}_v^1 + \hat{P}_v^2 - (\hat{P}_v^0\hat{P}_v^1 + \hat{P}_v^0\hat{P}_v^2 + \hat{P}_v^1\hat{P}_v^2) + (\hat{P}_v^0\hat{P}_v^1\hat{P}_v^2)$$

We outline feature-data reuse in Table 1. We can see that if only 5% of a graph is selected to perform a 2–hop sample around for a mini-batch there is greater than a 10% probability that feature data will be reused between two batches for all graphs we evaluated. This increases in relation to the ratio of batch sample size to graph size.

4.2 Implementation Details

To better understand the impact of the memory system on GNN training we evaluate two systems. First, we implement a baseline where each batch is copied to the GPU from disk before training. We copy from disk to GPU as this is the same path the data must take when using a Dragon-direct backed buffer. Second, we implement our test using Dragon-direct backed features, which should allow some feature data to remain resident on GPU between batches. For this approach, graph topology is sampled ahead of time and stored on disk. During each batch topology is read and feature data is coalesced. See Algorithms 1 and 2 for pseudo-code.

Table 2: Test Configurations

Test	Fanouts	Topology	Feature/Label
Baseline Pre-Sampled	[16, 16]	cudaMalloc/UVA	cudaMalloc/UVA
RDMA Pre-Sampled	[16, 16]	cudaMalloc/UVA	Dragon-direct

4.3 Target System

All experiments were run on a system with 2x AMD Epyc 7402 processors, 512GiB of host memory, 2x NVIDIA Titan RTX, and an ADATA XPG SX8200 NVMe PCIE 3.0 SSD. Due to the fact that Dragon-direct is built into a modified NVIDIA UVM driver this server is running a customized NVIDIA driver 455.23.05 with a CUDA 11.1 stack. The OS page-cache is cleared between tests to eliminate any variance it may cause.

4.4 Results

When considering pre-sampled training, we can see that the two smaller graphs that we test show promising speedup as reused feature data can remain resident on GPU. Note that latency when training `products` plateaus, this is likely due to the fact that the fanouts (16 at each layer) are substantially smaller than the average node degree of `ogbn-products` (50.5). This likely limits the

Table 3: Latency of Pre-Sampled Training

	Baseline		RDMA		
Graph (Sampled Center Vertices)	Accuracy	Avg Time Per Epoch	Accuracy	Avg Time Per Epoch	Speedup
arxiv (8467)	0.661	7.167	0.663	3.669	1.953
arxiv (16934)	0.674	8.207	0.675	4.607	1.781
arxiv (33869)	0.685	10.282	0.684	7.098	1.449
products (122452)	0.756	108.673	0.755	72.445	1.500
products (244903)	0.752	140.197	0.753	99.792	1.405
products (489806)	0.754	139.64	0.753	100.141	1.394
papers100M (55530)[a]	0.71	130.21	0.76	60783.90	0.002

[a] papers100M results collected after a single epoch due to execution time

input : Set of training subgraphs `S`, Untrained model `M` for target graph
output: Trained GraphSAGE model `M`
for $i \leftarrow 1$ *to* `num_epochs` **do**
  for `s` $\in$ `S` **do**
    read `s` from disk
    `inputs`, `labels` $\leftarrow$ feats(s), labels(s)
    `predictions` $\leftarrow$ `M`(s, `inputs`)
    Compute Loss, Backpropagate,
  end
end

Algorithm 1: Baseline Pre-Sampled Training

input : Set of training subgraphs `S`, Dragon-direct mapped features, labels `feats`, `labels`, Untrained model `M` for target graph
output: Trained GraphSAGE model `M`
for $i \leftarrow 1$ *to* `num_epochs` **do**
  for `s` $\in$ `S` **do**
    read `s` from disk
    `o_idx` $\leftarrow$ original vertex ids sampled from target graph
    `inputs`, `labels` $\leftarrow$ `feats`[`o_idx`], `labels`[`o_idx`]
    compact `inputs`, `labels` to dense format.
    `predictions` $\leftarrow$ `M`(s, `inputs`)
    Compute Loss, Backpropagate,
  end
end

Algorithm 2: RDMA Pre-Sampled Training

available speedup from feature reuse. Furthermore, the `arxiv` and `products` datasets can reside entirely on GPU, meaning that all feature pages should fit in GPU memory, and by the end of the test the entire graph will be resident, eliminating all transfers between batches. We can see however, that even we do not see such speedup for our large out-of-core test with `papers100M`. Full results can be seen in Table 3.

