2022 IEEE International Symposium on Workload Characterization (IISWC) | 978-1-6654-8798-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/IISWC55918.2022.00013

UVM Discard: Eliminating Redundant Memory Transfers for Accelerators

Weixi Zhu[1] Guilherme Cox[2] Jan Vesely[2] Mark Hairgrove[2]
Alan L. Cox[1] Scott Rixner[1]

[1] {wxzhu, alc, rixner}@rice.edu, Rice University
[2] {gcox, jvesely, mhairgrove}@nvidia.com, NVIDIA

Abstract

An increasing number of applications benefit from heterogeneous hardware accelerators. Such accelerators often require the application to manually manage memory buffers on devices and transfer data between host and device buffers. A programming model that unifies the virtual address space across the host and devices is appealing because it enables automatic memory transfers and simplifies application-level programming. However, the automatic memory transfers can sometimes be redundant, which decreases performance. NVIDIA's UVM (unified virtual memory) driver provides a unified virtual address space for CPU-GPU programming. This paper identifies redundant memory transfers (RMTs) as a common performance issue with UVM. To address this issue, this paper proposes a data discard directive, and evaluates two implementations of that directive, UvmDiscard and UvmDiscardLazy. This directive exploits application-level knowledge to avoid RMTs. The implementations were integrated with NVIDIA's open-source UVM driver to demonstrate their usefulness on real-world CUDA UVM applications. For example, the use of the discard directive increases training throughput by 61.2% on a large deep learning application that oversubscribes GPU memory.

1. Introduction

Heterogeneous computation with specialized accelerators – including GPUs, TPUs and FPGAs – is becoming ubiquitous in recent HPC platforms. For example, seven of the recent TOP10 supercomputers use NVIDIA's discrete general-purpose graphics processing units (GPGPUs) [8]. These GPUs provide massive thread-level parallelism and offer much higher memory bandwidth compared with CPUs. Furthermore, they have specialized hardware units to accelerate critical computing operations to further benefit scientific computation, machine learning and other domain-specific applications [9].

Unfortunately, accelerators still suffer from poor programmability which potentially undermines the efficient utilization of these petascale systems. While parallel programming frameworks including CUDA, OpenCL and OpenMP have simplified the writing of accelerator code, one of the biggest programming challenges remains – applications must carefully manage data placement between accelerators and the host in a correct and efficient manner. Accelerator memory capacities are much smaller than the host memory capacity, and are often smaller than the datasets upon which they are capable of operating. This requires applications to first allocate memory buffers on the accelerator and transfer data to those buffers before initiating a computation kernel. Once the computation is complete, the application must then coordinate the transfer of the results back to the host memory and potentially deallocate memory buffers. Furthermore, pointers cannot be transferred back and forth between the host and accelerator as such pointers will only be valid on the host or accelerator, but not both. Parallel programming frameworks provide various degrees of support for these operations, but generally the application must be aware of them all.

Such application-level manual memory management is difficult and error-prone. The host buffers and device buffers need to be simultaneously managed in both places, and the memory transfers must be manually orchestrated along with heterogeneous computations. Furthermore, such memory management is even more complex when the application must work across a family of accelerators (i.e., different GPU versions) that have differing memory capacities. The application then must manually manage the buffer sizes given the capacity of the specific accelerator installed in the system. In practice, many applications simply target a specific memory size. They require that size as a minimum and are unable to effectively use excess memory on larger devices. More sophisticated applications employ much more complex memory management strategies in which device buffers are manually migrated back and forth so as to enable the application to work with limited memory resources on the accelerator.

The use of a shared virtual address space can help alleviate some of these issues by enabling the system to automatically migrate memory when necessary. NVIDIA has implemented one such shared virtual address space system, called unified virtual memory (UVM) [5]. In UVM, the address space is shared across the host and devices, even when the devices are connected with a non-cache-coherent interconnect. This means that pointers are valid everywhere and the system can orchestrate data transfers between the host and devices without application intervention. The UVM

Weixi completed this work as an intern at NVIDIA mentored by Guilherme Cox.

978-1-6654-8798-6/22/$31.00 ©2022 IEEE
DOI 10.1109/IISWC55918.2022.00013

programming model frees the application and framework from specifically managing buffers on the GPU, greatly simplifying programming and enabling applications to readily take advantage of differing memory capacities across devices.

NVIDIA's UVM system by default tries to cache hot data in the GPU DRAM and uses CPU DRAM as swap space. This strategy is typically beneficial because most applications exhibit data locality. Memory accesses to remote data trigger host/device page faults which initiate memory transfers. With cache-coherent interconnects, the UVM system may switch to using remote memory accesses to benefit rare cases with poor data locality. While the fault-driven transfers lead to large latency spikes, the remote-access mode cannot trigger desirable data transfers. Consequently, the UVM programming model offers an explicit prefetch command. Explicitly prefetching data in an effective manner does add some complexity to the programming model. However, these prefetches are a performance optimization and are not required for correctness.

The UVM system allows applications to operate on datasets that exceed the GPU memory capacity without any extra application-specific code. This greatly simplifies the programming effort. Without UVM, more than 2,000 extra lines of application-specific code are required to support large training sizes in deep learning [11, 12]. However, using the UVM system comes at a price. That price, which this paper is the first to identify, is a ubiquitous performance issue introduced by UVM on applications that cannot fit in a GPU's memory – redundant memory transfers (RMTs). RMTs are automatic memory transfers orchestrated by the UVM system that are not needed for correctness. They arise due to a semantic gap between the UVM system and the user program. For example, when a buffer is transferred but then overwritten before being read, that transfer was redundant.

In addition, this paper makes the following contributions. First, it characterizes the RMTs that arise in real-world GPU applications. Second, it proposes a novel memory discard directive that bridges the semantic gap. Specifically, this directive allows the user to inform the UVM system that the data contained within a specified memory region will no longer be used, so the data does not need to be transferred for correctness. Third, this paper evaluates two implementations of the discard directive within NVIDIA's open-source UVM driver, *UvmDiscard* and *UvmDiscardLazy*. *UvmDiscard* caters to ease of use by the application programmer, but this comes at the cost of higher run-time overhead due to current GPU hardware limitations. In contrast, *UvmDiscardLazy* overcomes these hardware limitations, but compromises programmability. *UvmDiscardLazy* demonstrates the potential benefits of enhancing the GPU hardware, which would allow the ease of use of *UvmDiscard* with the performance of *UvmDiscardLazy*.

For a GPU database application with a data size twice the GPU memory, UvmDiscard enables a 4.17 times speedup by eliminating 85.8% of memory transfers. Similarly, with the addition of only tens of lines of application code, UvmDiscard can eliminate up to 60.9% of memory transfers by a compute-intensive recurrent neural network leading to 22.8% higher training throughput, and also decrease memory transfers by 60.6% on a memory-intensive convolutional neural network resulting in 61.2% higher training throughput.

```
malloc(...); // Allocate host buffers h_A, h_B, h_C
cudaMalloc(...); // Allocate device buffers d_A, d_B, d_C
Generate input data for h_A, h_B
cudaMemcpyAsync(...); // Copy input to d_A, d_B
Launch GPU code: vectorAdd(d_A,d_B,d_C)
cudaMemcpyAsync(...); // Copy output to h_C
cudaDeviceSynchronize();
print(h_C);
```

Listing 1: CUDA VectorAdd example

```
cudaMallocManaged(..); // Allocate UVM buffers A, B, C
Generate input data for A, B
cudaMemPrefetchAsync(..); // Prefetch A,B and prefault C
    on GPU (optional)
Launch GPU code: vectorAdd(A,B,C)
cudaMemPrefetchAsync(..); // Prefetch C to CPU (optional)
cudaDeviceSynchronize();
print(C);
```

Listing 2: UVM VectorAdd example

2. Background

2.1. NVIDIA's UVM programming model

Listing 1 shows a CUDA example that calculates the sum of two vectors. In this example, manual buffer management is required for separate device and host buffers. In addition, explicit data marshaling and copying must be performed and coordinated with host and device computation.

