공학박사학위논문

Data Management and Prefetching Techniques for CUDA Unified Memory

CUDA Unified Memory를 위한 데이터 관리 및 프리페칭 기법

2022 년 8 월

서울대학교 대학원

컴퓨터 공학부

정 재 훈

공학박사학위논문

Data Management and Prefetching Techniques for CUDA Unified Memory

CUDA Unified Memory를 위한 데이터 관리 및 프리페칭 기법

2022 년 8 월

서울대학교 대학원

컴퓨터 공학부

정 재 훈

Data Management and Prefetching Techniques for CUDA Unified Memory

CUDA Unified Memory를 위한 데이터 관리 및 프리페칭 기법

지도교수 이 재 진

이 논문을 공학박사학위논문으로 제출함

2022 년 3 월

서울대학교 대학원

컴퓨터 공학부

정 재 훈

정재훈의 공학박사 학위논문을 인준함

2022 년 6 월

위 원 장	김 진 수	(인)
부위원장	이 재 진	(인)
위 원	문 수 묵	(인)
위 원	정 창 희	(인)
위 원	조 형 민	(인)

Abstract

Unified Memory (UM) is a component of CUDA programming model which provides a memory pool that has a single address space and can be accessed by both the host and the GPU. When UM is used, a CUDA program does not need to explicitly move data between the host and the device. It also allows GPU memory oversubscription by using CPU memory as a backing store. UM significantly lessens the burden of a programmer and provides great programmability. However, using UM solely does not guarantee good performance. To fully exploit UM and improve performance, the programmer needs to add user hints to the source code to prefetch pages that are going to be accessed during the CUDA kernel execution.

In this thesis, we propose three frameworks that exploits UM to improve the ease-of-programming while maximizing the application performance. The first framework is HUM, which hides host-to-device memory copy time of traditional CUDA program without any code modification. It overlaps the host-to-device memory copy with host computation or CUDA kernel computation by exploiting Unified Memory and fault mechanisms. The evaluation result shows that executing the applications under HUM is, on average, 1.21 times faster than executing them under original CUDA. The speedup is comparable to the average speedup 1.22 of the hand-optimized implementations for Unified Memory.

The second framework is DeepUM which exploits UM to allow GPU memory oversubscription for deep neural networks. While UM allows memory oversubscription using a page fault mechanism, page fault handling introduces enormous overhead. We use a correlation prefetching technique to solve the problem

and hide the overhead. The evaluation result shows that DeepUM achieves comparable performance to the other state-of-the-art approaches. At the same time, our framework can run larger batch size that other methods fail to run.

The last framework is SnuRHAC that provides an illusion of a single GPU for the multiple GPUs in a cluster. Under SnuRHAC, a CUDA program designed to use a single GPU can utilize multiple GPUs in a cluster without any source code modification. SnuRHAC automatically distributes workload to multiple GPUs in a cluster and manages data across the nodes. To manage data efficiently, SnuRHAC extends Unified Memory and exploits its page fault mechanism. We also propose two prefetching techniques to fully exploit UM and to maximize performance. The evaluation result shows that while SnuRHAC significantly improves ease-of-programming, it shows scalable performance for the cluster environment depending on the application characteristics.

Keywords: GPU, CUDA, Unified Memory, Prefetching, Memory Latency Hiding, Large-scale DNN, Cluster

Student Number: 2014-21775

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past decade, heterogeneous computing has become a de-facto standard for high-performance computing. It incorporates different types of processors, such as CPUs, GPUs, FPGAs, and DSPs. Among many different types of processors, GPU is one of the most popular processors to accelerate system performance. According to the latest Top500[1] list, five out of the top ten systems use GPUs. Many programming models have been proposed for general-purpose computing on GPUs (GPGPU): CUDA[2], oneAPI[3], OpenACC[4], OpenCL[5], SYCL[6], etc. Among others, CUDA is one of the popular programming models for GPUs.

CUDA provides Unified Memory (UM) which is a memory pool that has a single address space and can be accessed by both the host and the GPU[7]. When UM is used, a CUDA program does not need to explicitly move data between the host and the device. The UM system exploits the page fault engine in the GPU[8], and it automatically migrates accessed pages between the host and the GPU. UM significantly lessens the burden of a programmer to manage

data distribution across the host and the GPU. However, using UM solely does not guarantee good performance. To fully exploit UM and improve performance, the programmer needs to add user hints to the source code to prefetch pages that are going to be accessed during the kernel execution.

By exploiting CUDA UM and fault mechanisms we try to solve several problems in high-performance computing area.

First, one of the major challenges in high-performance computing is hiding the memory transfer time between the host and the device as much as possible. By exploiting CUDA UM and fault mechanisms in both the CPU and the GPU, overlapping data transfers and computation can be well controlled. We propose a framework, called *HUM* (Hidden Unified Memory), as a solution of this problem. It automatically hides the *host-to-device memory copy* (in short, *H2Dmemcpy* hereafter) time by overlapping it with *host computation* or *kernel computation*. Here, the host computation is the execution of the host code that does not depend on H2Dmemcpy commands. It includes CPU computation, host memory allocation/deallocation, file I/O, etc. Moreover, we propose runtime techniques to maximize the overlapping of the H2Dmemcpy command with kernel execution. HUM is the first work that automatically hides the H2Dmemcpy time by overlapping it with host computation or kernel computation without any explicit UM command and any modification of the source code.

Second, the current trend in deep learning requires a tremendous amount of GPU memory and computation power because deeper and wider layers generally provide better accuracy[9, 10, 11]. To meet these conditions, hundreds to thousands of GPUs that have a large amount of GPU memory are used for training. Since public users are hard to utilize these amount of expensive high-end GPUs, most of the research and model training is led by big companies.

Fortunately, training a model from scratch in a large system (pre-train) and then fine-tuning the model in a relatively small system based on the individual purpose provides good accuracy. Therefore, it is a common approach to get a pre-trained model and fine-tune it individually in a small system. However, the problem is that the current state-of-the-art models are so big that even fine-tuning the models is hard to be performed in a small system, especially in a single GPU system. Many studies have been performed to solve memory capacity problem such as data compression[12, 13, 14, 15, 16], mixed-precision[17, 18, 19], data recomputation[20, 21, 22], and memory swapping[15, 22, 23, 24, 25, 26, 27, 28, 29]. Among others, we focus on memory swapping to overcome the memory capacity problem of deep neural networks (DNNs).

We propose a framework called DeepUM that exploits UM to allow oversubscribing GPU memory and implements several techniques to minimize the overhead caused by UM. While previous approaches tend to prefetch data in tensor level or layer level, DeepUM prefetches data in UM block (2MB) level with correlation prefetching. Since page fault addresses can be easily tracked in the Linux kernel module, we use these addresses for correlation prefetching. Correlation prefetching is synergetic with Unified Memory because the page fault mechanism requests data in a page level. While the previous approaches profile memory accesses in tensor level or layer level we can profile memory accesses in a more fine-grained manner. Moreover, the kernel execution patterns and the memory access patterns within the kernels are fixed and repeated in the training phase of a DNN. Thus, it is desirable to memorize the repeated patterns and exploit the information through correlation prefetching. We introduce the correlation prefetching specialized to DNN workload and a new page-eviction policy coupled with correlation prefetching. DeepUM is the first work that uses UM to target large-scale DNN workload.