4.5 Memory System Performance

To understand the limitations of a paged RDMA system with out-of-core workloads, we further measured memory system performance. These experiments outline the limitation of CPU managed, page based systems, and highlight their particular failings when working with irregular access patterns.

First, we examine the effect of page size on memory pressure. UVM (and by extension, Dragon-direct), utilize 2 MiB va-blocks as a course-grained boundary for page operations. This is controlled by a defined parameter in the `nvidia-uvm` driver. We evaluated memory pressure generated by various va-block sizes from 4 KiB to 2 MiB, when accessing 8192 floating-point features of dimension 128. These features were spaced out such that each feature lie 2 MiB apart in memory. The physical size of only the features accessed is therefore 4 MiB. We can see that highly excessive memory pressure is created, up to ~4,342x. Full results are in Table 4. Note that this extra memory pressure does include Pytorch management structures, however as these structures are not allocated by UVM or Dragon-direct, their size does not change as we change the UVM page size.

While excessive memory pressure leading to thrashing does increase the latency of memory accesses (and therefore the training pipeline), long page fault handling times also lead to this latency. To examine this, we can turn to the throughput of each memory system, which will showcase how well each system utilizes the bandwidth of each tier in the GPU memory hierarchy. For each test, we are measuring the throughput of reading data from NVMe to GPU memory. For all tests except `gdsio` and `cudaMalloc`, we utilize a test written for Dragon-direct that optimizes accesses per-warp for each UVM page for the page-fault initiated transfers. We compare these results to `gdsio`, the Nvida utility for measuring GPU Direct RDMA performance with NVMe, which is the underlying technology that Dragon-direct builds on. The `gdsio` tests represent a best-case scenario of sequential RDMA transfers, and showcase the theoretical upper limit of performance for a NVMe -> GPU RDMA system. Note that `gdsio` requires a newer NVIDIA driver to function (520.61.05). The Dragon and Dragon-direct tests are run over 256 MiB test files to mitigate the performance impact of the 256 MiB BAR1 on the testing GPUs. All other tests are over a 1 GiB test file. The theoretical bandwidth of the SSD in the system is 3500 MB/s per the manufacturer specifications.

Table 4: UVM Memory Pressure

`va_block` Size (KiB)	Memory Pressure (MiB)
2048	17368.37
1024	1392.19
512	1342.12
256	1316.06
128	1304.0
64	1298.0
32	910
16	910
8	910
4	910

We observe in Table 5, Dragon and Dragon-direct do not utilize the full bandwidth of the NVMe device, and comparing Dragon-direct against `gdsio`, we can see that Dragon-direct does not equal theoretical bandwidth for RDMA transfers on the system hardware, implying that utilizing a paged system memory system has severe limitations.

5 ANALYSIS

Our experiments show that out-of-core pipelines leveraging a unified memory system do not affect the accuracy of GNN training, and could be used as a most basic extension to out-of-core training, at the cost of efficiency. In the case of using Dragon-direct for NVMe backed training, such as system could be effectively leveraged to allow out-of-core training on graphs larger than GPU *and* host memory, given the graph can fit within NVMe.

However, we see that the efficiency of training drops dramatically when we approach out-of-core workloads. Primarily, this inefficiency comes from the inability of the UVM page system to keep up with such irregular workloads. First, the large page size leveraged by UVM created excessive memory pressure on the GPU when used with sparse accesses. Second, page-faults are generated and handled inefficiently.