NVIDIA's UVM simplifies the CUDA programming model by providing a unified virtual address space between the host and device. As shown by Listing 2, the program is no longer required to manage separate device buffers or insert explicit memory transfer commands. The pointers to the UVM buffers are valid for both host code and device code, and memory transfers are orchestrated automatically driven by page faults.

UVM allows the user to optionally issue prefetch operations, so that data transfers can be better overlapped with computation. The prefetch command "pre-faults" the virtual memory region. If the UVM buffer has never been touched, then the buffer will be mapped to zero-filled physical memory on the destination processor. If, however, the buffer was previously mapped and used, then it will be migrated to the destination processor. Therefore, future accesses from the destination processor become local accesses and will not trigger page faults. This makes prefetching desirable for GPU buffers, because GPU page faults can significantly hinder the thread-parallelism of GPU kernels.

2.2. An overview of NVIDIA's UVM driver

NVIDIA's UVM driver requires a GPU that supports page faults, so that will be assumed for the remainder of the paper. By default, NVIDIA's UVM driver treats the GPU's DRAM as a cache and tries to optimize data locality for

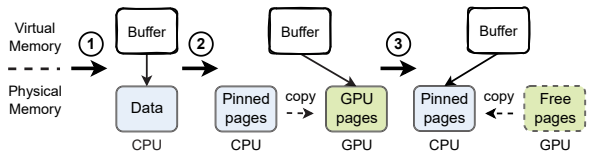


Figure 1: A typical lifetime of NVIDIA's UVM buffer.

the device code. Figure 1 illustrates the typical lifetime of a UVM buffer allocated by the CUDA API call named `cudaMallocManaged`: ①The buffer is initially mapped to zero-filled CPU pages when the host code triggers CPU page faults while writing the initial data to the buffer. ②The buffer is migrated to GPU pages when a CUDA prefetch is issued or the device code triggers GPU page faults while accessing the buffer. The CPU pages remain pinned while the buffer is mapped to GPU pages. ③The buffer is migrated back to CPU pages when a CUDA prefetch is issued, the host code triggers CPU page faults by accessing the buffer, or the eviction process wants to make room to cache other buffers on the GPU. The GPU pages are reclaimed after the buffer is migrated back to the CPU.

To summarize, the UVM driver enables fault-driven data locality by maintaining coherent page tables among CPU and GPUs, where a physical page is exclusively mapped by one of the page tables. Therefore, besides migrating data with CUDA prefetch commands, any memory accesses to a virtual address mapped by a remote page table will trigger page faults that perform automatic memory transfers.

In addition, the UVM driver allows GPU memory oversubscription. When a GPU is under memory pressure, the UVM driver can automatically evict GPU physical pages back to the host memory, so that migration-involved GPU page faults or prefetch commands can succeed. In contrast, the pre-UVM CUDA programming model requires the programmer to find any inactive GPU buffers, transfer the data back to the CPU buffers if still useful, and free the inactive GPU buffers. Such a manual process is error-prone and requires ad-hoc efforts for each application.

2.3. Cache-coherent remote memory access

Recent IBM power CPUs have supported NVIDIA's cache-coherent CPU-GPU interconnect, NVLink and NVSwitch, that can abstract certain types of NVIDIA GPUs as NUMA nodes and provide much higher bandwidth than PCI Express [10, 33, 40]. Similarly, an Intel QPI-based hardware interconnect has been proposed to support shared virtual memory on heterogeneous CPU-FPGA platforms [32]. While cache-coherent interconnects migrate data automatically as it is accessed, remote accesses can easily become a bandwidth or latency bottleneck unless the application or system carefully places and migrates data.

Cache-coherent remote memory access among CPUs and GPUs will therefore not eliminate the need to optimize application performance through page placement and migration in heterogeneous CPU-GPU systems, just as the emergence of cache-coherent NUMA architectures, such as Stanford's DASH architecture [28], did not eliminate

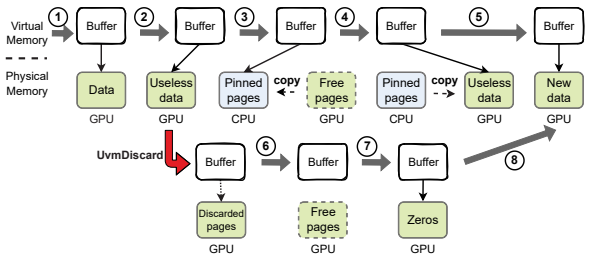


Figure 2: UvmDiscard eliminates redundant memory transfers.

that on NUMA architectures. In fact, the effects of page misplacement can be worse in a heterogeneous system. For example, on a heterogeneous system equipped with the most recent NVIDIA A100 GPUs and CPUs with common DDR4-3200 SDRAM connected by NVLink or NVSwitch, the GPU local memory bandwidth is over 2TB/s, but the GPU-to-GPU remote access bandwidth is limited to 600GB/s. Furthermore, the GPU-to-CPU remote access bandwidth is limited to 25GB/s, which is much lower than the GPU local memory bandwidth.

3. Redundant memory transfers in UVM

This paper is the first to identify the ubiquitous redundant memory transfer issue that happens frequently on intermediate buffers in heterogeneous computations. This section discusses this issue with an example of deep learning training.

3.1. Transferring useless data

Figure 2 illustrates an example of redundant memory transfers in the following steps: ①The UVM buffer is initially mapped to zero-filled GPU pages when the GPU code writes short-lived data to the buffer. ②The GPU code finishes using the short-lived data and so it has no further use for any of the data in the buffer. ③Nonetheless, the buffer is automatically migrated to the CPU due to GPU memory pressure, even though it contains no useful data because the system has no way to know that the data are useless. ④Before the GPU code can write new data to the buffer, it is automatically migrated to the GPU even though it contains no useful data again because the system cannot tell if the data are useless. ⑤New data are written to the buffer.

Redundant memory transfers can happen at both directions, and the root cause is the underlying UVM driver cannot recognize useless data and has to migrate for correctness. The user program may choose to free and reallocate the intermediate buffer. However, such a solution is not applicable when only part of the buffer becomes useless. Additionally, for temporary buffers only used by the GPU, repeatedly freeing and reallocating them imposes other overhead beyond redundant memory transfers.