Third, early GPU applications focused on fully exploiting the computing power of a single GPU. As data size gets bigger and the complexity of applications increases, programmers start to use multiple GPUs. To use multiple GPUs, programmers need to rewrite their GPU code for multiple GPUs. Sometimes, this requires much more effort than just rewriting. They need to fully understand the characteristics of an application and how to use a programming model to orchestrate multiple GPUs. The problem gets worse if they use multiple GPUs spread out over multiple nodes of a heterogeneous cluster. Communication libraries, such as MPI, are required to distribute workloads and manage data across the nodes.

To overcome this problem, we propose a framework called *SnuRHAC* (Runtime for Heterogeneous Accelerator Cluster) that provides an illusion of a single GPU for the multiple GPUs spread out over multiple nodes of a cluster. SnuRHAC provides wrapper functions of CUDA commands so that all CUDA commands are handled by the SnuRHAC runtime system. When a programmer writes a CUDA program for a single GPU and runs it under SnuRHAC, SnuRHAC can automatically distribute workloads to multiple GPUs in the cluster and manages data across the nodes. No code modification is required. This significantly lessens the burden of a programmer to rewrite their code for multiple GPUs. One key componet of the SnuRHAC is Cluster Unified Memory which is an extended version of a CUDA Unified Memory to work across multiple nodes. In addition, SnuRHAC uses two page prefetching techniques: static prefetching and dynamic prefetching. Static prefetching uses a static memory access pattern analyzer to analyze the access range of memory operations in CUDA kernels. To handle memory access patterns that cannot be analyzed statically, SnuRHAC uses the dynamic prefetching technique. For dynamic prefetching, SnuRHAC uses the adjacent page prefetching technique

and correlation prefetching technique. SnuRHAC is the first work that automatically distributes and manages workloads across the multiple GPUs in a cluster.

Major contributions of this thesis are as follows:

- We propose a framework called HUM, which exploits CUDA UM and fault mechanisms of both the host and the GPU. It automatically hides the H2Dmemcpy time by overlapping it with the host or kernel computation.
- We propose a framework called DeepUM, which exploits CUDA UM to allow GPU memory oversubscription for DNNs. It automatically prefetches data using correlation prefetching specialized for DNNs. Correlation tables record the history of the kernel executions and the page access patterns during the training phase of DNNs.
- In addition, DeepUM uses several optimization techniques to hide or eliminate the time consumed for handling GPU faults when memory is oversubscribed.
- We propose a framework called SnuRHAC, which provides an illusion of a single GPU for the multiple GPUs in a cluster. It distributes workloads to multiple GPUs and manages data across the nodes. It also is fully automatic and transparent to users.
- SnuRHAC uses static and dynamic prefetching techniques to exploit UM fully and improve its performance. The static prefetching allows SnuRHAC to prefetch data using information from a static memory access pattern analyzer MAPA[30]. The dynamic prefetching complements the static prefetching technique when MAPA cannot statically analyze memory access patterns in CUDA kernels.
- We evaluate HUM using 51 CUDA benchmark applications from Parboil[31],

Rodinia[32], and CUDA Code Samples[33]. The evaluation result shows that executing the applications under HUM is, on average, 1.21x faster than executing them under original CUDA. The speedup is comparable to the average speedup of 1.22 that is obtained by manually porting and optimizing the applications with Unified Memory.

- We evaluate DeepUM using various large-scale DNNs in MLPerf[34], PyTorch examples[35], and Hugging Face[36]. The evaluation result shows that DeepUM achieves comparable performance to the other memory swapping approaches. Moreover, DeepUM can run larger batch size configurations that other methods fail to run.
- We evaluate SnuRHAC using 18 CUDA benchmark applications from CUDA Code Samples[33], Parboil[31], PolyBench[37], and Rodinia[32]. The evaluation result shows that SnuRHAC achieves scalable performance for cluster environments while significantly reducing a programmer's burden.

Chapter 2

Related Work

Many techniques for overlapping host-to-GPU data transfers and GPU kernel computation have been proposed[38, 39, 40, 41, 42, 43]. While they require a user to manually overlap the data transfers and the kernel computations, HUM automatically does it without any code modification. Overlapping communication and CPU/GPU computation in a cluster has also been widely studied[44, 45, 46, 47, 48]. White III and Dongarra[44] show the effect of overlapping CPU/GPU computation, inter-node communication, and CPU-GPU communication. Danalis et al.[45], Fishgold et al.[46], and Danalis et al.[47] introduce compiler techniques that transform MPI code to overlap inter-node communication and CPU computation. Gysi et al.[48] propose a framework that automatically overlaps inter-node communication and GPU computation. Compared to previous approaches, HUM focuses on automatic overlapping of data transfers and GPU computation in a node by exploiting Unified Memory.

There are two categories of studies that have been performed to overcome the memory capacity problem through the memory swapping technique. The

first category is using pure GPU memory with swapping-in/swapping-out memory objects to CPU memory or NVMe device[15, 22, 23, 25, 26, 27, 28, 49].

vDNN[23] is the first paper that introduces the GPU memory swapping for deep learning workloads. However, the DNN models should be designed using vDNN API and it only supports convolutional neural network (CNN) models.

TFLMS[25] is an open-source project developed by IBM. It schedules swap-in/swap-out commands by modifying computational graphs of TensorFlow. It requires modifying the TensorFlow framework code and TensorFlow user script.

Both Superneurons[22] and FlashNeuron[15] take a DNN model as input and derive an optimal tensor offloading schedule. While Superneurons offloads memory to the CPU memory, FlashNeuron[15] utilizes NVMe SSD to offload the memory.

AutoTM[27] and SwapAdvisor[28] also take a DNN model as input to schedule the memory operations. However, AutoTM uses integer linear programming not only to schedule data movement but also to reduce device memory fragmentation. On the other hand, SwapAdvisor[28] uses a genetic algorithm to schedule operators, memory allocations, and swap decisions. Capuchin[26] identifies the tensor access patterns in runtime and schedules tensor eviction/prefetching and recomputation.

Sentinel[29] is based on TensorFlow and dynamically gathers tensor access information from both the TensorFlow runtime and the operating system. It is similar to DeepUM because it exploits the page fault mechanism to profile and obtain the memory access patterns. However, the Sentinel uses the CPU page fault mechanism while DeepUM uses the GPU page fault mechanism of the GPU. In the profiling phase of the Sentinel, tensors are allocated in pinned memory of the CPU, and the GPU accesses it. Another difference between the Sentinel and DeepUM is that the Sentinel requires modifying TensorFlow and

TensorFlow user scripts to insert Sentinel profiling API functions calls. The evaluation result of Sentinel shows that it outperforms vDNN, SwapAdvisor, AutoTM, and Capuchin in training throughput.

DeepSpeed[49] is a highly optimized deep learning framework that is widely used for multiple GPU environments. It keeps track of the sequence of DNN operations by hooking PyTorch APIs. Then, it provides memory offloading to the main memory or NVMe SSD. However, DeepSpeed supports offloading model parameters, gradients, and optimizer states only. Activation memory and temporary buffers should be managed manually by the programmer.

The other category of the swapping approach exploits CUDA UM with prefetching[24, 50]. OC-DNN[24] manually inserts prefetch commands in front of each DNN operation. DRAGON[50] uses an NVMe SSD as a backing store for UM. It targets general applications and uses a DNN workload as a showcase. It also requires user code modification and device driver modification.

DeepUM differs from the previous approaches in that we exploit UM to allow oversubscribing GPU memory and use the correlation prefetching technique to predict future memory accesses. We also propose various optimization techniques to reduce the GPU page fault handling time. While DeepUM needs very few code modifications in the PyTorch framework, it neither requires any user python code modification nor modifications to OS kernel or NVIDIA device driver.

