Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory

Sangjin Choi and Taeksoo Kim, *KAIST;* Jinwoo Jeong, *Ajou University;* Rachata Ausavarungnirun, *KMUTNB;* Myeongjae Jeon, *UNIST;* Youngjin Kwon, *KAIST;* Jeongseob Ahn, *Ajou University*

https://www.usenix.org/conference/atc22/presentation/choi-sangjin

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

多任务，抢占空闲
previous studies: cpu-gpu（UVM)
HUVM:cpu-gpus
利用其他gpu空闲显存进行数据移动，充分利用带宽，预卸载数据到cpu，增大数据移动单元，多gpu加载数据，预加载数据

Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory

Sangjin Choi* *KAIST*
Taeksoo Kim* *KAIST*
Jinwoo Jeong *Ajou University*
Rachata Ausavarungnirun *KMUTNB*
Myeongjae Jeon *UNIST*
Youngjin Kwon *KAIST*
Jeongseob Ahn† *Ajou University*

Abstract

With the ever-growing demands for GPUs, most organizations allow users to share the multi-GPU servers. However, we observe that the memory space across GPUs is not effectively utilized enough when consolidating various workloads that exhibit highly varying resource demands. This is because the current memory management techniques were designed solely for individual GPUs rather than shared multi-GPU environments.

This study introduces a novel approach to provide an illusion of virtual memory space for GPUs, called hierarchical unified virtual memory (HUVM), by incorporating the temporarily idle memory of neighbor GPUs. Since modern GPUs are connected to each other through a fast interconnect, it provides lower access latency to neighbor GPU's memory compared to the host memory via PCIe. On top of HUVM, we design a new memory manager, called memHarvester, to effectively and efficiently harvest the temporarily available neighbor GPUs' memory. For diverse consolidation scenarios with DNN training and graph analytics workloads, our experimental result shows up to 2.71× performance improvement compared to the prior approach in multi-GPU environments.

1 Introduction

As the demand for GPUs explodes, it is now a common practice in both academia and industry to equip multiple GPUs in a single server and make them shareable. Many enterprises in the industry have built large GPU clusters comprised of a set of multi-GPU servers to satisfy the demand for a variety of workloads from deep learning [1, 13, 18, 26, 36] to graph applications [6, 10, 19, 31] while saving the infrastructure cost by sharing. However, as a downside, achieving high GPU resource efficiency in such multi-GPU servers remains a challenge. Figure 1 presents that the current memory management technique is not effective enough for shared multi-GPU environments where multiple jobs are running across GPUs independently. Although a small amount of memory ranging from hundreds of MB to a few GB remains idle in one or a few GPUs, other GPUs under heavy memory pressure rely on the host memory as a swap device that is significantly slower than remote GPUs within the same server.

Meanwhile, GPU vendors have faced the challenge of scaling the memory capacity of single GPUs. To overcome the limited capacity of GPUs, a train of previous studies provides an illusion of infinite memory space with the host memory [11, 14, 17, 25, 28]. However, none of the work does utilize the idle memory of neighbor GPUs in commodity multi-GPU systems. As modern GPU servers are commonly equipped with 8~16 GPUs connected via high-speed interconnect such as NVLink, accessing the idle memory of neighbor GPUs is much faster than that of the host. For instance, NVIDIA DGX-2 has 16 GPUs with point-to-point connections through NVLink 3.0, yielding a large pool of 512GB GPU memory at 600GB/s bidirectional bandwidth [23]. On the other hand, swapping GPU memory to host DRAM via the latest PCIe 4.0 could utilize up to 32GB/s bandwidth only.

In this study, we introduce a new approach providing an illusion of virtual memory space for GPUs called *hierarchical unified virtual memory (HUVM)* comprised of local GPU, spare memory of neighbor GPUs, and the host memory. HUVM opens up a new opportunity for memory virtualization by increasing the effective memory space with minimal performance overhead. When the local GPU memory does not have free space, HUVM leverages the spare[1] memory in neighbor GPUs *as a victim cache* between the GPU and host instead of directly swapping out data to the host memory.

However, it is challenging to effectively and efficiently harvest the spotty-available, small fraction of neighbor GPUs' memory because the amount of idle memory is highly variable and unknown a priori. Beyond the single GPU perspective, we redesign the memory management scheme for modern multi-GPU servers. HUVM systems have to find the best

*Co-first authors
†Corresponding author

[1]The terms `spare`, `idle`, and `harvested` are used interchangeably.

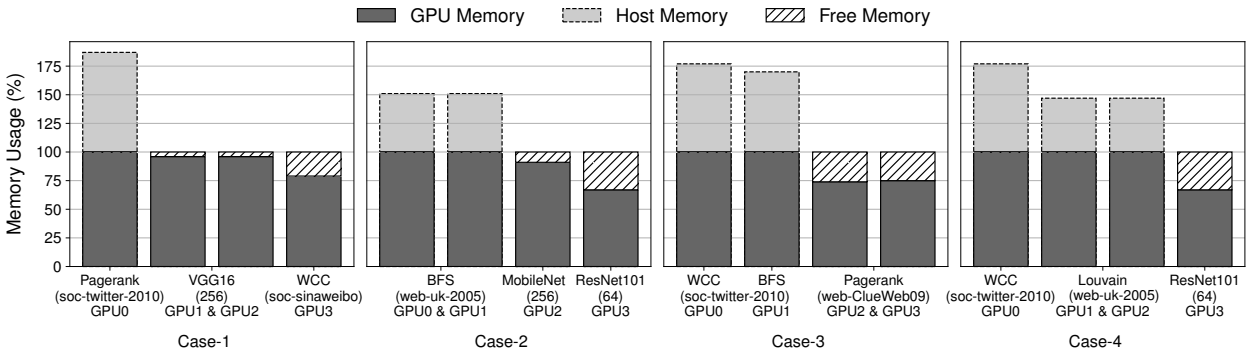

Figure 1: Memory usage snapshot in multi-GPUs hosting memory-intensive workloads (Section 6.1 and Table 2 present the detailed information for workloads and experimental environment)

way to utilize the small fraction of harvested memory while minimizing the performance impact on workloads running in the neighbor GPUs.

To that end, we propose a memory manager of HUVM, called *memHarvester*, implemented in the GPU driver layer. memHarvester functions as a centralized coordinator for datapath in HUVM. As an essential part of memHarvester, we propose a new multi-path parallel prefetcher, which exploits the path diversity of HUVM, comprised of PCIe and NVLink. Unlike many previous approaches [11, 14, 17, 25, 28] relying only on the host memory via PCIe, memHarvester first prefetches data from the spare memory to the local GPU through NVLink. Meanwhile, if the PCIe channel attached to the neighbor GPU is not contended, memHarvester allows for prefetching the data from the host memory to the spare memory of the neighbor GPU through the PCIe channel in parallel. Therefore, we can convert the latency of fetching data from the host memory to that of the spare memory effectively. memHarvester manages the space of harvested memory. Due to the limited space of spare memory, memHarvester is unable to keep all the evicted data in the harvested memory, leading to the host memory swap eventually. To reduce the performance overhead of data eviction from GPU to host, memHarvester supports eviction with 2MB large pages to host memory instead of 4KB base pages.

When HUVM and memHarvester host multiple workloads, it can improve memory utilization and overall server efficiency. However, on a downside, memHarvester may cause performance interference to the applications running on the GPU yielding idle memory. Thus, memHarvester immediately reclaims the spare memory to give it back to its original physical memory space with minimal latency whenever the application running on the yielding GPU needs additional memory, thereby minimizing performance interference.