The scenario in Figure 2 appears in real-world applications. For example, the forward phase for deep learning

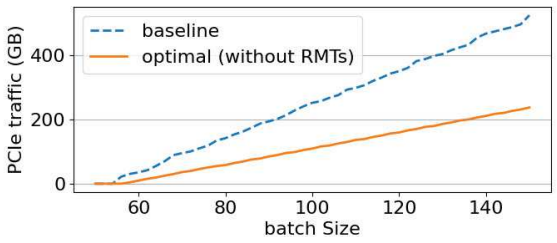


Figure 3: PCIe traffic of ResNet-53.

training stores a lot of intermediate results which becomes useless after each backward step. If the model cannot fit in the GPU's memory, a large number of redundant memory transfers could be triggered and degrade the training throughput.

Figure 3 quantifies the redundant memory transfer issue when training a convolutional neural network ResNet-53 on an ImageNet dataset with different workload sizes (denoted by training batch size). When the workload size grows beyond the GPU memory capacity, UVM starts to automatically transfer memory back and forth. However, the actual required, i.e., non-redundant, amount of memory transfer is less than half of the amount of memory transfer ordinarily performed by UVM.

Although some redundant memory transfers can be avoided with manual effort, Section 6 elaborates that it is more difficult and less efficient compared with using the discard directive proposed in this paper.

3.2. Discussion

The redundant transfers of useless data that arise automatically motivate the new kind of UVM memory directive, "discard", that is proposed by this paper. A discard directive marks the data within a specific region of virtual memory as discarded and informs the UVM driver that the next automatic memory transfer of it can be skipped. Figure 2 illustrates how this discard directive eliminates redundant memory transfers. After the data become useless, the application issues the discard operation so the eviction process (⑥) can skip migrating useless data and directly reclaim free GPU pages from discarded GPU pages. Before the buffer gets written (⑦), the driver can skip migrating old temporary data and directly zero-fill new GPU pages. Therefore, this advice can save redundant memory transfers in both directions.

As discussed in Section 2.3, most applications can benefit from improved data locality brought by automatic data transfers. Therefore, a UVM system that supports cache-coherent remote memory accesses still needs a discard directive to eliminate redundant memory transfers.

4. Semantics and application-level usage

This section defines the semantics of UvmDiscard, and illustrates how to use it in applications to eliminate RMTs. It is a CUDA API call that takes arguments defining a virtual memory region.

```
cudaMallocManaged(..); // UVM buffers A, B
Initialize(A)
cudaMemPrefetchAsync(..); // Prefetch/fault A,B (optional)
Launch GPU kernel(A, B);
UvmDiscardAsync(A, ..); // Enqueued after previous GPU kernel
...
cudaMemPrefetchAsync(..); // Prefault A on GPU (optional)
Launch GPU kernel(B, A);
cudaDeviceSynchronize();
print(A);
```

Listing 3: A CUDA example of UvmDiscard

4.1. Semantics

UvmDiscard notifies the UVM system that the data values within the specified unified virtual address range are no longer useful. After a virtual page gets discarded, a subsequent read by either a CPU or a GPU can return either zeros or old data values. The old data values are not necessarily the most recently written values, but they are values written previously by the program. On the other hand, a new value written after the discard operation by either a CPU or a GPU is guaranteed to be seen by a subsequent read, until a future discard operation is made. As shown in Listing 3, the UvmDiscard operation works like other CUDA APIs.

4.2. Application-level usage

The user must be aware of when a buffer becomes useless in order to use UvmDiscard correctly. Nonetheless, it is straightforward to use UvmDiscard in a CUDA program where all CUDA APIs and GPU kernels are submitted through the same CUDA stream. Moreover, rather than trying to infer the exact data that will be uselessly transferred, users should simply call UvmDiscard on all useless data.

In practice, it is beneficial to overlap computation with memory operations, e.g. prefetch operations, but the user must synchronize and order them correctly. Similarly, UvmDiscard should be ordered like a memory operation with other CUDA APIs and computation. CUDA has provided a rich set of synchronization APIs to help with the overlapping.

The discard operation must be called after the host or GPUs are done using the data. In Listing 3, the discard operation must be ordered after the completion of the first GPU kernel.

It is preferable to order a memory prefetch operation after the discard operation, so the buffer can be prefaulted before being repurposed again. Otherwise, the benefit of discard operation can be suppressed by extra page faults. In Listing 3, the program re-purposes buffer A to save new data. It inserts a memory prefetch operation after the previous discard operation completes and before launching the new GPU kernel. Therefore, the new GPU kernel will not trigger GPU page faults if the buffer was reclaimed.

5. Two alternative implementations

Accelerators like GPUs are still evolving to support more virtual memory functionalities. Currently, NVIDIA GPUs don't have per-PTE (page table entry) access or dirty bits [4], so the driver cannot tell whether a page has been accessed or modified since it is discarded if the driver keeps it mapped. This restricts the way to implement *UvmDiscard* in NVIDIA's UVM driver. However, the GPU hardware limitation can be overcome by maintaining a software dirty bit with the coordination of the user program. This compromises user-level programmability, but helps explore the performance implications of new GPU hardware features. This leads to an alternative implementation named *UvmDiscardLazy*.

5.1. UvmDiscard eagerly destroys mappings

UvmDiscard eagerly destroys all virtual mappings within the specified virtual region. If a discarded virtual page is later re-accessed, a page fault will be triggered. The driver then gets notified that the underlying physical page may hold new values, so it can no longer reclaim the page without transferring memory.

However, eager unmapping introduces two potential overheads. First, the unmapping request in the UVM system can be very expensive. In NVIDIA's UVM system, the page table mapping may exist in a CPU, a GPU and even be replicated by a cache-coherent peer GPU. So, UvmDiscard may need to send GPU PTE clearing and GPU TLB invalidation commands via CPU-GPU interconnects and wait for the GPU to acknowledge their completion. Second, the eager unmapping can sometimes be unnecessary and further introduce unnecessary GPU page faults when a discarded buffer is repurposed to be used by the same GPU, which may severely degrade GPU performance. However, as Section 2.1 explains, this can be alleviated with a prefetch operation, though the cost of waiting for GPUs to destroy and reestablish PTEs is unavoidable.

The eager unmapping is a design choice forced by NVIDIA GPU hardware limitations. If the GPU supports per-PTE dirty bits, then the discard operation can clear the dirty bits and rely on the hardware to automatically set them back to notify the driver of new modifications to a discarded virtual page.

5.2. UvmDiscardLazy requires programmer support

To overcome the above GPU hardware limitations, the driver can track dirty bits in software for discarded virtual regions. Therefore, *UvmDiscardLazy* works by clearing the software dirty bits of the specified virtual memory without eagerly destroying any of its virtual mappings. Because the hardware cannot automatically track the dirty bits, the user-level program is required to notify the driver if it intends to reuse the discarded memory. This notification must be done before the program is about to re-purpose a discarded virtual region. Therefore, the driver will not reclaim the GPU physical pages that hold new values as discarded ones.

Prefetches are commonly issued before using any memory on the GPU. Such prefetches either transfer data from the host to the GPU or they allocate, zero, and map new buffers. This eliminates the expensive, on-demand GPU page faults that would otherwise occur. Since it is best practice to use such prefetches in CUDA, they should also be used before re-accessing discarded memory. Therefore, the prefetch operation is modified to also set the software dirty bits of the prefetched virtual region. This prefetch operation is now mandatory in order to make use of a region discarded by *UvmDiscardLazy* and serves two purposes. If the region was not reclaimed, then the prefetch operation simply sets the software dirty bits, so the driver knows that the memory is no longer "discarded". If the region was reclaimed, it allocates, zeroes, and maps new physical memory for that virtual memory.