Many studies have been performed to provide a framework that automatically distributes workloads to multiple devices in a single node[51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61]. Most of these previous approaches perform static analysis on kernels to determine memory access patterns. If memory access patterns can be analyzed statically, they transfer only necessary data to each device. Otherwise, they transfer entire data to devices. SnuRHAC is similar

to the previous approaches in that it prefetches data based on static information. However, SnuRHAC does not prefetch the entire input data when memory access patterns cannot be analyzed statically. Instead, it uses dynamic prefetching techniques so that useless memory traffic can be minimized. Also, previous approaches require merging output data at the end of the kernel. However, SnuRHAC does not need this stage because pages are automatically migrated between devices when there are page sharing between devices.

Some studies focus on exploiting multiple devices in a cluster[62, 63, 64]. Both DS-CUDA[64] and rCUDA[63] enable access to remote GPUs by virtualizing the GPUs to the users. Moreover, DS-CUDA provides a fault tolerance mechanism. CUDASA[62] propose an extension to the CUDA programming model to support GPU-cluster environments. Similar approaches have been made to the OpenCL[5] programming model[65, 66, 67, 68, 69, 70, 71, 72]. SnuCL[69] is an OpenCL framework called SnuCL. It has a central node that executes the OpenCL host program and other compute nodes that execute commands from the host. DistCL[70] distributes OpenCL kernel workload across multiple devices in a cluster. However, a programmer must provide information to DistCL on how to partition the workload, resulting in source code modification. dOpenCL[68] and clOpenCL[67] provide wrapper functions for OpenCL host API functions for remote services.

Previous approaches in the above allow the programmer to utilize devices in a cluster as if they were in a single node. As a result, a programmer does not need to use an additional communication library to manage remote devices or transfer data between nodes. However, it still requires a programmer to distribute workloads and manage data between multiple devices. While it is similar that these approaches and SnuRHAC target clusters, SnuRHAC provides a more abstract view to a programmer because the programmer does not

need to consider multiple devices available.

There exist some previous studies related to using CUDA Unified Memory[24, 50, 73, 74, 75, 76, 77, 78, 79]. Balhaf et al.[73], Awan et al.[24], and Gera et al.[79] exploit UM to accelerate a specific application. Garg et al.[74] proposes checkpointing for UM. Ganguly et al.[75] and Yu et al.[76] propose a new page eviction policy for UM. Kim et al.[78] propose a technique for reducing GPU page fault handling time of UM. DRAGON[50] and GAIA[77] exploit the page fault mechanism of UM to allow NVM storage to be directly accessible from the GPU. While it is common that these approaches and SnuRHAC use UM, none of the previous approaches extend UM to work for a cluster environment.

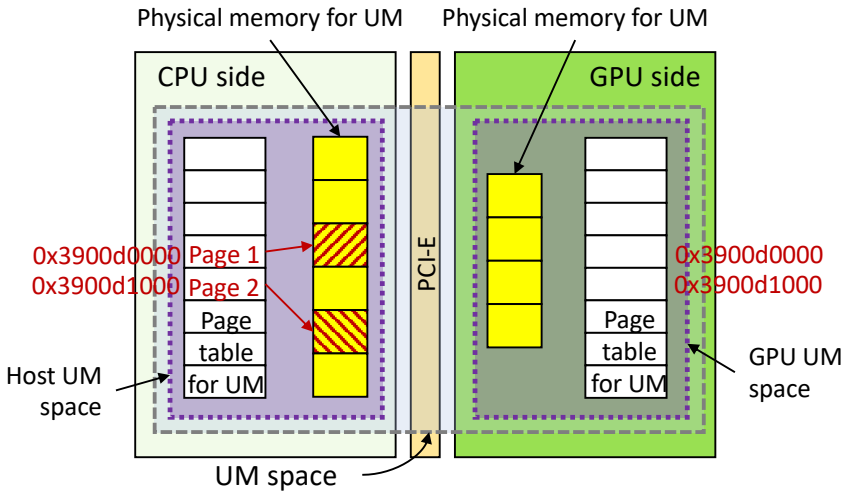
Chapter 3

CUDA Unified Memory

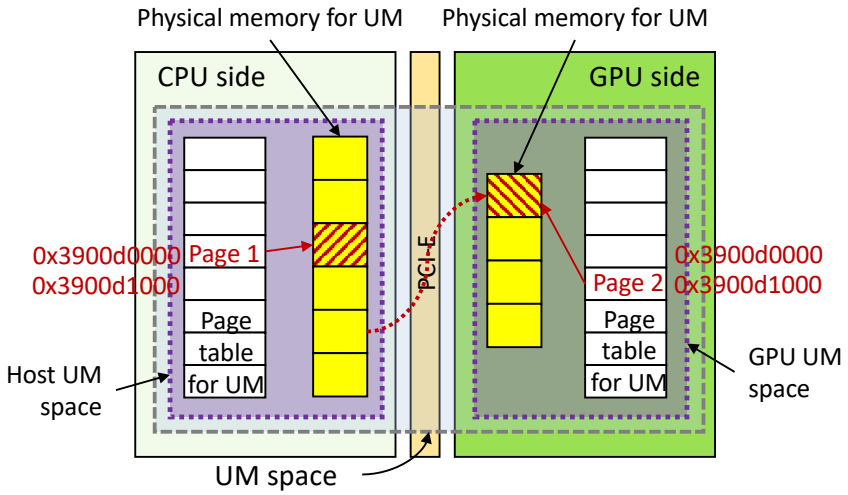
CUDA Unified Memory (UM) is a key component in this thesis. Therefore, before introducing the frameworks for UM, we introduce the details of UM in this chapter.

UM provides ease-of-programming by enabling CUDA programs to access the host memory and the GPU memory without the need to manually copy data from one to the other. UM behaves like the programmer has a single address space between the host and the GPU[80]. It allows a CUDA application to allocate memory objects that can be read or written from both the host and the GPU.

As shown in Figure 3.1(a), physical memory spaces are allocated to UM in both the host side and the GPU side. Pages in the host side space are pinned. UM page tables in the host side and the GPU side are managed by the CUDA runtime. To allocate a UM object, the CUDA program invokes `cudaMallocManaged()`, an allocation function that returns a pointer to the memory object. The pointer is accessible from both the host and the GPU.



(a) After the host has accessed page 1 and page 2.



(b) After the GPU has accessed page 2.

Figure 3.1: CUDA unified memory.

However, the memory object may not be physically allocated when the call to `cudaMallocManaged()` returns. In other words, the pages and page table entries of the memory object may not be created until it is accessed by the GPU or the CPU.

Pages in a UM object are automatically migrated between the host and the GPU on demand. This automatic page migration exploits page faults. The host reads and writes pages in the host memory and the GPU reads and writes pages in the device memory. The CUDA runtime takes care of the page migration, hence there is no need to call `cudaMemcpy()` or `cudaMemcpyAsync()` at all.

For example, suppose that a UM object has been allocated by `cudaMallocManaged()` and that the host has accessed two pages of the object, page 1 (at virtual address `0x3900d0000`) and page 2 (at virtual address `0x3900d1000`). Figure 3.1(a) shows the current status of page tables and physical memory spaces of UM. Now, suppose that the GPU accesses page 2 at virtual address `0x3900d1000`. Since page 2 is not residing in the GPU side, a page fault occurs and a page fault interrupt signal is raised. The page fault is handled by the NVIDIA display driver. It catches the signal and migrates the faulted page, page 2, between the host UM space and the GPU UM space as shown in Figure 3.1(b). Then, it makes the GPU replay the access. To avoid excessive page faults, the NVIDIA driver uses some heuristics for the page migration[80].