5.1 UVM Page Size

By default, UVM uses 2 MiB va-blocks for its core page size. This provides a low-granularity logical address block for page operations, and allows for effective prefetching within va-blocks for sequential workloads. Each va-block is further divided into system pages, meaning that within a va-block, not every system-level page may be serviced. Consequentially, we observed on our test system that even if only a single 4 KiB system page within a va-block has been serviced by the driver, the GPU memory manager marks an entire 2 MiB section of memory as allocated, increasing memory pressure by an extra 2044 KiB compared to the actual usage.

While this behavior has little impact when accessing a contiguous region of memory, as excess memory pressure would only be generated by edge regions, it is hazardous when accessing sparse data. In the worst case, sparse data may entirely lie within independent va-blocks, effectively creating 2 MiB of memory pressure *per item accessed*. As a result of this abnormally high memory pressure, the GPU memory manager may issue more page-faults than needed, causing thrashing on the GPU, and critically impacting performance. This effect can be somewhat mitigated by lowering the UVM va-block size, allowing for better page handling for random, sparse workloads. However, in the case of the `papers100M` dataset, an individual feature has an in-memory size of 512B, so even using 4 KiB va-blocks results in 8x extra memory usage. Fundamentally, to be effective for irregular access patterns on GPU, a page system needs to leverage smaller page sizes, or a user-controlled page size.

Table 5: Memory System Throughput

Test	Throughput (MiB/s)
fread + cudaMalloc	1804
fread + UVM	1665
Dragon	727
Dragon-direct	901
gdsio	2183

5.2 Memory System Throughput

We measured the throughput of the various memory system workflows copying data from NVMe to GPU memory, as it can indicate where a system has limitations. Our evaluation shows that initiating data transfer by page-fault has inherent, measurable slowdown. This can be seen in slowdown between the `cudaMalloc` and `UVM` experiments and the `gdsio` and Dragon-direct results. Note that `UVM` leverages intra va-block prefetching which cannot be easily disabled, boosting its performance. As these tests were performed over sequential buffers, a more optimal memory layout, they also represent a best-case throughput measurement. We can infer that the random performance is no better than the measured results.

We theorize that the CPU serves as a limitation for the number of page faults that can be processed at any given time in the UVM system. This is touched on within Allen and Ge's analysis of UVM [1], however we find that when leveraging Dragon-direct backed tensors, and therefore giving up the intra va-block prefetching that UVM utilizes, the effect is pronounced. Fundamentally, a GPU kernel operating on a large, sparse dataset may generate a large number of page faults at any given time. Consider NVIDIA's H100, which features 144 streaming multiprocessors (SMs) per GPU, with each SM capable of running 64 warps concurrently [2]. If utilizing an architecture that issues 1 page fault per warp, such a system could theoretically generate 9216 page faults simultaneously. This highlights the capability mismatch of a SIMD device generating page faults across sparse, irregular access patterns, and the SISD nature of a kernel thread handling page faults.

Given the latency of handling page-faults, and the systems limited ability to process page-faults on demand, it is important for page-faults to be issued as optimally as possible. Unfortunately, this can be a struggle with random, sparse accesses as we see during feature aggregation during GNN training. When analyzing fault handling utilizing NVIDIA's `nsys` profiling utility, we noticed how few page-faults are handled at any given time. When profiling the page-faults of our throughput tests, which accesses pages in a warp-optimal pattern, we observe the driver handling 2.64 page faults per handler on average. This is despite the fact that the majority of page faults over the test 256 MiB region should be issued at the same time.

UVM performs slightly better, but also has a similar bottleneck, achieving an average of 3.28 page faults per handler. This highlights that page-faults are not arriving at the driver efficiently. Note that while these numbers were observed using `nsys` profiling, the profiler should not impact that all page-faults should be issued at the same time. This may be evidence of GPU memory-manager inefficiency when issuing page-faults, which would require further examination. Primarily, we can see that beyond the logical many-few bottleneck of having a CPU handle page faults, these page faults appear to not be issues optimally, compounding the bottleneck effect.

These results show us that, beyond page size considerations, a GPU page system must be able to handle page faults simultaneously, as well as generating and sending those page faults to their handler more efficiently. Our analysis shows that UVM's page system is not capable of this, and restricts its effectiveness with irregular workloads. Furthermore, we show that even when utilizing RDMA

to simplify the disk - GPU memory hierarchy, the page system remains the primary limitation.