Note that maintaining the dirtiness of GPU pages in software is significantly cheaper than unmapping or mapping GPU PTEs, so UvmDiscardLazy can significantly outperform UvmDiscard if its eager unmapping requests are mostly unnecessary when very few discarded pages are ultimately reclaimed. Section 7 will elaborate the performance advantages of using UvmDiscardLazy.

5.3. Skipping memory transfers

After a virtual page gets discarded, the UVM system will skip the memory transfers of the underlying physical page until it gets accessed again. In UvmDiscard this means a page fault occurs or the virtual page is prefaulted by a prefetch operation. In UvmDiscardLazy this means the user program indicates the dirtiness of a virtual page with a prefetch operation. There are two major scenarios of memory transfers that can be avoided.

The first and most common case happens when the automatic eviction process selects a virtual page to reclaim after the discard operation. When a 2MB GPU allocation happens under GPU memory pressure, the UVM driver will start the automatic eviction process to reclaim GPU physical pages. The reclamation process involves swapping out a 2MB GPU physical page which is believed not recently used to the CPU DRAM. Discarded GPU physical pages are prioritized by the reclamation process, because they can be reclaimed without a memory transfer. In this case, the discard operation saves GPU-to-CPU redundant memory transfers.

The second scenario happens when accessing or prefetching a virtual page to a GPU after it gets discarded and the underlying GPU physical page gets reclaimed. Since the underlying GPU physical page has been reclaimed, a new GPU physical page will be allocated, zero-filled, and mapped. Without the discard operation, the underlying GPU physical page would have been swapped out to the CPU, and a prefetch or GPU access to the virtual page will transfer the swapped physical page back to the GPU. In this case, the discard operation saves CPU-to-GPU redundant memory transfers.

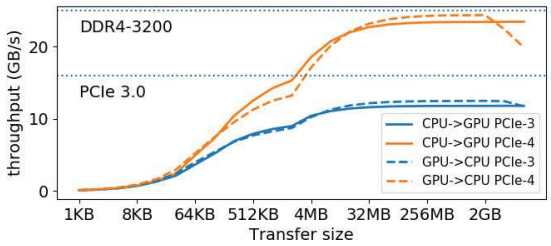


Figure 4: `cudaMemPrefetchAsync` throughput on PCIe-3/4.

5.4. 2MB granularity

NVIDIA's recent GPUs support 2MB and 4KB mappings. Using 2MB mappings in their GPUs can greatly increase the coverage of GPU TLBs and reduce GPU address translation overhead. At the same time, just like non-temporal page zeroing functions on the CPU [43], the GPU copy engine can achieve higher bandwidth when zeroing a larger contiguous GPU memory chunk. Consequently, and similar to Linux's 2MB transparent huge page (THP) support, NVIDIA's UVM driver will allocate, zero, and map a 2MB physical page upon first touch to a 2MB-aligned virtual region. In order to increase the availability of 2MB-aligned physical memory, the UVM driver uses a similar defragmentation method of evicting inactive 2MB GPU physical pages as Quicksilver [43].

Because of the above features in NVIDIA's UVM system, the discard operation prefers full 2MB-aligned virtual regions and sometimes ignores partial ones. This helps avoid splitting 2MB GPU mappings from partial unmapping requests.

Additionally, this better utilizes the CPU-GPU interconnect bandwidth. As shown in Figure 4, the bandwidth is better utilized with larger transfer sizes. If the discard operation is partially performed on a 2MB GPU physical page at a 4KB granularity, then the rest of the 2MB page may be transferred at higher cost. This can outweigh the benefit of the saved memory transfers.

5.5. Discarded GPU page queue

NVIDIA's UVM driver maintains three physical page queues for each GPU, including free, unused and used page queues. The free page queue contains free pages that are readily available to be allocated. The unused page queue is a FIFO queue containing leftover pages by the eviction process. These pages are not being used by the GPU and can be reclaimed directly. The used page queue is a pseudo-LRU queue which contains all physical pages being used. Upon a GPU page fault or a prefetch operation, the underlying GPU physical pages are moved to the most recently used side of the queue.

The eviction process starts when the GPU's free page queue is empty. It first tries to dequeue and reclaim a 2MB GPU page from the unused page queue. If it fails, it then tries to reclaim the least recently used side of the used page queue by swapping the physical data out to the CPU DRAM.

The discard operation adds another GPU page queue with FIFO order, which maximizes the time to keep each discarded GPU page in the page queue so that they have a higher chance to be recovered when they get accessed by the same GPU after a discard operation. This helps save the cost of GPU page zeroing. The eviction process is thus modified. It starts reclaiming pages from the discarded page queue after it finds the unused page queue empty, because reclaiming GPU physical pages from these page queues does not trigger memory transfers.

5.6. Delayed physical reclamation

After a discard operation, the underlying physical page of the discarded virtual page is not immediately reclaimed. If the virtual page was mapped on a CPU, then the underlying CPU page remains pinned. If the virtual page was mapped on a GPU, then it gets moved from the used GPU page queue to the discarded GPU page queue. Under GPU memory pressure, the underlying physical page may be selected by the eviction process to be reclaimed. The reclamation can involve sending unmapping requests if the page was discarded by UvmDiscardLazy instead of UvmDiscard.

5.7. Access after discard

After a UvmDiscard operation, if a virtual page gets reused by the same GPU and its underlying GPU physical page has not been reclaimed, then the GPU physical page is re-mapped and moved from the GPU's discarded page queue to the most recently used side of the GPU's used page queue. However, the cost of the same case is cheaper with UvmDiscardLazy – there is not need to reestablish any GPU mappings that were destroyed eagerly but unnecessarily.

Extra GPU page zeroing is also required if the discarded physical page to be re-purposed was not fully prepared before. For example, the UVM driver may allocate a 2MB physical page without zero-filling it if a GPU touches the first 1MB of a 2MB buffer that holds data on another GPU. Therefore, discarded pages cannot be assumed to have been prepared. To address this issue, a new data structure is added to track whether each 2MB GPU physical page has been fully prepared before, i.e., each of its 4KB pages has either been zeroed or migrated over. If necessary, the whole 2MB GPU physical page will be zeroed.

6. Programmability

This section presents pseudo code for a deep learning training program taken from Darknet which is also representative of the approaches used by TensorFlow and PyTorch [11, 12]. The program is shown with three different memory management methods in Listings 4, 5 and 6. Lines that are executed on the host or perform control operations are shown in black. Lines that correspond to CUDA API calls are shown in bold. And lines that correspond to device kernel execution are shown in italics.