Figure 3.2 shows the diagram of page fault handling. When an NVIDIA GPU raises a page fault interrupt signal, the NVIDIA driver catches the interrupt signal and handles it. A fault buffer is a circular queue in the NVIDIA GPU. It stores faulted access information. The GPU can generate multiple faults concurrently, and there can be multiple fault entries for the same page in the fault buffer[80]. A UM block is a group of maximum 512 contiguous pages and a unit of management by the NVIDIA driver. The maximum size of a UM block is

$4KB \times 512 = 2MB$, and all pages in the same UM block are processed together by the NVIDIA driver. Each UM block object contains the information of all pages in the UM block, such as which processor has the pages and whether the pages are mapped with read protection or write protection. If a UM block contains a faulted page, we call the UM block *the faulted UM block* hereafter.

In Figure 3.2, the NVIDIA driver fetches the page addresses and access types of the faulted accesses from the fault buffer in the GPU (❶). Then, the NVIDIA driver preprocesses the faults (❷). It removes the duplicate addresses and groups them according to their UM blocks. Next, the NVIDIA driver checks the available GPU memory space for each faulted UM block (❸). If no GPU memory space is available for the faulted UM block, it evicts some pages from

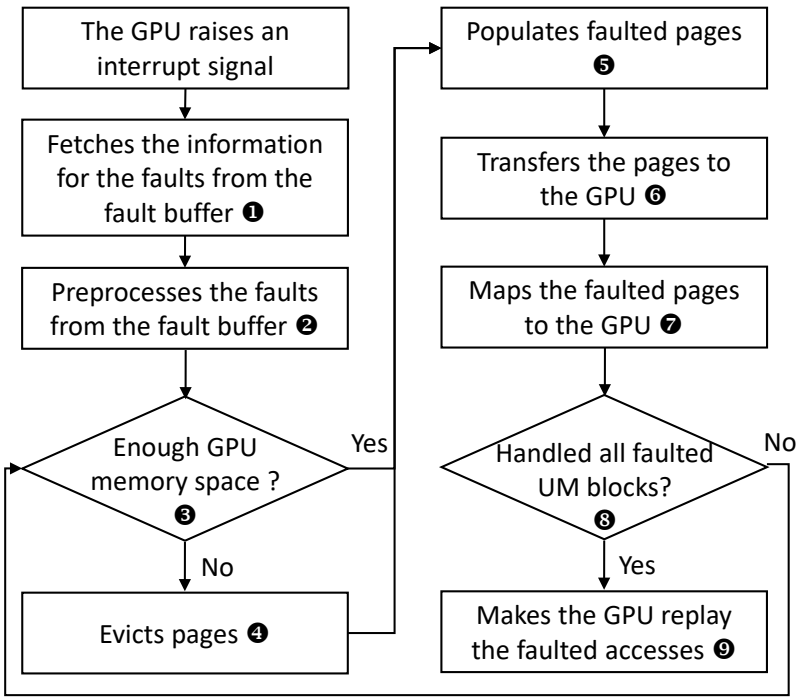


Figure 3.2: The behavior of NVIDIA page fault handler.

the GPU to the CPU (❹). Then, it populates faulted pages in the GPU (❺) (i.e., it allocates GPU memory space to the faulted pages), and it transfers the pages to the GPU (❻). When the transfer is done, the faulted pages of the UM block are mapped to the GPU (❼). This process repeats until all faulted UM blocks are handled (❽). Finally, the NVIDIA driver sends a replay signal to the GPU, and the fault handler finishes (❾).

In addition, representative CUDA commands[81] used in this thesis are summarized in Table 3.1.

Table 3.1: Representative CUDA commands used in this thesis.

`cudaError_t cudaMalloc(void** devPtr, size_t size)` allocates `size` bytes on the device and then returns in `*devPtr` a pointer to the allocated memory. It is a synchronous function.
`cudaError_t cudaMemcpy(void* dst, const void* src, size_t count, cudaMemcpyKind kind)` copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`, where `kind` specifies the direction of the copy. We are interested in `cudaMemcpyHostToDevice` as the value of `kind` in this paper.
`cudaError_t cudaMemcpyAsync(void* dst, const void* src, size_t count, cudaMemcpyKind kind, ...)` behaves the same as `cudaMemcpy()` except that it is asynchronous with respect to the host.
`cudaError_t cudaMallocManaged(void** devPtr, size_t size, ...)` allocates `size` bytes on the device and returns in `*devPtr` a pointer to the allocated memory that is automatically managed by the UM system.
`cudaError_t cudaMemPrefetchAsync(const void* devPtr, size_t count, int dstDevice, ...)` prefetches UM memory to the specified destination device. `devPtr` is the base pointer of the UM memory space to be prefetched and `dstDevice` is the destination device. `count` specifies the number of bytes to prefetch. It is asynchronous with respect to the host.

Chapter 4

Framework for Maximizing the Performance of Traditional CUDA Program

In this chapter, we present the design and implementation of HUM. HUM exploits the page fault mechanism of UM to automatically overlaps *host-to-device memory copy (H2Dmemcpy) and host computation* or *H2Dmemcpy and kernel computation* without any code modification.

4.1 Overall Structure of HUM

As shown in Figure 4.1, HUM consists of two components: *HUM runtime* and *HUM driver*. The NVIDIA driver[82] is a part of the CUDA framework that bridges the CUDA runtime and NVIDIA GPUs. It resides in the kernel address space. Similar to the NVIDIA driver, the HUM driver resides in the kernel address space. It intercepts signals going into the NVIDIA driver and takes some actions. Then, it calls appropriate NVIDIA driver functions for the signals

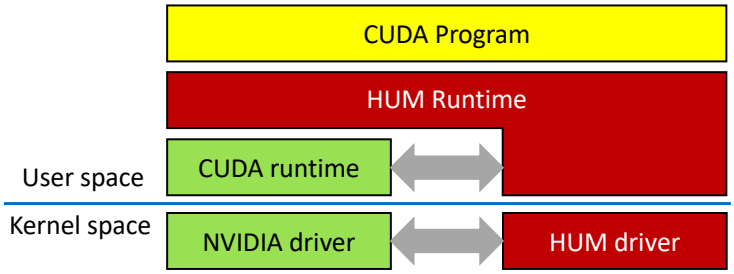


Figure 4.1: Components of HUM.

if needed. The HUM runtime is a thin layer that schedules CUDA commands and offloads command execution to the original CUDA runtime. It provides wrapper functions of CUDA API functions and interacts with the HUM driver and the CUDA runtime.

In CUDA, a stream is a sequence of commands that execute in issue-order on the GPU[83]. Commands in different streams may execute out of order with respect to one another or concurrently. When a CUDA program generates a request to create a new stream, the HUM runtime creates a stream object that is a wrapper of a new CUDA stream and provides it to the CUDA program. The HUM stream object is managed by HUM, and a host thread in the HUM runtime, called the *command scheduler*, periodically visits all existing streams in a round-robin manner. The HUM runtime also has several *worker threads*. When the command at the front of each stream is ready to execute, the command scheduler takes it from the stream and dispatches it to a worker thread. The worker thread executes the command (note that the command is actually the wrapper function of a CUDA command) and enqueues the CUDA command to the CUDA stream managed by the CUDA runtime. Finally, the CUDA runtime executes the command.