We implement our prototype system on top of NVIDIA's unified virtual memory (UVM) driver version 460.67, which is publicly accessible [29]. Without any modifications to applications or machine learning platforms, memHarvester transparently detects the availability of spare memory and dynamically constructs a new memory hierarchy. We quantify the effectiveness of memHarvester with HUVM for diverse consolidation scenarios on an AWS `p3.8xlarge` instance. The server has four NVIDIA V100 (16GB) GPUs connected through NVLink. Our experimental result shows that `memHarvester` can achieve significant throughput improvement for the large graph analytics workloads. For diverse consolidation scenarios with DNN training and graph analytics workloads, our experimental result shows up to 2.71× performance improvement compared to the prior approach in multi-GPU environments with minimum interference of other workloads running on the same server.

2 Motivation and Background

This section characterizes memory usage behaviors of emerging workloads in shared multi-GPU environments and discusses opportunities to improve overall memory utilization by exploiting idle memory of neighbor GPUs.

2.1 Memory Usage in Shared Multi-GPUs

With the ever-growing demands of GPUs from development to deployment, most organizations allow semi-trusted users (e.g., employees in a company) to share the multi-GPU servers, reducing the cost of building the infrastructure [9, 13, 18, 35, 36]. Due to the shared nature, such GPU servers consolidate a wide range of workloads. In particular, many of the jobs running in the shared GPU servers are DNN training workloads for computer vision and natural language processing [11, 25, 28] that take a long time to complete. Thus, it leads to limited resource availability of certain GPUs at a time. Another widely witnessed application in multi-GPU servers is graph analytics [10, 19, 31], which figures out the relationships between objects in a given graph.

When looking at the memory consumption of such two emerging workloads, it is usually required for DNN training

jobs to tune the batch size to almost fit on GPUs to achieve maximum resource utilization [28]. On the other hand, the memory consumption of graph analytics jobs depends on the number of edges and vertices of a given graph. For a large dataset that does not fit in the given GPU memory, one can leverage a graph partitioning approach to make each partition fit on individual GPUs. However, the graph partitioning task introduces additional complexity in implementation [4] such that some of the graph algorithms are not supported to run on multi-GPUs (e.g., `Betweenness Centrality` in cuGraph [6]). To overcome the memory space limitation, we can leverage host-side memory as a swap device to the GPUs through the unified virtual memory (UVM) technique that provides an illusion of infinite memory space [29]. Although this enables us to run analytics on large graphs or DNN training with large batches without out-of-memory errors, it significantly degrades performance.

In shared multi-GPU servers, we observe that a small amount of memory space across GPUs remains idle. Figure 1 shows memory snapshots of a 4-GPU (V100) server hosting multiple memory-intensive workloads. We profile four running scenarios each of which runs either graph analytics or DNN training jobs using one or multiple GPUs. The experimental environment is described in Section 6.1. The result shows that some GPU memory is not fully utilized, causing memory imbalance across workloads. In `Case-1`, `VGG16` and `WCC` leave a small amount of memory of GPU-1, 2, and 3, whereas `Pagerank` has to use the host memory for a swap device to run the large dataset. Another example is to run `Pagerank` with a relatively small dataset (`web-ClueWeb09`) with two GPUs in `Case-3`. In this case, the memory of GPU-2 and 3 is not fully utilized. Even though `Louvain` and `BFS` experience heavy memory pressure, the current memory management technique does not leverage the idle memory of the neighbor GPUs. These results confirm that the current memory management design is still not efficient in shared multi-GPU systems, wasting valuable GPU memory capacity. Considering that each workload has highly varying memory demands, achieving high global memory utilization in multi-GPU system is a challenging problem.

2.2 Exploiting Neighbor GPU Memory

Modern GPU servers provide useful primitives to facilitate memory harvesting, i.e., leveraging fast intra-server GPU interconnects without modifying applications or frameworks.

Fast interconnect. Modern GPU servers are commonly equipped with 8~16 GPUs connected via high-speed interconnect such as NVLink. For instance, NVIDIA DGX-2 has 16 GPUs with direct peer GPU access through fully connected NVLink topology, yielding a large pool of 512GB GPU memory at 600GB/s GPU-to-GPU bidirectional bandwidth [23]. With this HW specification, modern GPU servers can provide an attractive option for GPU-to-GPU communication to transmit data without using the expensive PCIe interconnect.

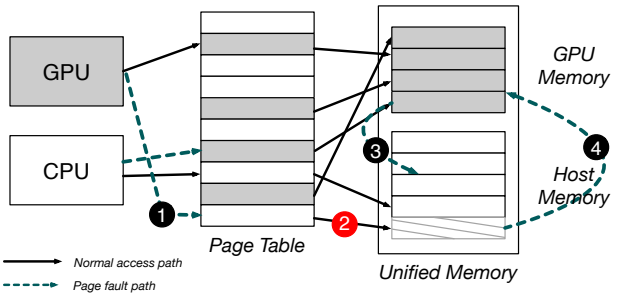

Figure 2: Unified address space in modern GPUs

Transparent control. Memory harvesting must address the cases that transfer data between GPUs running jobs across different applications. For example, in Figure 1, the graph analytics jobs and DL training jobs use their own frameworks. The unified virtual memory (UVM), introduced by NVIDIA and AMD, enables large-memory applications to seamlessly oversubscribe the limited GPU memory [29]. The technology is implemented as a GPU driver, so implementing memory harvesting in UVM requires no modification to the GPU applications or ML frameworks.

This section first explains how the current UVM driver works to leverage host memory for extending the limited GPU memory capacity. The UVM driver takes advantage of the host memory as a swap space for the GPU. Figure 2 presents how the UVM driver provides the unified address space across the GPU and host memory. The driver identifies whether the page being accessed by GPU is located on the GPU or the host memory through the page fault mechanism. With a single unified page table, UVM translates GPU virtual address into either GPU physical address or host physical address. If UVM identifies that the page accessed by the GPU kernel is mapped to the host by referring to the page table ❶, a page fault exception is raised ❷ and the UVM driver brings the page into the GPU memory (typically via PCIe interconnect) ❹. Meanwhile, when no free space is available in GPU, the driver needs to evict an old page from GPU memory before transferring the faulted page to GPU ❸. Moving these pages requires page table entries to be properly updated to map new locations.

Opportunity. As observed in Figure 1, if one GPU needs more memory than its memory capacity, it can spill the fraction of oversubscription to one or more neighbor GPUs that still consume less than the total memory capacity with the fast interconnect. Moreover, by leveraging the unified address space supported by the UVM driver, we can serve diverse jobs ranging from graph analytics to DL training jobs without modifying applications or frameworks. Such *harvesting* of idle memory in neighbor GPUs opens up a unique opportunity to lower performance overhead under memory oversubscription by increasing the effective memory space. To work well

	PCIe	NVLink (speedup)
Throughput (GB/s)	12.3	40.1 (3.3×)
Latency (μs)	16.7	5.1 (3.2×)

Table 1: Throughput and latency with PCIe and NVLink

with a very small fraction of idle memory, the mechanism for harvesting idle memory of neighbor GPUs should be timely and efficient. In Section 3.2, we address the design challenges in more detail.