Listing 4 describes the pseudo code where memory is managed by initially manually allocating host and device buffers and then orchestrating data transfers along with

```
// This will not work if device buffers exceed GPU
     capacity
For layer i from 1 to N:
   Allocate host buffers h_output_i and h_weight_i
   Allocate device buffers d_output_i and d_weight_i
   Initialize d_weight_i with random values
Allocate host buffers h_data, h_labels
Allocate device buffers d_data, d_labels, d_gradients

For each batch:
   Generate h_data, h_labels
   Transfer h_data, h_labels to d_data, d_labels

   Set d_output_0 to point to d_data
   For layer i from 1 to N:
      d_output_i = forward_i(d_output_i-1, d_weight_i)

   Set d_output_N+1 to point to d_labels
   For layer i from N to 1:
      d_gradients = backward_i(d_output_i+1, d_output_i, d_weight_i)
      // d_output_i now holds useless data
      d_weight_i = update_i(d_gradients, d_weight_i)

For layer i from 1 to N:
   Transfer d_weight_i back to h_weight_i
```

Listing 4: A DL example with manual memory management

```
// Device buffers can exceed GPU memory capacity
For layer i from 1 to N:
   Allocate host buffers h_output_i and h_weight_i
   Initialize d_weight_i with random values
Allocate host buffers h_data, h_labels

For each batch:
   Generate h_data, h_labels
   Allocate device buffers d_data, d_labels
   Transfer h_data, h_labels to d_data, d_labels

   Set d_output_0 to point to d_data
   For layer i from 1 to N:
      Allocate device buffers d_output_i, d_weight_i
      // No need to swap in d_output_i which will be
           overwritten
      Transfer h_weight_i to d_weight_i
      d_output_i = forward_i(d_output_i-1, d_weight_i)
      Transfer d_output_i to h_output_i
      Deallocate d_output_i
      // No need to swap out d_weight_i which was not changed
      Deallocate d_weight_i
   Deallocate d_output_0

   Set d_output_N+1 to point to d_labels
   For layer i from N to 1:
      Allocate device buffers d_output_i, d_weight_i, d_grad_i
      Transfer h_output_i, h_weight_i to d_output_i, d_weight_i
      // No need to swap in d_grad_i which will be overwritten
      d_grad_i = backward_i(d_output_i+1, d_output_i, d_weight_i)
      // d_output_i+1 now holds useless data
      Deallocate d_output_i+1
      d_weight_i = update_i(d_grad_i, d_weight_i)
      // d_grad_i now holds useless data
      Deallocate d_grad_i
      Transfer d_weight_i to h_weight_i
      Deallocate d_weight_i
```

Listing 5: A manual solution to oversubscribe GPU memory

computations. Such a program is easy to write, but can only work if *all* buffers fit on the GPU. This is unrealistic for most programs, as dataset sizes are growing faster than GPU capacity, and programs generally need to support various GPU models whose memory capacities can differ by an order of magnitude.

Listing 5 is a more realistic application that supports datasets that are larger than the GPU memory capacity.

```
For layer i from 1 to N:
   Allocate UVM buffers output_i, weight_i
   Initialize weight_i with random values
Allocate UVM buffers data, labels, gradients

For each batch:
   Generate data, labels
   Prefetch data, labels to device

   Set output_0 to point to data
   For layer i from 1 to N:
      Prefetch output_i to device
      output_i = forward_i(output_i-1, weight_i)

   Set output_N+1 to point to labels
   For layer i from N to 1:
      Prefetch output_i, gradients to device
      gradients = backward_i(output_i+1, output_i, weight_i)
      // output_i+1 now holds useless data
      Discard output_i+1
      weight_i = update_i(gradients, weight_i)
      // gradients now holds useless data
      Discard gradients
```

Listing 6: A DL example with UVM and UvmDiscard

	40	50	60	70	80
PyTorch-LMS	16/112	17/118	17/148	19/113	18/150
DarkNet-UVM	29/2	29/2	25/45	22/104	20/152
DarkNet-Discard	29/2	29/2	28/10	26/34	24/58

TABLE 1: Throughput(img/sec)/PCIe traffic(GB) of training VGG-16 with different batch sizes on GTX 1070.

The code is functionally equivalent to Listing 4. However, if a single neural network layer's memory footprint already exceeds the GPU memory capacity, then this will crash.

Finally, Listing 6 shows the UVM solution. The program is just as easy, if not easier, to write compared to Listing 4. Although in the deep learning case there is not need of data marshaling, it still benefits from the unified address space by eliminating dual buffer management. Further, the UVM solution allows GPU oversubscription without any burden on the programmer. Inserting discard directives to eliminate redundant transfers of useless data is as easy as, if not easier, than inserting allocation and deallocation API calls as is done in Listing 5.

Table 1 shows the training throughput of VGG-16 with PyTorch and UVM (with and without discard). For batch sizes of 60 or larger, the GPU memory is oversubscribed. PyTorch implements a manual swapping approach [11], as shown in listing 5. PyTorch augments that approach with a manual caching mechanism to avoid costly allocation and deallocation API calls (costs are shown in Table 2), exploit reuse and eliminate some redundant transfers of useful data. PyTorch's manual caching and swapping approaches cost 1,806 and 2,509 lines of code, respectively. UVM by itself outperforms PyTorch's mechanisms, both in terms of higher throughput and lower PCIe traffic.[1] Furthermore, when equipped with UvmDiscard, the PCIe traffic is reduced and the training throughput is further increased when the GPU memory is oversubscribed.

1. This explains why companies like Facebook deploy their large-scale recommendation systems with UVM [31].

Buffer Size	2MB	8MB	32MB	128MB
cudaMalloc	48	184	726	939
cudaFree	32	38	63	1184
UvmDiscard	4	7	20	70

TABLE 2: Cost of CUDA API calls in μs.

7. Evaluation

This section evaluates UvmDiscard and UvmDiscardLazy using both micro-benchmarks and real-life applications. The evaluation not only quantifies their bottom-line performance impact on benchmarks and applications, but also presents results from driver-level instrumentation to elaborate their impact. While UvmDiscard is normally expected to improve performance by eliminating RMTs, its eager unmapping implementation can cause non-negligible overheads when RMTs do not exist. Additionally, the evaluation contrasts it with how UvmDiscardLazy alleviates such overheads and encourages new GPU hardware features to be supported.

7.1. Methodology

The evaluation platform consists of a 12-core AMD Ryzen 3900X processor along with NVIDIA's 3080Ti GPU. The GPU memory capacity is 12GB, and the CPU DRAM capacity is 64GB. The CPU DRAM is DDR4 3200, so PCIe-4 throughput is bottlenecked at 25GB/s. The AMD B550-based motherboard supports switching between PCIe-3 and PCIe-4, and we collect results with both PCIe generations.

We evaluate three systems. *UVM-opt* as a baseline uses UVM programming model and adopts optimizations of memory prefetching and overlapping CUDA APIs with computation. *UvmDiscard* exercises the implementation of UvmDiscard over *UVM-opt*. *UvmDiscardLazy* further replaces the UvmDiscard operations that are paired with prefetch operations in *UvmDiscard*, but not all of them because the prefetch operation sometimes introduces GPU memory thrashing.

We use two different methodologies for evaluating the effects of GPU memory oversubscription. For the micro-benchmarks and GPU database application, we fix the input sizes of the applications and run an idle GPU program that occupies specific amounts of GPU memory to create oversubscription ratios of <100%, 200%, 300% and 400%. The oversubscription ratio is the ratio of the GPU memory consumption of the application to the available GPU memory. For deep learning, we fix the training data set and gradually increase the training batch size to oversubscribe memory.

The microbenchmarks include FIR and Radix-sort from existing benchmark suites [25, 38]. They exhibit relatively simple memory transfer patterns, which makes it easier to analyze and explain how UvmDiscard and UvmDiscardLazy are affecting their performance.

The real-world applications include a common GPU database operation and deep learning training. They exhibit much more complicated memory transfer patterns and provide more realistic use cases for understanding both the applicability and effects of UvmDiscard and UvmDiscardLazy.