Basically, a wrapper in the HUM runtime for a CUDA API function calls

the original CUDA API function. For example, when the host program calls `cudaGetDeviceCount()`, which is a wrapper in the HUM runtime and returns the number of available devices, the wrapper calls original CUDA `cudaGetDeviceCount()`. Exceptions are the cases when the host program calls `cudaMalloc()` or `cudaMemcpy()`. When the host program calls `cudaMalloc()`, the HUM runtime allocates a memory space in the UM region by invoking CUDA `cudaMallocManaged()` to exploit the page fault mechanism of UM. This is why our framework is called HUM (Hidden Unified Memory). When the host program calls `cudaMemcpy()` for host-to-device memory copy, the wrapper in the HUM runtime invokes a custom memory copy function implemented in the HUM driver. Details are discussed in Section 4.5.

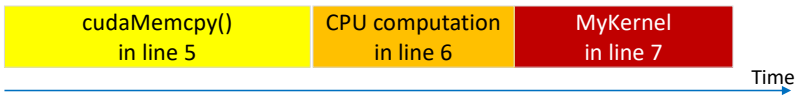
4.2 Overlapping H2Dmemcpy and Computation

Synchronous H2Dmemcpy. Figure 4.2 shows some examples of the memory copy commands. In Figure 4.2(a), the CUDA program allocates a host memory space, say hA, pointed to by `hostA` using `malloc()` in line 2 and a device memory space, say dA, pointed to by `devA` in line 3. It writes some data to hA in line 4. Then, it copies the contents of hA to dA by invoking synchronous `cudaMemcpy()` in line 5. After the memory copy has completed and some host computation has been performed in line 6, a kernel `MyKernel` that accesses dA is launched in line 7. Figure 4.2(b) shows the timeline of executing the code in Figure 4.2(a) under CUDA.

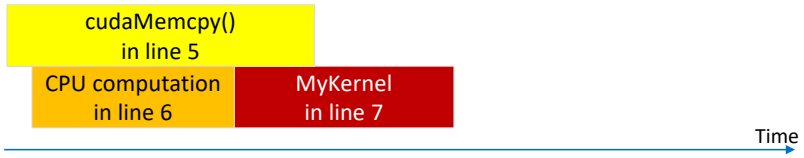
When the same code is executed under HUM, `cudaMemcpy()` returns immediately after initiating the memory copy even though the copy has not completed. This enables overlapping the memory copy in line 5 and the host computation in line 6. It may further overlaps the memory copy in line 5 and the

```
1: ...
2: hostA = malloc(size);
3: cudaMalloc(&devA, size);
4: ... // write to hostA
5: cudaMemcpy(devA, hostA, size, cudaMemcpyHostToDevice);
6: ... // some CPU computation
7: MyKernel<<<...>>>(devA);
8: ...
```

(a) Overlapping H2Dmemcpy and CPU computation and overlapping H2Dmemcpy and kernel computation.



(b) Executing the code in (a) under CUDA.



(c) Executing the code in (a) under HUM.

Figure 4.2: Example 1 of overlapping H2Dmemcpy and computation.

kernel execution in line 7. Figure 4.2(c) shows the timeline of executing the code in Figure 4.2(a) under HUM. Compared to the timeline under CUDA in Figure 4.2(b), `cudaMemcpy()` in line 5 is fully overlapped with the CPU computation in line 6 and partially overlapped with the kernel computation in line 7. As a result, the total execution time is significantly reduced.

Even though the kernel starts its execution before the memory copy in line 5 completes, the kernel correctly executes under HUM. The reason is that a page fault is raised at the device side when the kernel accesses a page that has not been copied yet to the device side. The page fault is handled by the HUM driver and it makes the kernel waits until the faulted page is copied to the device side. Then the page access request from the kernel is replayed.

However, if the host computation in line 6 modifies hA, the memory copy in line 5 and the host computation in line 6 may not be overlapped to guarantee data consistency and correctness. In this case, the timeline of executing the code in Figure 4.2(a) under HUM is the same as that under CUDA in Figure 4.2(b). The HUM runtime detects such a case using a simple runtime technique. The technique will be described later in Section 4.3.

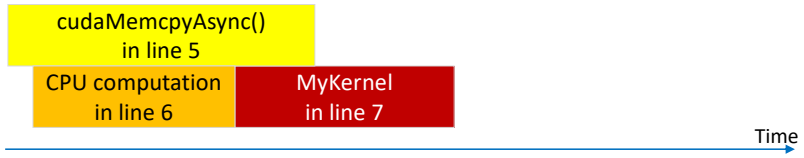
Asynchronous H2Dmemcpy. In Figure 4.3(a), the CUDA program calls asynchronous `cudaMemcpyAsync()` in line 5, hence the memory copy is performed in the background. As a result, the host side computation in line 6 can be overlapped with the memory copy in line 5. However, the kernel launched at line 7 cannot be overlapped with the memory copy in line 5 because all tasks placed in one stream are executed sequentially (the default behavior of CUDA). Figure 4.3(b) shows the timeline of executing the code in Figure 4.3(c) under CUDA.

When the same code is executed under HUM, even if the asynchronous mem-

```
1: ...
2: hostA = malloc(size);
3: cudaMalloc(&devA, size);
4: ... // write to hostA
5: cudaMemcpyAsync(devA, hostA, size, cudaMemcpyHostToDevice);
6: ... // some CPU computation
7: MyKernel<<<...>>>(devA);
8: ...
```

(a) Overlapping H2Dmemcpy and kernel computation.

(b) Executing the code in (d) under CUDA.



(c) Executing the code in (d) under HUM.

Figure 4.3: Example 2 of overlapping H2Dmemcpy and computation.

ory copy in line 5 has not finished yet, the GPU may start executing the kernel in line 7. This enables overlapping the H2Dmemcpy and the kernel computation for the same reason as the case of overlapping the synchronous H2Dmemcpy and computation mentioned above. Figure 4.3(c) shows the timeline of executing the code in Figure 4.3(a) under HUM. As a result, we see that the total execution time is significantly reduced.

Note that even though the HUM runtime overlaps the H2Dmemcpy and the host or kernel computation, it preserves the CUDA semantics of synchronization commands, such as `cudaDeviceSynchronize()`. `cudaDeviceSynchronize()` in the HUM runtime is also a wrapper function and invokes the original CUDA command.

```
01: ...
02: host_A = malloc(size);
03: cudaMalloc(&dev_A, size);
04: ... // write to host_A
05: cudaMemcpy(dev_A, host_A, size, cudaMemcpyHostToDevice);
06: ...
07: ... // write to host_A or free host_A
08: ...
09: MyKernel<<<...>>>(dev_A);
10: ...
```

Figure 4.4: A problematic scenario.

4.3 Data Consistency and Correctness

Consider the CUDA program in Figure 4.4. After performing `cudaMemcpy()` to copy the contents of the memory object, say hA, pointed to by `host_A` to the device memory object, say dA, pointed to by `dev_A` in line 5, the program mod-

ifies the contents of hA or frees hA in line 7. Under the CUDA semantics, this program has no problem at all. However, it may cause a problem under HUM. The data transfer caused by `cudaMemcpy()` to the device may still continue when the contents of hA is modified in line 7. Thus, the device may receive some pages that contain the modified contents. As a result, the kernel may access inconsistent and incorrect data.

To solve this problem, the HUM runtime exploits the access protection of pages using a POSIX function `mprotect()`[84] that changes the access protection of the memory pages of the calling process. When the H2Dmemcpy caused by `cudaMemcpy()` or `cudaMemcpyAsync()` is initiated, the HUM runtime changes the protection of pages in the source host memory object to *read-only*. For example, the protection of the pages in the object pointed to by `host_A` in Figure 4.4 is changed to *read-only* when the H2Dmemcpy of `cudaMemcpy()` is initiated.