3 Hierarchical Unified Virtual Memory

This section first measures the performance benefits of accessing the neighbor GPU's memory connected through the high-speed interconnect (NVLink). Based on the measurements, we introduce a new unified memory, called *hierarchical unified virtual memory*, tailored for multi-layered memory systems comprised of local GPU, neighbor GPUs, and host memory. With the new memory organization, this section discusses how to incorporate the spare memory of neighbor GPUs into the memory hierarchy to bridge the performance gap between local GPU memory and host memory. Accessing neighbor GPUs is faster than accessing the host memory, but the spare memory space is dynamic and often limited, and may be shared by multiple harvesters. Therefore, utilizing a small amount of spare memory is crucial for effective harvesting. Finally, we discuss design challenges when leveraging the spare memory of neighbor GPUs with limited capacity.

3.1 Data Path with HUVM

To understand the performance benefit of harvesting the neighbor GPU memory, we conduct a performance analysis to measure the throughput migrating 2MB[2] data from a GPU memory to the host memory via PCIe and from a GPU memory to the other GPU memory via NVLink on an AWS `p3.8xlarge` instance. Two GPUs are connected through NVLink 2.0 with two lanes, providing up to unidirectional bandwidth of 50GB/s. Table 1 shows a comparison of bandwidth and latency between PCIe and NVLink. As expected, such NVLink provides more than 3× better performance in terms of throughput and latency, indicating that data transfer time can be significantly reduced if we evict pages to neighbor GPUs. We anticipate that this performance gap will be higher in the next generation of high-speed interconnect.

Using the fast interconnect, i.e., NVLink, we build a new data path exploiting the spare memory of neighbor GPUs. Our approach brings two advantages in terms of performance. First, the new data path accelerates memory eviction. When evicting a memory chunk from the local GPU due to the lack of memory space, if there is idle space in the neighbor GPU, our approach uses NVLink rather than PCIe to evict the memory chunk. Second, the new data path can reduce the latency of fetching. As the spare memory can act as a victim cache, we populate as many evicted chunks as possible on the spare memory. If these chunks are accessed again shortly, the chunks are fetched to the local GPU with the fast NVLink.

[2]Currently, the unit of memory eviction is a 2MB chunk in the UVM.

3.2 Design Challenges

Although our hierarchical unified virtual memory can reduce the performance penalty of GPU memory oversubscription, utilizing the spare memory poses several challenges.

① **Effective harvesting:** If the spare memory is not sufficient to serve all the evicted pages from the GPU applications, the effectiveness of memory harvesting would be limited to only buffering the evictions.

② **Minimal interference:** Harvesting may cause performance interference of the application running on the yielding GPU. Since we borrow the NVLink and PCIe bandwidth of the neighbor GPUs for the spare memory, our harvesting technique needs to minimize the performance interference.

③ **Low overhead:** Since UVM relies on the page fault mechanism, our approach inherits the same performance overhead, manipulating page table entries. Such overhead can prevent us from using full PCIe and NVLink bandwidth.

④ **Framework-agnostic:** All the GPU jobs do not rely on a specific framework. Our design and implementation need to be generalized to host a wide range of GPU workloads.

4 Memory Management for HUVM

This section presents a new memory manager, called *memHarvester*, for HUVM. memHarvester aims to hide the latency from the performance-critical path in accessing host memory with small but fast spare memory of neighbor GPUs. memHarvester opportunistically harvests the spare memory of neighbor GPUs while minimizing the performance penalty of memory-intensive applications that require more memory than available in one or more GPUs. To do so, memHarvester identifies the availability of spare memory in neighbor GPUs and plugs the spare memory into the memory hierarchy dynamically. To effectively harvest the spare memory, we explore a set of techniques: hierarchical and background eviction, fetching data in parallel, and prefetching in a neighbor GPU memory.

4.1 Overview

By harvesting spare memory in multi-GPU systems, memHarvester creates an illusion of GPU applications having a small cache between a GPU and host memory. Figure 3 presents the

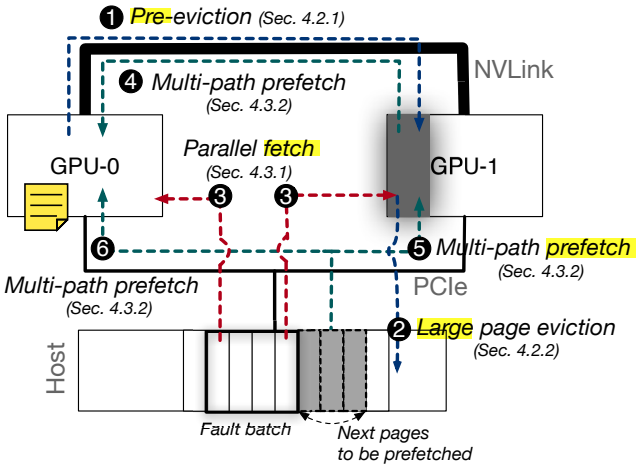

Figure 3: Exploiting spare memory with path diversity

overview of our proposed techniques. memHarvester allows GPU-0 to utilize the spare memory of GPU-1. ❶ Instead of evicting data to the host memory directly, memHarvester uses the spare memory connected through NVLink as a victim buffer to shorten the latency of memory evictions. To reduce the cost of memory evictions, memHarvester takes advantage of pre-eviction to the spare memory. With the spare memory, memHarvester reduces the number of evicting data to the host memory. However, it cannot eliminate accessing the host memory. ❷ To alleviate the penalty of populating pages on the host, we introduce a large page eviction scheme, amortizing the cost of evicting multiple base pages. ❸ If accessing data in the host memory is inevitable, we exploit the parallelism with path diversity in the multi-GPU systems when handling page fault batches. This approach utilizes the individual PCIe lanes attached to the harvested GPU and the local GPU at the same time.

To further hide the latency of migrating data to local GPU memory, we introduce a new *multi-path parallel prefetcher* exploiting the harvested memory and the path diversity in the multi-GPU systems. ❹ The data residing in the spare memory is prefetched into the local GPU through NVLink. Meanwhile, we prefetch the data belonging to the host memory to either the spare memory ❺ or local GPU memory ❻, depending on the PCIe congestion. Specifically, when there are multiple harvesters exercising the same spare memory, the harvested GPU can receive excessive prefetching requests, which saturate the PCIe bandwidth and adversely delay transfers of prefetch data. To address the problem, we have a facility dynamically enabling and disabling the use of spare memory in prefetching data from the host memory. Also, memHarvester prioritizes the memory eviction demands over prefetching on the harvested area because serving memory evictions is on the critical path of handling page faults.

Each GPU process has a separate page table. Even though evicted pages are located in the neighbor GPU yielding spare memory, the process running in that neighbor GPU is not allowed to access the evicted pages. This is because the page table of the process in the neighbor GPU does not have the mapping information for the evicted pages, like the existing UVM driver [29]. Therefore, we do not violate the integrity and secrecy of evicted pages.

4.2 Hiding Eviction Latency to Host

Once the memory capacity is full, it is not allowed to bring data to the GPU before completing the memory eviction. The memory eviction is a part of the performance-critical path. Since migrating data to GPU memory is faster than to the host memory, our harvesting can reduce the latency of handling GPU page faults. After evicting the pages, memHarvester rapidly moves to the next step of fetching requested pages. While the harvesting GPU fetches the required pages, memHarvester invokes a background *writeback* thread to make a copy of the evicted page present in the harvested memory to the host memory. After copying pages, memHarvester marks the pages backed in host memory as `removable`. The purpose of the background copy is to immediately return the harvested space to the original GPU with negligible overhead. Once the application in the yielding GPU requires more memory than it currently has, it causes a GPU page fault. Then, memHarvester reclaims the harvested (removable) pages for the yielding GPU to use without the eviction. It picks the pages that come first into the yielding GPU. If there is no free or removable page, the yielding GPU may need to wait until the pages in the harvested region become removable. Although it can potentially incur the performance overhead, it rarely happens when evaluating our technique.