Ovsp. rate	<100%	200%	300%	400%
UVM-opt	1/1	1/1	1/1	1/1
UvmDiscard	1/1.01	0.51/0.52	0.62/0.65	0.71/0.71
UvmDiscardLazy	1/1.00	0.52/0.52	0.62/0.66	0.72/0.71

TABLE 3: Normalized runtime of FIR (PCIe 3/4).

Ovsp. rate	<100%	200%	300%	400%
UVM-opt	5.66	11.44	13.38	14.34
UvmDiscard	5.66	5.88	7.81	8.78
UvmDiscardLazy	5.66	5.88	7.81	8.78

TABLE 4: PCIe traffic (GB) of FIR.

7.2. FIR

FIR runs a finite impulse response (FIR) filter to produce an impulse response to any length of a finite input. The program iterates through a large input buffer, prefetches a window of the host data to the FIR GPU kernel and calculates the FIR filter. The target buffer to discard is the sliding window of the input buffer at the end of each iteration, because the sliding window becomes useless.

Since the problem size is fixed, 5.66 GB of input data is prefetched to the GPU for all configurations. However, both UvmDiscard and UvmDiscardLazy consistently eliminate 5.56GB of redundant memory transfers (eviction from CPU to GPU), as shown in Table 4. Correspondingly, Table 3 shows their reduced GPU runtime. The reduction of overall GPU runtime is slightly higher with PCIe 4, because the application spent less time in memory transfers compared with PCIe 3. When memory is not oversubscribed, the API cost of UvmDiscard can be overlapped with computation.

7.3. Radix-sort

Radix-sort sorts a large input array of keys and values. In each iteration, it launches a GPU kernel to perform local radix sorts with results saved in a temporary buffer. At this time, the input buffer can be discarded. It then launches another GPU kernel, reorders the local partitions from the temporary buffer and overwrites the results back to the input buffer. At this time, the temporary buffer can be discarded. In this application, GPU thrashing happens when the application oversubscribes memory, where each GPU kernel oversubscribes memory.

Table 5 demonstrates the normalized GPU runtime when all systems use proper prefetching operations, which are performed only when memory is not oversubscribed. Without prefetch operations when memory is not oversubscribed, UvmDiscard may introduce as high as a 3.9x slow-down (not shown) merely from extra GPU page faults which only reestablishes GPU mappings unnecessarily destroyed by UvmDiscard. Although alleviated with prefetch operations, the extra unmapping and mapping operations from discard and prefetch operations still introduced a slow-down over 1.2x. In contrast, UvmDiscardLazy makes such overheads negligible, encouraging new GPU hardware features. The benefit of eliminating RMTs from both discard implementations diminishes when GPU kernel starts to thrash memory, which becomes a dominant bottleneck as shown in Table 6.

Ovsp. rate	<100%	200%	300%	400%
UVM-opt	1/1	1/1	1/1	1/1
UvmDiscard	1.21/1.28	0.87/0.83	0.95/0.93	0.97/0.97
UvmDiscardLazy	1.00/1.02	0.87/0.83	0.95/0.92	0.97/0.99

TABLE 5: Normalized runtime of Radix-sort (PCIe-3/4).

Ovsp. rate	<100%	200%	300%	400%
UVM-opt	5.00	300.80	345.40	356.85
UvmDiscard	5.00	244.93	315.50	339.76
UvmDiscardLazy	5.00	244.92	315.52	339.76

TABLE 6: PCIe traffic (GB) of Radix-sort.

The thrashing issue is why UvmDiscard or UvmDiscardLazy have less impact on Radix-sort compared with FIR. However, it remains difficult to solve GPU thrashing. The GPU does not follow a deterministic pattern to access parallel columns of data. If the GPU cannot fit all data consumed by a single GPU kernel, then manually prefetching its data usually does more harm. Therefore, the best practice to alleviate GPU thrashing is still rewriting the program with better data locality so that the automatic memory migration driven by GPU page faults can work better.

7.4. Hash-join

Hash-join is a common operation for databases that can be accelerated by GPUs. Because modern databases are large, the memory footprint can easily exceed the memory capacity of a GPU. We exercise a GPU hash-join application which cannot fit in the GPU memory [37].

The application first launches two GPU kernels that preprocess two database tables. Both kernels use many intermediate buffers that can be discarded and their outputs become the input of the third GPU kernel that computes the joined database table of the final results. The results then get discarded and such a process is repeated by reusing the existing buffers, which simulates what happens in a GPU database.

Table 8 shows the amount of memory transfers and Table 7 shows the total GPU runtime of the two hash-join operations. By discarding the intermediate buffers, the redundant memory transfers are eliminated and the total GPU runtime is reduced. Unlike the previous two applications, *UvmDiscardLazy* introduces no more than 4% overhead when memory is not oversubscribed, because in this case not all UvmDiscard calls can be replaced with UvmDiscardLazy. However, it still alleviates the overhead compared with *UvmDiscard*.

7.5. Deep learning

Training a deep learning neural network is a common task accelerated by GPUs. Each training step contains two phases: the forward process and the backward process. The forward process generates outputs of each layer which overwrite their output buffers, and the backward process uses the previously computed outputs to generate gradients and update the weights of the neural network model.

Plenty of buffers can be discarded in the training process. For example, after forwarding or backwarding each layer, intermediate buffers used by the CUDNN library can be discarded. Also, after backwarding each layer, its intermediate buffers that save the outputs and gradients can be discarded, as they will be overwritten in the next training loop.

Ovsp. rate	<100%	200%	300%	400%
UVM-opt	1/1	1/1	1/1	1/1
UvmDiscard	1.05/1.09	0.24/0.31	0.51/0.54	0.86/0.89
UvmDiscardLazy	1.02/1.04	0.24/0.31	0.51/0.54	0.86/0.88

TABLE 7: Normalized runtime of Hash-join (PCIe-3/4).

Ovsp. rate	<100%	200%	300%	400%
UVM-opt	2.98	34.62	36.42	58.23
UvmDiscard	2.98	4.89	16.19	46.61
UvmDiscardLazy	2.98	4.89	16.19	46.44

TABLE 8: PCIe traffic (GB) of Hash-join.

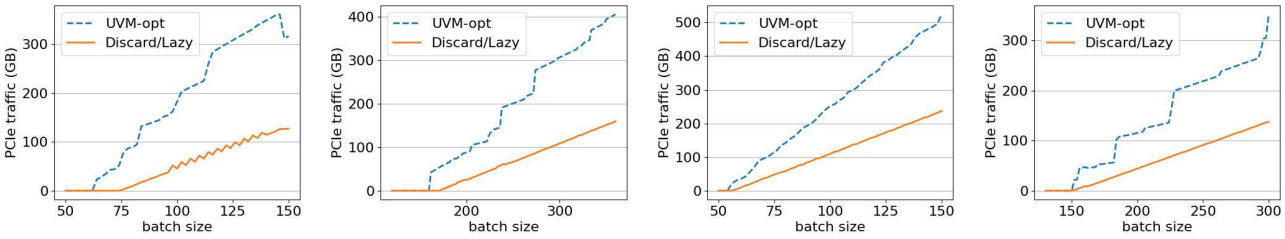
We exercise the discard directive on a deep learning platform named Darknet [3] which is written in pure C and CUDA after converting it to use the UVM programming model.