After copying all pages in the source host memory object finishes, the HUM runtime tries to restore the protection. To do this, the HUM runtime first looks up all scheduled host-to-device memory copy commands and checks if there are commands that have an overlapping range of source addresses. If not, the HUM runtime restores the protection. Otherwise, the HUM runtime restores the protection of non-overlapping source memory regions only. The protection of overlapping source memory regions will be restored later when the following scheduled memory copy command finishes.

When the CUDA program in Figure 4.4 modifies a page in hA in line 7 in a manner (*e.g.*, write) that violates the protection, the linux kernel generates a SIGSEGV signal. The signal handler installed by the HUM runtime handles the signal. When it receives the signal, it waits until all H2Dmemcpy commands for the page complete and the protection is restored. This method allows the

HUM runtime to execute H2Dmemcpy commands in an asynchronous manner without any data consistency violation or any segmentation fault.

4.4 HUM Driver

Intercepting interrupts. To overlap H2Dmemcpy and kernel execution, HUM makes the GPU pend when the page accessed by the GPU has not been transferred to the GPU yet. In this case, a GPU page fault occurs in HUM. The HUM driver handles the page fault. The HUM driver hooks the interrupt handler of the NVIDIA display driver and intercepts the page fault signal. In Linux for the x86 architecture, the *interrupt descriptor table* (IDT) contains all information about interrupts, such as interrupt number, interrupt name, address of the interrupt handler, interrupt flags, etc. When the HUM driver is installed, HUM looks up the existing IDT entries and finds the entry for the NVIDIA interrupt handler. HUM replaces the entry with the information of its own interrupt handler.

Handling page faults. Figure 4.5 shows the actions occurring when the HUM interrupt handler handles a page fault in the GPU side. When the HUM interrupt handler receives an interrupt signal, it checks the fault buffer in the GPU if there is a pending GPU page fault. The fault buffer is a circular queue implemented in the GPU by NVIDIA. It stores page faults information from the GPU. If there is no pending fault in the fault buffer, the HUM interrupt handler invokes the original NVIDIA interrupt handler because the interrupt is not a page fault and there is nothing to do for the HUM interrupt handler. Otherwise, it checks whether faulted pages are allocated through `cudaMalloc()` call or `cudaMallocManaged()` call in the host program.

For pages that are allocated through `cudaMalloc()` call, the HUM driver

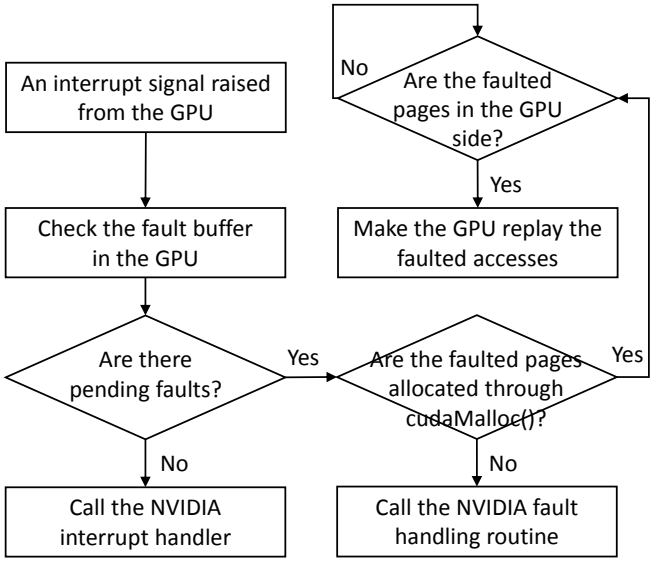


Figure 4.5: Actions of the HUM interrupt handler.

handles the fault. The HUM interrupt handler waits until all the faulted pages arrive and are mapped to the GPU. Then, the HUM driver sends a replay signal to the GPU so that the GPU replays the faulted memory accesses. For pages that are allocated through `cudaMallocManaged()` call, the HUM driver calls the fault handling routine in the CUDA display driver. Thus, the programmer can use both `cudaMalloc()` and `cudaMallocManaged()` in their host program without causing any problem. However, HUM does not optimize the memory transfers for the memory regions that are allocated through `cudaMallocManaged()` calls in the host program.

4.5 HUM H2Dmemcpy Mechanism

When the GPU accesses a page that has not been copied from the host side to the GPU side, the HUM runtime makes the GPU waits until the page arrives. As a result, a kernel can be executed even the transfer of the data to be accessed

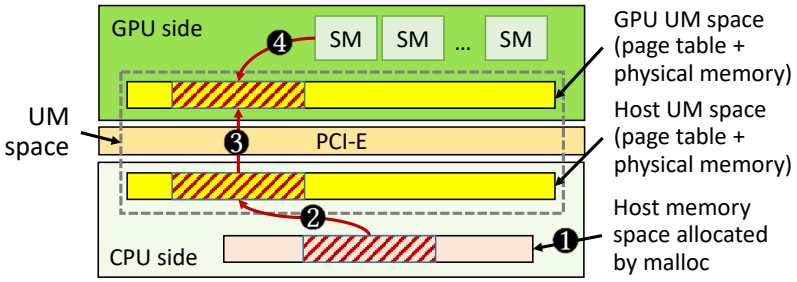


Figure 4.6: How the HUM H2Dmemcpy function works.

by the kernel is still ongoing. However, to implement the H2Dmemcpy in HUM, we may not use `cudaMemcpy()` and `cudaMemcpyAsync()` because they cause a serious interrupt handling problem.

Problems of CUDA memory copy commands. For example, suppose that the HUM runtime uses `cudaMemcpy()` to copy data from the host to the device and that the GPU is trying to read a page that has not been yet copied to the GPU side. Then, a read page fault is raised and the HUM driver catches it. The HUM driver waits until the page comes to the GPU side. Calling `cudaMemcpy()` triggers write page fault when the page arrives at the GPU because the page has not been mapped to the GPU yet. The HUM driver catches the write page fault and maps a blank page to the GPU UM space. Then, it sends a replay signal to the GPU. This makes the GPU reads stale data in the blank page. In turn, the page arrived updates the GPU UM space. Since interrupts caused by memory requests are processed sequentially one by one in the GPU, the kernel reads the stale data in the blank page first, and the page update by the memory copy follows this read. To get the correct result, the memory copy should have completed before the kernel reads the stale page. However, changing the order of interrupt processing is not supported by the current NVIDIA driver.

HUM H2Dmemcpy functions. To solve this problem, the HUM driver has its own H2Dmemcpy function. Figure 4.6 shows how the HUM H2Dmemcpy function works. A CUDA program first writes data to the host memory space that is generally allocated through `malloc()` (❶). Suppose that the program uses `cudaMemcpy()` or `cudaMemcpyAsync()` to perform the H2Dmemcpy. As mentioned before, the HUM runtime implements wrappers of `cudaMemcpy()` and `cudaMemcpyAsync()`. In the wrappers, the HUM runtime calls the HUM driver rather than calling the original CUDA `cudaMemcpy()` or `cudaMemcpyAsync()`.

The HUM driver first copies the data from the host memory space to the host UM space (❷ in Figure 4.6). This makes pages and page table entries of the memory object to be created on the host side, effectively increasing the required memory space on the host side as much as the size of the host UM space. Then, it invokes the page migration function provided by the NVIDIA driver to migrate the pages in the host UM space to the GPU UM space (❸). To use the migration function, source pages of the migration must reside in the host UM space. The page migration function is synchronous and migrates maximum 512 pages at a time, *i.e.*, maximum 2 MB at a time. When the migration completes, the pages are mapped to the GPU, and the GPU can access the pages without any page fault (❹).