Therefore, our approach of using spare memory as a victim buffer allows that with a small fraction of memory, memHarvester turns the latency of host memory access into that of a neighbor GPU memory almost entirely, eliminating the host access latency from performance-critical eviction path.

4.2.1 Pre-eviction

When evicting pages to host memory, it is known that the pre-eviction scheme cannot hide the eviction latency entirely because the pre-eviction rate is limited to the PCIe bandwidth [25, 28]. In addition, pre-eviction requests contend with fetch requests occasionally, adding extra latency to critical fetch requests by stalling them. On the other hand, when evicting pages to harvested memory, it uses abundant NVLink bandwidth without contending the fetch requests from host memory. Once the memory consumption of harvesting GPU reaches a threshold of total physical memory (by default, if less than 50 free chunks are available), memHarvester invokes a pre-eviction thread. The pre-eviction has a good match with the background writeback technique because pre-eviction and writeback are pipelined. Ultimately, such pre-eviction allows

free memory available most of the time and effectively eliminates the eviction time from the GPU page fault latency. memHarvester uses the well-known LRU policy to select pages to be evicted. Pre-eviction with LRU policy works well with the memory access patterns of graphs and DNN workloads because most of them exhibit a cyclic memory access pattern with long reuse distances [2, 19, 35, 37, 38].

4.2.2 Large page eviction

The granularity for page faults is supposed to be the same size as the host architecture because the UVM driver relies on the demand paging scheme. On the other hand, the UVM driver uses a 2MB chunk as an eviction unit to simplify memory management. While evicting a 2MB chunk from the GPU to the host, the UVM driver splits the 2MB chunk into 512 4KB pages and performs the page population for the 512 pages because the driver is conservatively implemented to use the base page in the host architecture. To avoid such undesired inefficiency, memHarvester allocates 2MB of large pages in host memory by using the kernel's contiguous memory allocator [21]. Hence, memHarvester performs a single operation for populating a 2MB page between GPU and host memory instead of performing for individual 512 4KB pages. With or without our harvesting scheme, we can apply this large page eviction to all cases where a GPU has to interact with host memory.

4.2.3 Eviction policy

As multiple GPUs can leave a small amount of idle memory, as shown in Figure 1, memHarvester selects a target in a round-robin fashion to avoid hotspot contention and maximize the available GPU-to-GPU bandwidth in the system. Additionally, the round-robin policy enables each yielding GPU to make the removable pages in parallel through individual PCIe lanes. Eviction to the spare memory can incur performance interference to applications running on that yielding GPUs. We evaluate the negative performance impact of harvesting memory in Section 6.2.1.

When multiple harvesting requests are concentrated on a single spare memory, memHarvester handles the requests following their arrival orders. Note that the spare memory is used as a shared cache across harvesters.

4.3 Hiding Fetch Latency from Host

Evicting pages to the harvested memory reduces fetching latency if the local GPU accesses its evicted pages in a short period because memHarvester can fetch pages from the victim buffer in the neighbor GPU. In addition to that, memHarvester deploys two proactive schemes to hide fetch latency from the host memory with the limited space of harvested memory: fetching pages in parallel on page faults and pre-fetching pages with diverse paths.

4.3.1 Fetching pages in parallel

To reduce the cost of handling page faults, modern GPUs batch multiple page faults. The number of faults in a batch varies from time to time.[3] The UVM driver handles requested pages corresponding to the page faults one by one. On the other hand, with the availability of harvested memory, memHarvester parallelizes handling multiple page faults in the same batch. memHarvester invokes page fault handling threads for each GPU (i.e., harvesting GPU and yielding GPUs), dividing tasks to each handling thread. As shown in Figure 3 (❸), one thread for harvesting GPU takes faults from the head of the fault buffer, while the other thread for yielding GPU traverses the buffer from the tail and places the data corresponding to the fault on the harvested memory. memHarvester effectively reduces the latency of handling faults by fetching pages to local and the spare memory in parallel, so it utilizes the individual PCIe lanes attached to each GPU. Later, the fetched faults on the spare memory will be consumed through NVLink, reducing the fault latency further.

4.3.2 Multi-path parallel prefetcher

To hide the latency of accessing host and spare memory, we design a *multi-path parallel prefetcher* exploiting the path diversity in multi-GPU systems. When prefetching multiple memory chunks across the host and spare memory, there is no dependency between chunks. Our multi-path prefetcher can exploit the parallelism fetching the chunks with PCIe and NVLink. memHarvester first places the pages in the spare memory to the local GPU memory via NVLink (❹ in Figure 3). For the pages in the host memory, memHarvester prefetches them on either the spare memory (❺) or the local GPU memory through PCIe (❻). The policy paragraph below explains how to select the target memory when prefetching from the host dynamically.

For selecting candidates to be prefetched, memHarvester uses a simple yet effective next line and stride prefetchers, which are good enough for graph analytics [2, 19, 38] and DNN training workloads [7, 32, 33, 35, 37]. By extracting the memory access pattern from the page fault history, memHarvester prefetches the next couple of chunks from either the host memory or the harvested memory, depending on where the chunks are located. We empirically select the amount of prefetch as 32MB and will show the sensitivity study in Section 6.2.1.

4.3.3 Prefetch policy

Unless the PCIe lane attached to the spare memory is crowded, we observe that prefetching to the spare memory can further reduce the fetch latency compared to the prefetch directly from the host to the local GPU. This is because leveraging

[3] The stock UVM driver handles up to 128 faults in a batch.

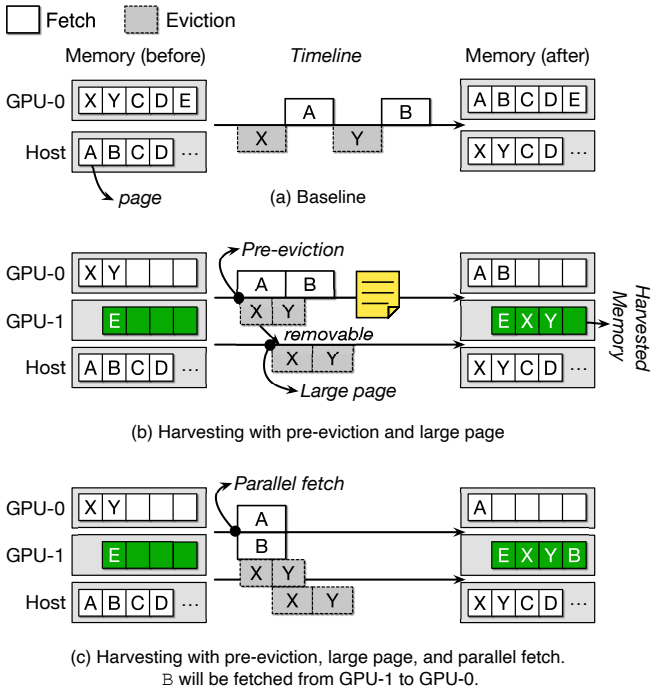

Figure 4: Timeline comparison for pre-eviction, large page, and parallel fetch (Suppose that GPU-0 is harvesting the idle memory of GPU-1)

spare memory can reduce both 1) the number of page faults by proactively fetching pages and 2) the page fault latency by placing the pages highly likely to be accessed on the spare memory. On the other hand, as the number of active harvesters increases, the PCIe lane attached to the spare memory can be congested. Then, it slows down supplying the pages to the spare memory due to the limited PCIe bandwidth, increasing the fault latency.