A wide variety of popular neural networks are selected, named VGG-16, Darknet-19, ResNet-53 and RNN [15, 24, 30, 36]. The first three CNN-based neural networks are trained on the ImageNet dataset [20], while the RNN neural network is trained on the Shakespeare novel corpus. Their training workload increases linearly as the batch sizes increases. For CUDA buffers, VGG-16 allocated 12.0 GB and 21.1 GB at batch sizes of 75 and 150; Darknet-19 allocated 11.2 GB and 23.4 GB at batch sizes of 171 and 360; ResNet-53 allocated 10.8 GB and 28.5 GB at batch sizes of 56 and 150; RNN allocated 10.2 GB and 20.0 GB at batch sizes of 150 and 300. The 3080 Ti GPU reports a total of 11.77GB physical memory. We trained each neural network three mini-batches and measure the average throughput of the next seven mini-batches. These measurements exclude the pre-processing of input data.

7.5.1. When DL fits the GPU. Figure 6 and Figure 7 demonstrate the training throughput. When everything fits in the GPU, *UVM-opt* demonstrates slightly lower throughput compared with *No-UVM* for the three CNN models because the highly-optimized CNN-related GPU kernels from CUDNN library cannot fully overlap the cost of prefetch operations, which neither transfer or prefault memory but only update the recency of page accesses. In this case, *UvmDiscard* can degrade the throughput by 16% because of its unnecessary eager unmapping, which also introduces unnecessary prefaulting in the prefetch operations. *UvmDiscardLazy* on the other hand significantly reduces such overheads and demonstrates negligible overhead compared with *UVM-opt*.

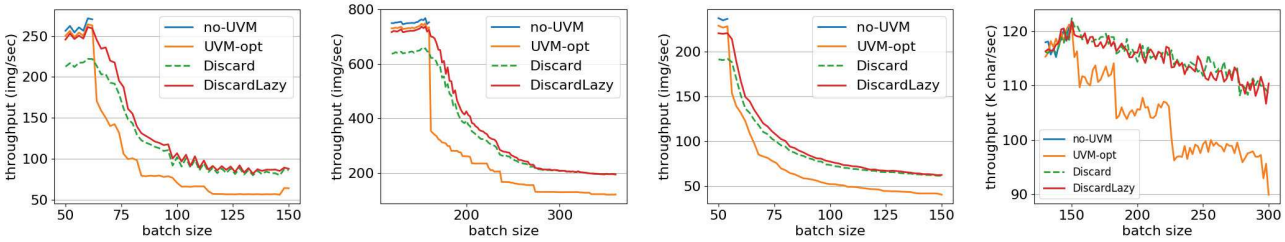
When computation is more intensive, the overlapping works better and the degradation from CUDA APIs becomes negligible, as shown in Figure 6d and Figure 7d.

7.5.2. DL with large training size. Sometimes a deep neural network may oversubscribe GPU memory when it uses a large training size, e.g. the model is very large or it uses a large batch size for fine-tuning. As shown in



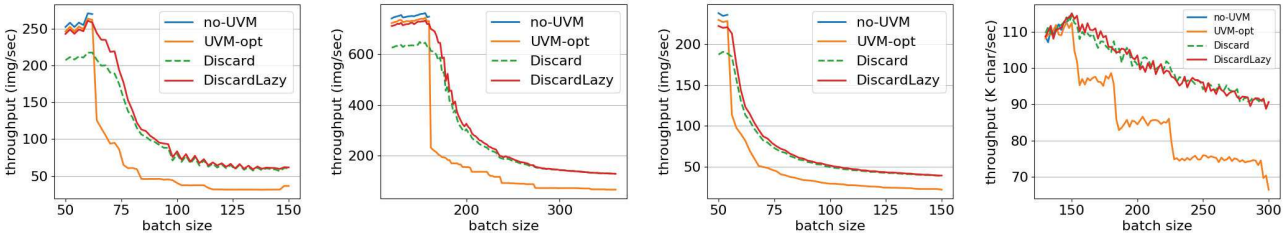
(a) VGG-16 (b) DarkNet-19 (c) ResNet-53 (d) RNN

Figure 5: PCIe traffic in deep learning. UvmDiscard and UvmDiscardLazy fully eliminate RMTs.



(a) VGG-16 (b) DarkNet-19 (c) ResNet-53 (d) RNN

Figure 6: Training throughput of deep learning with PCIe-4.



(a) VGG-16 (b) DarkNet-19 (c) ResNet-53 (d) RNN

Figure 7: Training throughput of deep learning with PCIe-3.

Figures 5, the PCIe traffic increases along with the batch size. In some circumstances, the amount of data transfers may drastically increase because the CUDNN library switches to a different algorithm that uses a different size of workspace buffer. However, in all four neural networks, the PCIe traffic can all be dramatically decreased with discard operations. A huge amount of redundant memory transfers results from swapping in and out intermediate buffers or useless data.

The training throughput is consistently improved with either PCIe-3 or PCIe-4 as in Figure 6 and Figure 7 whether using UvmDiscard or UvmDiscardLazy. When the batch size increases and the four neural networks start to oversubscribe GPU memory, the training throughput decreases and `UvmDiscardLazy` is consistently the best performer. Furthermore, both UvmDiscard and UvmDiscardLazy consistently improve the training throughput regardless of whether the neural network is compute-intensive (RNN) or memory-intensive (Darknet-19, VGG-16 and ResNet-53).

8. Related work

In addition to cache-coherent interconnects, recent architectures have adopted a heterogeneous design that lets the CPU and GPU share the physical memory or use the same cache, e.g. AMD's APU and Apple's M1 chips [16, 18, 19]. Such a heterogeneous architecture can bring advantages like lower power consumption due to the smaller cost of data transfers. However, their GPU DRAM bandwidth is usually much lower than what a discrete GPU card can achieve.

Unix-like systems have the `madvise` system call to enable applications to provide the system with user-level knowledge of memory usage patterns [1, 2]. For example, MADV_SEQUENTIAL and MADV_WILLNEED give the operating system hints about future memory access patterns, so that instead of general memory management policies, the operating system can prefetch memory according to these hints from the user program. The prefetching effect is similar as `cudaMemPrefetchAsync` in NVIDIA's UVM driver, except that the latter operation is guaranteed to prefetch memory.

Another example is MADV_FREE/MADV_DONTNEED. These two directives can help reduce the cost of reclaiming unneeded physical memory and increase throughput for applications like Redis [17]. Specifically, they can reduce

I/O costs since their semantics discard the data content of the specified virtual memory region, which eliminates the need to save that data back to the disk under memory pressure. The effects of these directives, discarding data content and freeing physical memory, is similar to those of UvmDiscard and UvmDiscardLazy.

Several techniques have been developed to deal with applications that consume more memory than the GPU's physical memory. These techniques include rewriting the algorithms to reduce their workload sizes to fit in the GPU memory [23, 35], compressing sparse data to minimize data migration [34, 39, 42] and optimizing the runtime library to avoid RMTs [13]. In addition, optimizing data replacement policies with predictions of memory accesses can also reduce memory transfers [21, 22]. These techniques can be used in conjunction with UvmDiscard and UvmDiscardLazy. Furthermore, a compiler-assisted approach that detects the buffer reuse distance can be extended to diagnose the insertion of UvmDiscard API calls [29].