When there is a H2Dmemcpy request of size M MB ($M > 2$), the HUM driver divides the request into multiple requests of size 2 MB. We take the maximum size because frequent memory-copy requests cause heavy copy initiation overhead.

4.6 Parallelizing Memory Copy Commands

Consider a vector addition CUDA program in Figure 4.7. It adds two vectors A and B, and the result is stored in vector C. Figure 4.7 shows timelines of executing the program. Since the memory copy command and the kernel execution command are issued in the same stream to guarantee correctness, they are sequentially executed as shown in Figure 4.7(a) under CUDA semantics.

The timeline of executing the vector addition program under the HUM design discussed so far is shown in Figure 4.7(b). HUM may execute the kernel as early as possible when the memory copy for vector B has initiated. As a result, the time when the kernel completes under HUM maybe much earlier than that under normal CUDA. Since the page migration function provided by NVIDIA driver used in the HUM H2Dmemcpy function is synchronous, the memory copy for vector B has to be initiated after the memory copy for vector A has completed.

Using the HUM H2Dmemcpy function, the time spent on memory copying is much larger than using `cudaMemcpy()` or `cudaMemcpyAsync()`. This is because HUM copies the data twice: from the host memory space to the host UM space, and then to the GPU UM space.

To reduce the copy time from the host memory to the host UM space (❷ in Figure 4.6), HUM exploits multiple host threads for the memory copy. The multiple threads simultaneously copy different parts of the source host memory to the host UM space. HUM divides the source host memory object into multiple 2MB chunks and each thread takes care of copying a 2MB memory chunk to the host UM space at a time. While this approach reduces the copy time from the host memory to the host UM space, it may result in interference with other CPU threads depending on the application.

```
01: ...
02: host_A = malloc(size);
03: host_B = malloc(size);
04: host_C = malloc(size);
05: ... // write to host_A and host_B
06: cudaMalloc(&dev_A, size);
07: cudaMalloc(&dev_B, size);
08: cudaMalloc(&dev_C, size);
09: ...
10: cudaMemcpyAsync(dev_A, host_A, size,
11:                 cudaMemcpyHostToDevice);
12: cudaMemcpyAsync(dev_B, host_B, size,
13:                 cudaMemcpyHostToDevice);
14: ...
15: vec_add<<<...>>>(dev_A, dev_B, dev_C);
16: ...
```

Figure 4.7: Vector addition CUDA program.

4.7 Scheduling Memory Copy Commands

When more than one CUDA H2Dmemcpy commands are issued consecutively from a CUDA program, the HUM runtime copies their divided 2MB chunks from the host UM space to the device UM space in a round-robin manner. In the HUM runtime, there is a pool of *page migration queues* (PMQs) to queue the page migration requests of 2MB chunks. Moreover, there exists a different PMQ for each CUDA H2Dmemcpy command issued.

For a H2Dmemcpy command from the CUDA program, after dividing the source host memory object into 2MB chunks and copying them to the host UM space with multiple threads, the page migration request of each chunk from the host UM space to the GPU UM space is inserted in the associated PMQ. A host thread called the *page migration thread* (PMT) is taking care of visiting non-empty PMQs in the pool in a round-robin manner. The PMT processes the page migration request at the head of each PMQ by calling the page migration function provided by the NVIDIA driver.

In this case, there must not exist any dependence between destination locations of the consecutively issued CUDA H2Dmemcpy commands. Since the HUM runtime has all information about the CUDA H2Dmemcpy commands issued from a CUDA program, it performs a simple and conservative address range overlapping check between the destinations of memory copy commands. Note that at run time, the real addresses are known. When CUDA H2Dmemcpy is enqueued to the HUM runtime, the runtime checks if there is an in-flight memory copy command that has an overlapping range of destination addresses with the enqueued command. If there is no overlapping, the HUM runtime schedules the memory copy command normally. Otherwise, it pends scheduling the memory copy command until the in-flight memory copy command finishes. However,

such dependences are hardly found in real applications (none in our benchmark applications).

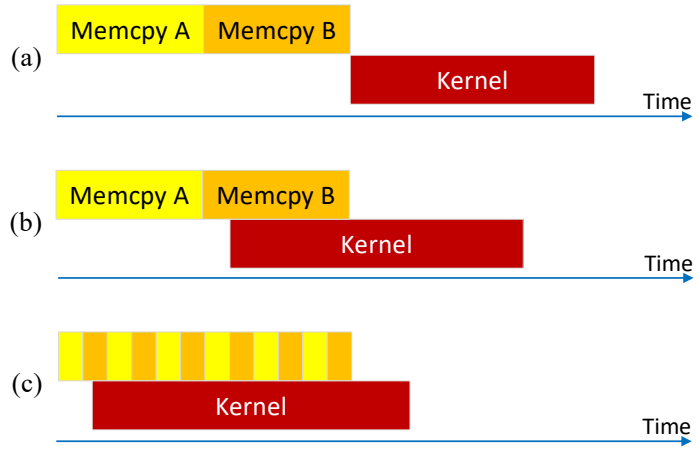


Figure 4.8: Executing the vector addition program in Figure 4.7.

By doing so, we can schedule the kernel launch as early as possible. As a result, the kernel may access required pages sooner and its execution may finish earlier. This case is illustrated in Figure 4.8(c). The kernel execution can be initiated after the execution of the H2Dmemcpy command of the vector B has been initiated. In general, with regards to H2Dmemcpy commands, the execution of a kernel command K under HUM can be initiated as early as possible at the time point that satisfies all of the following conditions:

- The last command preceding K in the same stream is a CUDA H2Dmemcpy command, say C, on which K's arguments depend.
- The execution of C has been initiated.
- All target pages of C in the device UM space have been unmapped once to the GPU after the initiation of executing C.

Chapter 5

Framework for Running Large-scale DNNs on a Single GPU

In this chapter, we present the design and implementation of DeepUM. DeepUM allows GPU memory oversubscription for deep neural networks by exploiting Unified Memory and using CPU memory as a backing store.

5.1 Structure of DeepUM

Figure 5.1 shows the overall structure of DeepUM. It consists of the DeepUM runtime and the DeepUM driver. The driver is a Linux kernel module. DeepUM targets PyTorch, one of the most popular deep learning frameworks, and PyTorch runs on top of the DeepUM runtime.

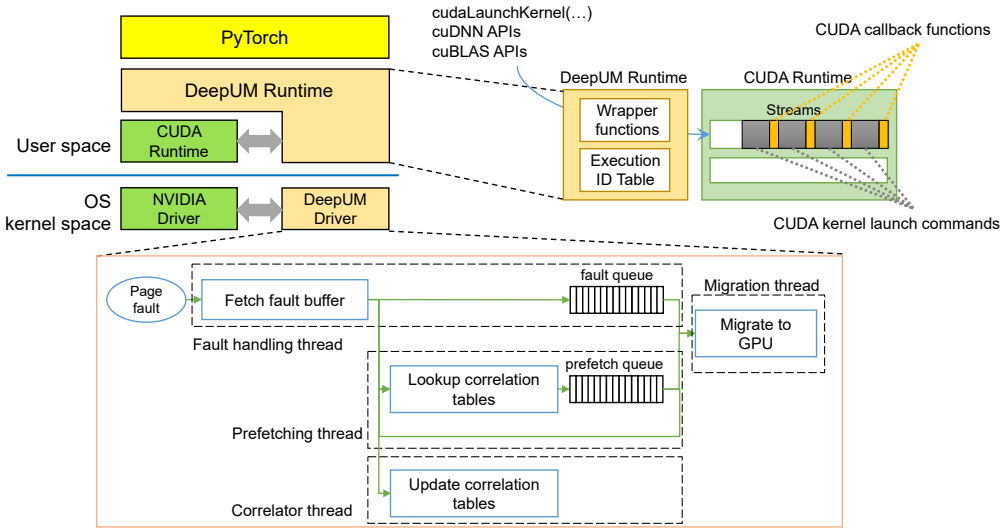


Figure 5.1: Overall structure of DeepUM.