To deal with diverse harvesting scenarios in multi-GPU servers, we have a policy in prefetching to dynamically select where the data in the host memory to be prefetched to either the spare memory (❺) or the local GPU memory (❻) based on the number of active harvesters.

4.4 Putting It All Together

Suppose the application running on the GPU-0 is accessing `page A` and `B` on the host memory. The GPU-0 memory is fully occupied except for the small number of reserved pages that hide memory eviction time in handling page faults. Figure 4a and Figure 4b compare how memHarvester eliminates the memory eviction latency from the critical path. While handling the page faults, if the number of the reserved chunks falls below the threshold (set as 50 chunks), memHarvester triggers the pre-eviction task to secure the free space for upcoming page faults without the memory eviction. In this example, the oldest `page X` and `Y` in the GPU-0 are evicted

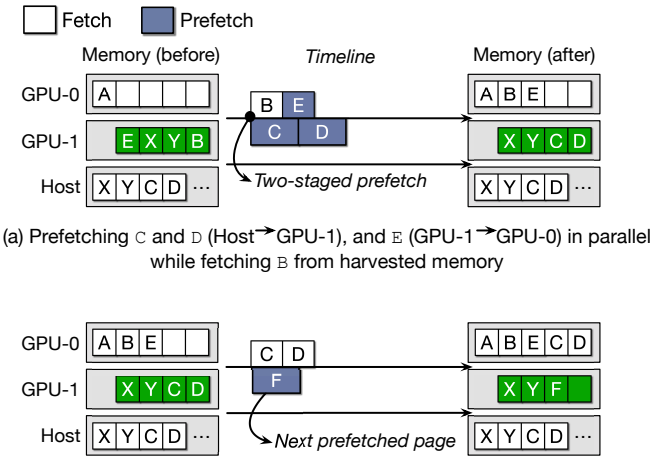

Figure 5: Timeline for our multi-path parallel prefetcher

to the harvested memory of the GPU-1 as background. Once it is completed, memHarvester makes a copy of that pages to the host and marks them as `removable`.

While handling the fault for `page A`, memHarvester looks up the faulted pages belonging to the same batch group. Figure 4c presents that `page A` and `B` are in the same batch. Thus, the fault batch can be processed to the local and harvested memory in parallel. While fetching `page A` to the local GPU memory, memHarvester places `page B` on the harvested memory. Since each GPU has its PCIe lane, we can fetch `page A` and `B` in parallel. Although it cannot reduce the number of page faults, we can hide the latency for fetching `page B`. Eventually, it does reduce the fault latency by fetching `page B` from the harvested memory connected through NVLink, rather than the host via PCIe.

Since the `page A` and `B` are faulted in order, memHarvester prefetches the next `page C, D,` and `E`. Assume that the number of pages to be prefetched is three in this example. Figure 5a depicts how the multi-path parallel prefetcher works. As `page C` and `D` are in the host memory, those can be prefetched into either the harvested memory or the local GPU memory through the PCIe lane. In this example, we assume that the PCIe lane is not contended so that memHarvester selects the harvested memory as the target. On the other hand, `page E`, which is in the harvested memory, is prefetched to the local GPU memory in parallel via NVLink. It can eliminate the fault if `page E` is accessed from the application. Figure 5b presents that we can potentially reduce the fault latency for `page C` and `D` by fetching such pages from the harvested memory when they are accessed.

5 Implementation

We implement our prototype, memHarvester, in the NVIDIA UVM driver version 460.67. The modification of the UVM

driver is 1,838 lines of C code measured by SLOCCount. We do not require any modifications to runtime and frameworks.

5.1 Managing Spare Memory

As UVM, memHarvester manages GPU physical memory as 2MB chunks. Each chunk has metadata, which describes the states and physical address of the 2MB chunk. For each GPU, memHarvester maintains a linked list of the metadata for free 2MB chunks. By referencing the free list of GPUs, memHarvester can easily extract available spare memory in the system. Once a GPU harvests a neighbor GPU's memory, memHarvester marks the metadata to indicate that a 2MB chunk is harvested from the neighbor GPU.

5.2 Managing GPU Memory Eviction

To evict a chunk, memHarvester selects the oldest chunk from per-GPU LRU[4] lists as the stock UVM. The pre-eviction path diverges depending on the availability of harvested memory. If memHarvester has harvested memory, the background thread evicts chunks to the harvested memory. Otherwise, it evicts chunks to host memory. When memHarvester evicts a chunk to the harvested memory, memHarvester moves the chunk metadata from the LRU list to the evicted list. Since the evicted chunk resides only on the harvested memory, memHarvester does not allow the chunk to be reclaimed by the harvestee. Once the eviction to the harvested memory is completed, memHarvester invokes a writeback thread to duplicate the chunk in the harvested memory to host memory. After that, memHarvester marks the chunk as removable and moves the chunk to the removable list. When the harvested memory has to be returned to the original GPU, memHarvester reclaims the removable chunks first. Before completing the writeback task, we are not able to reclaim the harvested space for serving the other request. Thus, the throughput of the writeback thread is critical. memHarvester increases the eviction size with large page to accelerate writeback thread and eviction.

5.3 Managing Fetch Requests

When a page fault exception occurs, the fault exception handler supplies faults from the head of the fault batch as usual. At the same time, memHarvester wakes up another kernel thread to serve fault entries from the tail of the fault batch to the harvested memory in parallel. As a result, we spend less time completing the fault batch with harvested memory. To coordinate consuming the fault batch shared between two threads, we use the mutex lock to handle a fault entry from the buffer synchronously.

After handling the demand fault batch, memHarvester triggers our multi-path prefetcher utilizing both the PCIe and NVLink bandwidth. memHarvester employees a simple but effective next-line prefetcher for capturing the memory access pattern observed in graph analytics and DNN training. It keeps track of the addresses for the faulted pages to identify the direction of previous page fault addresses. Once the direction is determined, memHarvester prefetches the next 32MB as default. For the selected chunks, we examine the metadata to filter out the chunks already in the local GPU. We separate the selected chunks into two different queues according to their origin, either the host or the harvested memory, and then conduct the multi-path prefetch. While prefetching the chunks in the host to the harvested memory through PCIe, we use a kernel thread to prefetch the other chunks in the harvested memory to the local GPU via NVLink. We mark prefetched chunks as removable so that it can be immediately reclaimed when needed. If we encounter on-demand faults while prefetching, we abandon the ongoing prefetches to serve the demand faults first.

[4]The LRU term presents the least recently swapped-in page.

6 Evaluation

6.1 Experimental Setup

To evaluate the effectiveness of memHarvester, we conduct performance comparisons with the stock version of the unified virtual memory (`Base`) and the prior approach employing the pre-eviction and prefetch techniques for the host memory [11, 14, 17, 25, 28] (`Pre-ef-host`). As the implementation of prior studies is not publicly available, we imitate the behavior of pre-eviction and prefetch on top of the stock UVM driver. The evaluation is performed on an AWS `p3.8xlarge` which has four NVIDIA V100 GPUs, each with 16GB of memory. These four GPU cards are connected to each other through NVSwitch and NVLink 2.0 [16] and 240GB of host memory is attached through PCIe 3.0.