Some applications like databases can benefit from caching the intermediate results in the GPU buffer for higher throughput [6], while other applications like deep learning use many intermediate results during the back-propagation process [14, 26]. These applications may implement ad-hoc manual memory management policies to migrate these intermediate buffers in and out [27]. However, such approaches can require thousands of lines of additional code to process data larger than GPU memory [7]. Other approach chooses to recompute intermediate results to save memory consumption, but it does not ultimately avoid RMTs [41].

9. Conclusion

Unified virtual memory has the potential to greatly simplify heterogeneous programming across host CPUs and accelerator devices, such as GPUs. However, performance hurdles remain. This paper has shown that the addition of a discard directive enables the elimination of some RMTs by enabling the application to notify the system that the contents of a memory buffer are no longer needed and can be discarded safely. Two implementations have been evaluated within NVIDIA's open-source UVM driver in order to demonstrate their usefulness on real-world applications characterized with RMTs. Both *UvmDiscard* and *UvmDiscardLazy* demonstrate up to four times speedup on a GPU database application. Furthermore, they can eliminate over 60% memory transfers across different types of deep neural networks and improve their training throughput from 23% to 61%. *UvmDiscardLazy* also consistently alleviates the API overhead of *UvmDiscard*, encouraging new GPU hardware features to combine the ease of use of *UvmDiscard* with the performance of *UvmDiscardLazy*.

References

[1] "madvise (2) linux manual page," https://man7.org/linux/man-pages/man2/madvise.2.html.

[2] "madvise freebsd manual pages," https://www.freebsd.org/cgi/man.cgi?query=madvise&sektion=2.

[3] "Darknet: Deep learning platform," 2021. [Online]. Available: https://github.com/AlexeyAB/darknet

[4] "Nvidia pascal mmu format," 2021. [Online]. Available: https://github.com/NVIDIA/open-gpu-doc/blob/master/pascal/gp100-mmu-format.pdf

[5] "Nvidia unified virtual memory," 2021. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/

[6] "Omniscidb: Advanced memory management," 2021. [Online]. Available: https://www.omnisci.com/platform/omniscidb

[7] "Tflms patch, github," 2021. [Online]. Available: https://github.com/IBM/tensorflow-large-model-support/blob/master/patches/tensorflow_v2.2.0_large_model_support.patch

[8] "Top 10 supercomputers," 2021. [Online]. Available: https://www.top500.org/lists/top500/2021/11/

[9] "Nvidia tensor cores," 2022. [Online]. Available: https://developer.nvidia.com/tensor-cores

[10] "Nvlink - nvidia," 2022. [Online]. Available: https://en.wikichip.org/wiki/nvidia/nvlink

[11] "Pytorch large model support from ibm," 2022. [Online]. Available: https://github.com/IBM/pytorch-large-model-support

[12] "Tensorflow large model support from ibm," 2022. [Online]. Available: https://github.com/IBM/tensorflow-large-model-support

[13] R. Asai, M. Okita, F. Ino, and K. Hagihara, "Transparent avoidance of redundant data transfer on gpu-enabled apache spark," in *Proceedings of the 11th Workshop on General Purpose GPUs*, 2018, pp. 22–30.

[14] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, and D. K. Panda, "Oc-dnn: Exploiting advanced unified memory capabilities in cuda 9 and volta gpus for out-of-core dnn training," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 143–152.

[15] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[16] A. Branover, D. Foley, and M. Steinman, "Amd fusion apu: Llano," *Ieee Micro*, vol. 32, no. 2, pp. 28–37, 2012.

[17] J. L. Carlson, *Redis in action*. Manning Publications Co., 2013.

[18] A. Corporation, "Introducing m1 pro and m1 max: the most powerful chips apple has ever built," 2021.

[19] M. Daga, A. M. Aji, and W.-c. Feng, "On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing," in *2011 Symposium on Application Accelerators in High-Performance Computing*. IEEE, 2011, pp. 141–149.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[21] D. Ganguly, R. Melhem, and J. Yang, "An adaptive framework for oversubscription management in cpu-gpu unified memory," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1212–1217.

[22] D. Ganguly, Z. Zhang, J. Yang, and R. Melhem, "Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 451–461.

[23] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[25] B. Hu and C. J. Rossbach, "Altis: Modernizing gpgpu benchmarks," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE Computer Society, 2020, pp. 1–11.

[26] Y. Ito, H. Imai, T. L. Duc, Y. Negishi, K. Kawachiya, R. Matsumiya, and T. Endo, "Profiling based out-of-core hybrid method for large neural networks," *arXiv preprint arXiv:1907.05013*, 2019.

[27] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "Tflms: Large model support in tensorflow by graph rewriting," *arXiv preprint arXiv:1807.02037*, 2018.

[28] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam, "The stanford dash multiprocessor," *Computer*, vol. 25, no. 3, pp. 63–79, 1992.

[29] L. Li and B. Chapman, "Compiler assisted hybrid implicit and explicit gpu memory management under unified address space," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–16.

[30] L. R. Medsker and L. Jain, "Recurrent neural networks," *Design and Applications*, vol. 5, pp. 64–67, 2001.

[31] D. Mudigere, Y. Hao, J. Huang, A. Tulloch, S. Sridharan, X. Liu, M. Ozdal, J. Nie, J. Park, L. Luo *et al.*, "High-performance, distributed training of large-scale deep learning recommendation models," *arXiv e-prints*, pp. arXiv–2104, 2021.

[32] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia *et al.*, "A reconfigurable computing system based on a cache-coherent fabric," in *2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2011, pp. 80–85.

[33] C. Pearson, I.-H. Chung, Z. Sura, W.-M. Hwu, and J. Xiong, "Numa-aware data-transfer measurements for power/nvlink multi-gpu systems," in *International Conference on High Performance Computing*. Springer, 2018, pp. 448–454.

[34] A. H. N. Sabet, Z. Zhao, and R. Gupta, "Subway: Minimizing data transfer during out-of-gpu-memory graph processing," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[35] K. Shirahata, H. Sato, and S. Matsuoka, "Out-of-core gpu memory management for mapreduce-based large-scale graph processing," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2014, pp. 221–229.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[37] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki, "Hardware-conscious hash-joins on gpus," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 698–709.

[38] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–10.

[39] R. Tang, Z. Zhao, K. Wang, X. Gong, J. Zhang, W. Wang, and P.-C. Yew, "Ascetic: Enhancing cross-iterations data efficiency in out-of-memory graph processing on gpus," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[40] I. P. N. team, "Functionality and performance of nvlink with ibm power9 processors," *IBM J. Res. Dev.*, vol. 62, no. 4–5, p. 9:1–9:10, jul 2018. [Online]. Available: https://doi.org/10.1147/JRD.2018.2846978

[41] M. Wahib, H. Zhang, T. T. Nguyen, A. Drozd, J. Domke, L. Zhang, R. Takano, and S. Matsuoka, "Scaling distributed deep learning workloads beyond the memory capacity with karma," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[42] X. Yang, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus: Implications for graph mining," *arXiv preprint arXiv:1103.2405*, 2011.

[43] W. Zhu, A. L. Cox, and S. Rixner, "A comprehensive analysis of superpage management mechanisms and policies," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 829–842.