6 CONCLUSION

In this work we have discussed the page-based system that UVM provides to the GPGPU environment, and how this system can be extended to encompass a memory hierarchy from NVMe disk to the GPU memory. We have show how a paged memory system can be attractive, even for irregular memory applications through our case study of GNN training workloads. We then analyze a paged, NVMe backed memory system and identify that its primary limitations arise from the underlying paged architecture.

We believe that an effective paged memory system for GPUs must encompass two primary traits – flexible, smaller page sizes, and parallel handling of page faults. In irregular, sparse workloads the detrimental effects of a large page size easily outweigh the simplified page-table architecture it affords, which quickly limits UVM's page system and subsequently Dragon-direct's ability to issue RDMA requests. However, utilizing smaller page sizes does not remedy the issue that a SIMD device like a GPU can generate a large number of page faults simultaneously, overwhelming traditional page-handling mechanisms operating in the kernel, such as in UVM and Dragon-direct. To fully utilize the potential of a parallel fault handling architecture, the GPU fault issuing system must also be optimized for a large number of faulting accesses at once.

Paged memory architectures offer many advantages to users, and can be powerful when leveraged effectively, however current GPU paged architectures are severely limited, in particular for irregular access patterns. By approaching GPU paging with a higher view of parallelism than is reflected in current architectures would open the doors for the advantages of paged memory the be realized in the GPU environment, and when used with systems like Dragon-direct would allow transparent access to terabyte-scale memory for GPU workloads.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation (NSF) under Grants No. CCF-2124010 and CCF-1750760. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] Tyler Allen and Rong Ge. 2021. In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1145/3458817.3480855

[2] Jack Choquette. 2023. NVIDIA Hopper H100 GPU: Scaling Performance. *IEEE Micro* 43, 3 (2023), 9–17. https://doi.org/10.1109/MM.2023.3256796

[3] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams. 2015. Convolutional networks on graphs for learning molecular fingerprints. *Advances in neural information processing systems* 28 (2015).

[4] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. 2021. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics* 22, 6 (2021), bbab159.

[5] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[6] 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. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[7] Youjie Li, Amar Phanishayee, Derek Murray, Jakub Tarnawski, and Nam Sung Kim. [n. d.]. Harmony: Overcoming the Hurdles of GPU Memory Capacity to Train Massive DNN Models on Commodity Servers. ([n. d.]).

[8] Pak Markthub. 2019. *Improving GPU-NVMe Data Transfer in Unified Virtual Memory Space.* Technical Report.

[9] Pak Markthub, Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter, and Satoshi Matsuoka. 2018. DRAGON: breaking GPU memory capacity limits with direct NVM access. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 414–426.

[10] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large graph convolutional network training with gpu-oriented data communication architecture. *arXiv preprint arXiv:2103.03330* (2021).

[11] Jeongmin Brian Park, Vikram Sharma Mailthody, Zaid Qureshi, and Wen-mei Hwu. 2023. Accelerating Sampling and Aggregation Operations in GNN Frameworks with GPU Initiated Direct Storage Accesses. *arXiv preprint arXiv:2306.16384* (2023).

[12] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. 2021. A Gentle Introduction to Graph Neural Networks. *Distill* (2021). https://doi.org/10.23915/distill.00033 https://distill.pub/2021/gnn-intro.

[13] Yingxia Shao, Hongzheng Li, Xizhi Gu, Hongbo Yin, Yawen Li, Xupeng Miao, Wentao Zhang, Bin Cui, and Lei Chen. 2022. Distributed Graph Neural Network Training: A Survey. *arXiv preprint arXiv:2211.00216* (2022).

[14] Roger Waleffe, Jason Mohoney, Theodoros Rekatsinas, and Shivaram Venkataraman. 2023. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Eighteenth European Conference on Computer Systems (EuroSys' 23)*.

[15] 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*. 974–983.

[16] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, et al. 2020. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454* (2020).