6.2 Experimental Results

6.2.1 Inter-job Harvesting

First, we evaluate our scheme in a shared multi GPU server hosting multiple DNN training and graph analytics workloads. We show how the idle memory of GPUs can be harvested by memory-intensive workloads running on other GPUs with inter-job harvesting. In this evaluation, all training workloads run with PyTorch (version 1.10.1), and all graph analytics workloads run with cuGraph (version 21.12).

Performance improvement. We evaluate the execution time of multiple workloads in four scenarios by varying the type of jobs and the number of harvesters to mimic a shared multi-GPU server environment. Table 2 presents the scenarios and the memory usage ratio for each job according to the given graph dataset or batch size. Figure 6 shows the speedup of the execution time to `Base` and also measures the performance

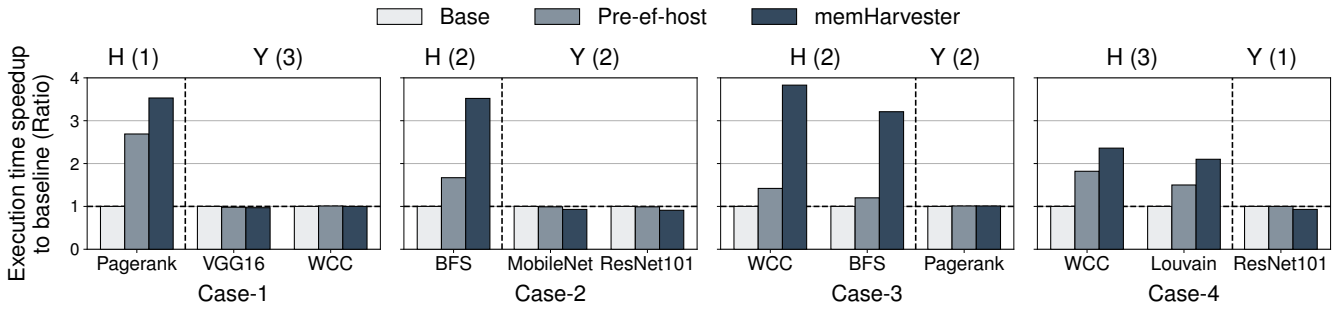

Figure 6: Execution time speedup of memory intensive workloads with `memHarvester` on four different harvesting scenarios (`H`: Harvesting GPU, `Y`: Yielding GPU, and the numbers in parentheses indicate the number of participating GPUs)

	GPU-0	GPU-1	GPU-2	GPU-3
Case-1 (ratio)	Pagerank (1.87x) soc-twitter-2010	VGG16 256 (0.96x)		WCC (0.79x) soc-sinaweibo
Case-2 (ratio)	BFS (1.51x) web-uk-2005		MobileNet 256 (0.91x)	ResNet101 64 (0.67x)
Case-3 (ratio)	WCC (1.77x) soc-twitter-2010	BFS (1.70x) soc-twitter-2010	Pagerank (0.74x) web-Clue09-50m	
Case-4 (ratio)	WCC (1.77x) soc-twitter-2010	Louvain (1.47x) web-uk-2005		ResNet101 64 (0.74x)

Table 2: Multi-job scenarios with memory usage ratio in parentheses, input graph, and batch size (Gray cell: harvester)

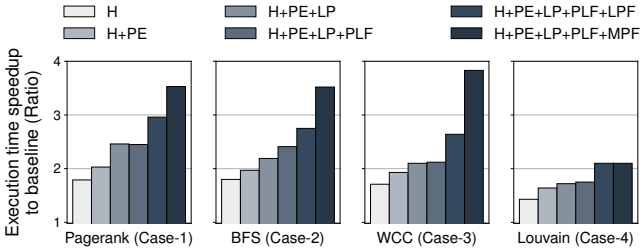

Figure 7: Effectiveness of individual techniques

of `Pre-ef-host` (i.e., the prior approach [17, 28]) for performance comparison. For all the cases, the execution time of harvesters, which do not fit on one or more V100 GPU memory (16GB), can be significantly improved by harvesting the idle memory of neighbor GPUs connected through NVLink.

In `Case-1`, `Pagerank` running on GPU-0 benefits from the spare memory of GPU-1 and 2 where `VGG16` is running in data-parallelism and spare memory of GPU-3 where `WCC` is running, leading to 3.53× and 1.31× improvement over `Base` and `Pre-ef-host`, respectively. The amount of total spare memory of GPU-1, 2, and 3 is around 4.64GB, which is much smaller than the overcommitted memory of 13.92GB by `Pagerank`. Although the spare memory is unable to serve all the evictions from `Pagerank`, our memHarvester effectively harvests the spare memory to hide the latency of accessing the host. In addition, the negative performance impact to the applications running on the yielding GPUs is negligible.

We also evaluate the performance changes by varying the number of harvesting and yielding GPUs. In `Case-2`, `BFS` running on GPU-0 and 1 harvests the spare memory of GPU-2 and 3 where `MobileNet` and `ResNet101` are running in each of the GPUs. memHarvester considers that the separated spare memory is logically unified like a shared cache across the harvesters. Our memHarvester can increase the performance of `BFS` by 3.52× and 2.1× compared to `Base` and `Pre-ef-host`, respectively. While harvesting, the performance impact of yielding GPUs is around 7~9%.

We show a different scenario in `Case-3` where two applications each of which runs on a single GPU contend for the idle memory of neighbor GPUs. `WCC` running on GPU-0 and `BFS` running on GPU-1 harvests the spare memory of GPU-2 and 3 where `Pagerank` is running with its graph partitioned across two GPUs. Our memHarvester can increase the performance of `WCC` by 3.83× and 2.71×, and that of `BFS` by 3.21× and 2.67× compared to `Base` and `Pre-ef-host`, respectively. There is no performance degradation in `Pagerank` running on the yielding GPU.

In `Case-4`, we evaluate the effectiveness of our approach when two workloads share a limited spare memory contributed by a single GPU. `ResNet101` yields 4.16GB idle memory that is shared by `WCC` running on GPU-0, and `Louvain` running on GPU-1 and 2. Although the throughput improvement is not much compared to other cases, it still outperforms `pre-ef-host` by 30~40%. Compared to `Base`, the harvesters show around 2.1~2.36× improvement while the performance impact of yielding GPU is around 7%.

Analysis of performance improvement. We decompose the contribution of the performance improvement to individual schemes constituting memHarvester. To this end, Figure 7 shows performance changes for each workload (from the left to the right) while we enable spare memory harvesting (`H`), pre-eviction (`PE`), large page support (`LP`), parallel fetch (`PLF`), local prefetcher (`LPF`) and multi-path parallel prefetcher (`MPF`) in order. Note that local prefetcher (`LPF`) is not a scheme of

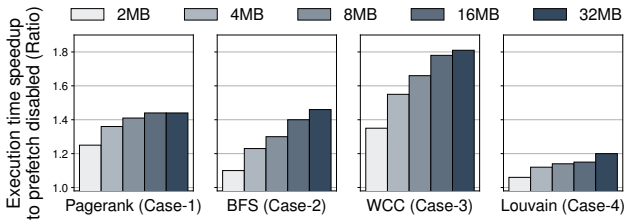

Figure 8: Sensitivity to the amount of prefetch

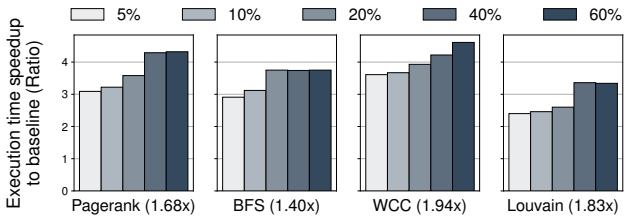

Figure 9: Sensitivity to the size of spare memory (The numbers in parentheses indicate the overcommitment ratio)

memHarvester and it is evaluated as a baseline to compare the performance gain of our multi-path parallel prefetcher (`MPF`).

In general, we observe higher performance gain while we enable each scheme one by one. This is because each scheme has its complementary benefit to memHarvester: i) spare memory harvesting (`H`) utilizes the spare memory as an eviction buffer and a victim cache to reduce the latency of migrating chunks by using NVLink rather than PCIe; ii) pre-eviction (`PE`) eliminates the eviction latency from the critical path by reducing on-demand page faults; iii) large page support (`LP`) reduces the time of making removable pages by writing back the chunks to host in batch; and iv) parallel fetch (`PLF`) reduces the latency of handling on-demand page faults by fetching the pages in the fault batch in parallel with both PCIe and NVLink. While the first three schemes (`H`, `PE`, `LP`) focus on optimizing the eviction penalty, parallel fetch (`PLF`) focuses on reducing the latency when handling page faults.

Finally, we investigate our multi-path parallel prefetcher (`MPF`) which drastically improves performance compared with local prefetcher (`LPF`). For `Case-1, 2,` and `3`, some of the evicted data reside in the host memory due to the lack of aggregated idle GPU memory in the system. Our multi-path parallel prefetcher utilizes the PCIe of the yielding GPU to prefetch data in host to the harvested memory. For `Case-4`, however, our multi-path parallel prefetcher (`MPF`) selects to prefetch data in the host memory to the local GPU memory rather than the harvested memory to avoid contention in the PCIe lane attached to the yielding GPU. Because there is more than one harvester per one spare memory, multi-path parallel prefetcher (`MPF`) changes the policy to directly fetch data from the host to local GPU memory. Thus, multi-path parallel prefetcher (`MPF`) has no performance gain compared to local prefetcher (`LPF`) in `Case-4`.

Sensitivity study. In this subparagraph, we present the sensitivity study of memHarvester to examine three aspects.

① **Prefetch size and stride:** We evaluate the sensitivity study for our next line and stride prefetches used in `multi-path parallel prefetcher`. Figure 8 depicts the execution time improvement by varying the amount of prefetches from 2MB to 32MB with a 2MB stride. Since both graph analytics and DNN training workloads exhibit sequential access patterns during the execution [2, 19, 35, 37, 38], our prefetcher extracts the direction of accessing the memory address and issues the prefetch requests for the next chunks to either the host memory or the spare memory depending on where the selected chunk is located (host to spare and spare to local). While evaluating our prefetcher, we observe that the 2MB stride shows better throughput than the next line (0MB stride). For all four cases, increasing the amount of prefetch steadily improves the performance. We select 32MB as our default prefetch size with a stride of 2MB. We will further investigate how the amount of prefetch can be dynamically adjusted for maximizing the prefetch effects in our future study.

② **Available spare memory:** In a shared multi-GPU server, we expect that the available memory for harvesting fluctuates. We evaluate the execution time of four graph analytics workloads by manually varying the amount of spare memory from 5% to 60%. We imitate a scenario with 2 GPUs with one GPU running a memory-intensive graph analytics workload and the other GPU yielding spare memory with an appropriate size of `cudaMalloc`. Figure 9 exhibits that the performance can be improved by harvesting more idle memory of neighbor GPUs. Interestingly, even with 5% (800MB) of spare memory, we can achieve more than 2× improvement for all four workloads. By effectively managing the small amount of idle memory of a neighbor GPU, the performance improvement is significant. However, when a certain amount of spare memory is harvested to accommodate all the active working sets to fit in the local GPU with the harvested memory, maximum performance is achieved and there is no additional improvement even if we increase the amount of spare memory. We observed that the amount of overcommitment size is larger than the active working set. For `Louvain` with an overcommitment ratio of 1.83×, the maximum performance is achieved when the spare memory size is 40% and even if the spare memory size is increased, there is no additional performance gain. We found a discrepancy between the size of the active memory working set and the size of memory `malloc'ed` by the user-level and will further investigate whether we can also harvest the unused memory space that is not included in the active working set in our future study.

③ **Performance interference:** While harvesting spare memory, our approach can cause performance interference to the applications running on GPUs that yield the spare memory. This is because the harvesters piggyback the

	ResNet101			
	Base	H	H+PE+LP	memHarvester
VectorAdd (1)	1	0.99	0.99	0.97
VectorAdd (3)	1	0.91	0.88	0.87

Table 3: Normalized execution time of ResNet101 by increasing the number of VectorAdd harvesters

memory and PCIe bandwidth of yielding GPUs. Table 3 presents the impact of performance interference through a VectorAdd microbenchmark designed to generate a significant memory harvesting traffic. By increasing the number of VectorAdd harvesters, we measure the performance of ResNet101, which yields idle memory on one GPU. To investigate the interference incurred by individual schemes, we decompose our schemes into harvesting only (H), with pre-eviction and large page (H+PE+LP), and with all the prefetchers (memHarvester). When running with a single harvester, the impact of performance interference for ResNet101 is negligible, up to 3% compared to the baseline. The maximum pressure to the GPU yielding idle memory is bounded by the network bandwidth of two GPUs through NVLink.

Meanwhile, three harvesters reduce performance by 13%. Even with harvesting only (H), it shows 9% performance degradation. As the number of harvesters increases, the amount of memory traffic to the idle memory can also increase, leading to the memory bandwidth contention. When enabling pre-eviction and large page (H+PE+LP) schemes, it can utilize the idle memory more effectively, but it poses an additional 3% overhead. When all the proposed schemes are applied (memHarvester), the performance interference is not much different. Since our multi-path parallel prefetcher (MPF) selects to prefetch data in the host memory to the local GPU memory rather than the harvested memory to avoid contention in the PCIe lane attached to the yielding GPU. We anticipate that throttling the harvesting traffic can reduce the performance interference though it decreases the benefits to the harvesters. We leave this optimization to future work.

6.2.2 Intra-job Harvesting.

Our memory harvesting technique can be applied to multi-GPU applications where the memory consumption of individual GPUs is not even. The representative example is multi-GPU DNN training exploiting pipeline parallelism [20]. Such memory usage imbalance is primarily due to non-identical model partitions placed across the GPUs to balance the computation of each partition, leading to different memory demands (e.g., some of the GPUs need to have multiple weight and activation versions in pipeline parallelism). We evaluate the effectiveness of HUVM with memHarvester for single DNN training jobs. For this evaluation, we select GNMT16, GNMT8, ResNet50, and VGG16 with PyTorch. For the baseline,

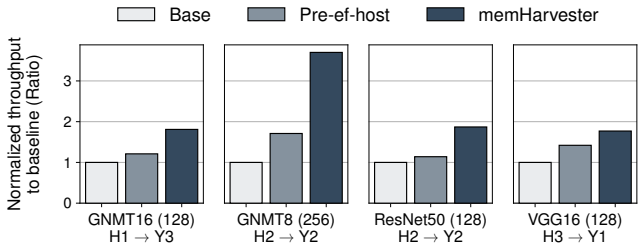

Figure 10: Throughput improvement for single training workloads (H: # harvesting GPU and Y: # yielding GPU)

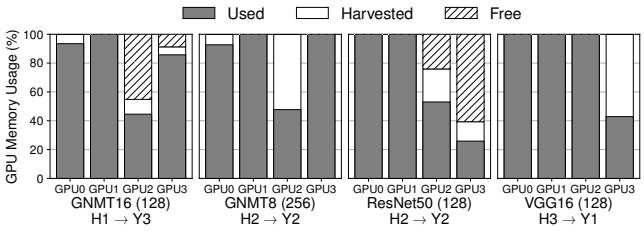

Figure 11: Memory usage for single training workloads (H: # harvesting GPU and Y: # yielding GPU)

the PyTorch framework is modified to support the memory oversubscription with NVIDIA UVM.

Performance improvement. Figure 10 shows the results of throughput comparison among memHarvester, Pre-ef-host, and Base. To initiate memory oversubscription, the batch size in each model is chosen such that at least one of the four GPUs goes beyond the local memory capacity. The figure shows that for all the models, memHarvester outperforms Pre-ef-host. In particular, for GNMT16 and ResNet50, memHarvester can effectively eliminate the host memory accesses in the increased batch size by harvesting only idle memory of neighbor GPUs. This leads to the throughput in memHarvester 1.5~1.6× higher than that in Pre-ef-host for the two models.

Figure 11 shows the memory profiles of the four models across the individual GPUs. It is worth noting that model training under memHarvester achieves throughput gains via diverse memory harvesting paths. For GNMT16, GPU-1 is the only GPU harvesting the spare memory of GPU-0, 2, and 3. On the contrary, for ResNet50, GPU-0 and 1 are the two harvesting GPUs that utilize the spare memory of the other two yielding GPUs, i.e., GPU-2 and 3.

More interestingly, for GNMT8 and VGG16, the aggregated idle memory across the yielding GPUs is not sufficient to serve the data evicted from the harvesting GPUs. In VGG16 with batch size 128, GPU-0, 1, and 2 use up all the idle memory of GPU-3 and then require using the host memory additionally. In spite of exercising the host memory, memHarvester shows meaningful throughput gains for both workloads. memHarvester improves throughput over Pre-ef-host by 2.16× and 1.24× for GNMT8 and VGG16, respectively.

7 Related Work

To the best of our knowledge, memHarvester is the first to propose a framework that allows GPU applications to utilize neighbor GPU's memory connected through the high-speed interconnect (NVLink). There have been significant efforts in both the architecture and systems community to support GPU memory oversubscription while minimizing the performance degradation of applications. Demand paging on GPUs [3, 22, 24] allows the GPUs to move pages from the GPU's memory to/from the CPU's memory automatically. We survey recent techniques that provide mechanisms to allow applications with a large working set to run on GPUs.

Framework-guided approach. For deep learning (DL) training workloads, prior studies proposed to have the framework insert the pre-eviction and pre-fetch operations by analyzing the dataflow graph [11, 17, 28]. Peng et al. introduced a sophisticated technique employing the pre-fetch and recomputation opportunistically without relying on the dataflow graph [25], these also require the intensive framework modification in terms of tensor allocation. Besides the pre-fetch technique, Animesh et al. proposed to compress the data, which shows the long reuse distance to save memory while the data is not being actively used [12]. Although this design approach can be effective, it requires the framework modification and understanding of the target applications, incurring the engineering overhead. Unlike such previous studies, we design and implement our solution in the GPU driver which can coordinate all the GPU memory in a centralized way.

Architectural approach. The architecture community also explored hardware techniques to minimize the overhead of GPU memory virtualization. Recently, Choukse et al. explored the advantage of leveraging the neighbor GPU memory which is connected through the high-bandwidth interconnect (NVLink). Unlike our approach, they studied a HW-based compression scheme to squeeze the limited neighbor GPU memory [5]. Ganguly et al. studied the prefetch technique used in the NVIDIA UVM driver in the overcommitted environment and proposed a HW-based pre-eviction and pre-fetch techniques [8]. Kim et al. exploited that modern GPUs handle the page faults in a batch and co-designed the GPU runtime and hardware to overlap the page eviction and migration effectively [14]. Li et al. proposed a framework for memory overcommitment [17] to efficiently virtualize GPU memory, but the disadvantage of those studies requires significant changes to the GPU runtime and hardware. Although such hardware approaches can further improve the performance as well as the efficiency, it requires new hardware structures.

Memory compression. Many previous studies [5, 15, 17, 27, 30, 34] proposed techniques to perform memory compression. While these techniques allow more data on the GPU memory, they are orthogonal to HUVM and can be used in conjunction to further improve the effectiveness of our proposal.

8 Conclusion

In this study, we propose a new approach of virtualizing multi-GPU memory, hierarchical unified virtual memory, by dynamically incorporating the spare memory of neighbor GPUs in multi-GPU systems. This can alleviate the memory fragmentation problem by creating the illusion of GPU applications having an increased effective memory space. To effectively utilize the small fraction of neighbor GPUs' memory, we introduced a memory manager for multi-GPU systems that has a set of techniques, including large page support, parallel fetch, and multi-path parallel prefetcher. Since our techniques can effectively reduce the latency of accessing host memory, for memory-intensive workloads, throughput performance is significantly improved compared to baseline and prior studies based on pre-eviction and prefetch techniques.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable comments and feedback. This research is supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(2021-0-02051) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation) and Electronics and Telecommunications Research Institute(ETRI) grant (22ZS1300). This work is also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2020R1C1C1014940).

References

[1] B. Acun, M. Murphy, X. Wang, J. Nie, C. Wu, and K. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[3] AMD. Radeon's Next-generation Vega Architecture. https://en.wikichip.org/w/images/a/a1/vega-whitepaper.pdf, 2017.

[4] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. Mcm-gpu: Multi-chip-module gpus for continued performance scalability. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[5] E. Choukse, M. B. Sullivan, M. O'Connor, M. Erez, J. Pool, D. Nellans, and S. W. Keckler. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[6] cuGraph. GPU Graph Analytics. https://github.com/rapidsai/cugraph.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv:1810.04805*, 2018.

[8] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019.

[9] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A gpu cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

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

[11] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[12] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[13] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019.

[14] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[15] J. Kim, M. Sullivan, E. Choukse, and M. Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[16] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Trans. Parallel Distributed Systems (TPDS)*, 31(1), January 2020.

[17] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A framework for memory oversubscription management in graphics processing units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[18] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[19] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. Emogi: Efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 2020.

[20] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[21] Michal Nazarewicz. A deep dive into cma, Mar. 2012. https://lwn.net/Articles/486301/.

[22] NVIDIA. NVIDIA Tesla P100 P100 GPU Architecture. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[23] NVIDIA. Nvidia dgx-2: The world's most powerful ai system for the most complex ai challenges., 2019. https://www.nvidia.com/en-us/data-center/dgx-2/.

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

[25] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[26] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[27] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. Keckler. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[28] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Processing of the 49th International Symposium on Microarchitecture (MICRO)*, 2016.

[29] Nikolay Sakharnykh. Maximizing unified memory performance in cuda, Nov. 2017. https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/.

[30] V. Sathish, M. Schulte, and N. Kim. Lossless and Lossy Memory I/O Link Compression for Improving Performance of GPGPU Workloads. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[31] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten Schwan. Graphreduce: processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[34] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. Mowry, and O. Mutlu. A Case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling Flexible Data Compression with Assist Warps. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[35] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[36] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[37] Peifeng Yu and Mosharaf Chowdhury. Fine-grained GPU sharing primitives for deep learning applications. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

[38] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. Making caches work for graph analytics. In *Proceedings of the IEEE International Conference on Big Data (Big Data)*, 2017.