5.1.1 DeepUM Runtime

The DeepUM runtime provides wrapper functions for CUDA memory allocation API functions to switch all GPU memory allocation requests to UM space allocation requests. It easily accomplishes GPU memory oversubscription by allocating all GPU memory objects in the UM space. Moreover, the DeepUM runtime provides wrapper functions for CUDA kernel launch commands and other CUDA library functions, such as those in cuDNN and cuBLAS. Note that CUDA library functions also launch CUDA kernels.

The DeepUM runtime manages a table called the *execution ID table*. The table holds kernel launch history and contains the hash value of each kernel's name and arguments. When a new kernel launch command comes to the DeepUM runtime, it computes the hash value of the kernel name and arguments. Then, it looks up the execution ID table to find the command of the same hash value. If it finds a matching command, it gives the same *execution ID* to the kernel.

Otherwise, it assigns a new execution ID to the kernel and saves the information in the table. Finally, the DeepUM runtime enqueues a CUDA callback function to the CUDA runtime just before enqueueing the kernel launch command. The callback function passes the execution ID of the following kernel launch command to the DeepUM driver through the Linux `ioctl` command. The DeepUM driver uses the passed execution ID for correlation prefetching.

5.1.2 DeepUM Driver

The DeepUM driver handles GPU page faults and prefetches pages to the GPUs. We observe that the kernel execution patterns and the memory access patterns within the kernels are fixed and repeated in the training phase of a DNN. Thus, memorizing the repeated patterns and exploiting the information for prefetching is desirable. Correlation tables managed by the DeepUM driver record the history of the kernel executions and their page accesses during the training phase of a DNN. The DeepUM driver prefetches pages based on the information in the correlation tables by predicting which kernel will execute next. In Section 5.2.2, we will describe the correlation prefetching mechanism used by the DeepUM driver.

There are four kernel threads in the DeepUM driver: *fault handling thread*, *correlator thread*, *prefetching thread*, and *migration thread*. The *fault handling thread* handles GPU page faults using the functions implemented in the NVIDIA driver, such as accessing the fault buffer and sending replay signals to the GPU. The NVIDIA GPU has a hardware fault buffer that is a circular queue. It accumulates the information for faulted accesses. The DeepUM driver intercepts page fault interrupt signals to the NVIDIA driver, and the fault handling thread reads the fault buffer. The fault handling thread passes the information of faulted accesses to the other three threads.

The *fault queue* is a single-producer/single-consumer queue that stores the UM block addresses of the faulted pages. It holds the highest priority items to be handled in the driver to make the GPU replay the faulted accesses as soon as possible.

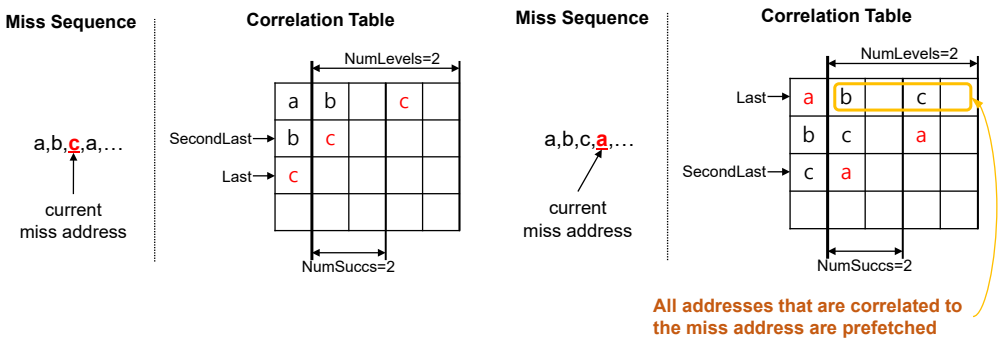
The *correlator thread* manages correlation tables. It updates the correlation tables based on the fault information from the fault handling thread. We will discuss the structure of the correlation tables and how the correlator thread updates them in Section 5.2.2.

The *prefetching thread* looks up the correlation tables and calculates the UM block addresses for prefetching with the faulted block address. Then, it enqueues the prefetch commands to the *prefetch queue*, a single-producer/single-consumer queue. A prefetch command consists of a UM block address to prefetch and the execution ID for which the corresponding UM block is predicted to be used.

Finally, the *migration thread* migrates the UM blocks between the CPU and the GPU. The fault queue has a higher priority than the prefetch queue. It first handles commands from the fault queue managed by the fault handling thread. When the fault queue is empty, it handles commands from the prefetch queue managed by the prefetching thread.

5.2 Correlation Prefetching for GPU Pages

Correlation prefetching[85, 86, 87, 88, 89] was originally developed for cache-line prefetching. DeepUM modifies the original correlation prefetching to adapt it to prefetching pages for DNN workloads. There are two methods in the original correlation prefetching: *stride-based* and *pair-based*. The stride-based correlation prefetching[87] finds stride patterns in the sequence of missed addresses, while the pair-based correlation prefetching[85, 86, 88, 89] finds a corre-



(a) (b)

Figure 5.2: Pair-based correlation prefetching.

lation between missed addresses. DeepUM is based on the pair-based correlation prefetching technique.

5.2.1 Pair-based Correlation Prefetching

The pair-based correlation prefetching records past sequences of missed addresses in a correlation table. When a cache miss occurs, it looks up the correlation table and prefetches all the addresses correlated with the missed address.

Figure 5.2 shows an example of pair-based correlation prefetching for cache lines. In Figure 5.2(a), we suppose that cache misses have occurred for addresses a and b. Following them, accessing address c causes a cache miss. A different missed address has a different row in the table. Each row of the correlation table has N-way associativity (Assoc) to reduce address conflict between UM blocks that map to the same row. In Figure 5.2, we assume associativity is one for easy understanding. For each set, there are NumLevels levels of successor miss addresses. In each level, NumSucc entries are MRU ordered from left to right. Last (points to c) and SecondLast (points to b) are pointers to the entries for the last and second last misses, respectively. The entry for a has b in the first

level because the miss for b occurred right after a. It also has c in the second level because the miss for c indirectly follows the miss for a after the miss for b.

In Figure 5.2(b), we suppose that the cache miss for a occurrs again after the miss for c. The entries for b, c, and the Last and SecondLast pointers are updated. At this point, since there exist successor entries recorded for a, all the entries in the row of a (b and c) are prefetched from the memory in addition to accessing a in the memory.

5.2.2 Correlation Prefetching in DeepUM

Correlation prefetching in the DeepUM aims to reduce page faults by prefetching pages expected to be accessed by CUDA kernels. Unlike the original correlation prefetching, DeepUM uses two types of correlation tables: *execution ID* and *UM block*. Both types of tables have a single level ($NumLevels = 1$) because the prefetching thread does chaining. Moreover, there is no reason to maintain multiple levels because of the characteristics of the DNN workloads. Note that DeepUM's correlation prefetching works at the UM block level rather than cache-line or page level.

Execution ID	Record 0	Record 1	Record 2	...
0	(7, 9, 92, 75)			
1	(89, 53, 24, 10)	(34, 52, 22, 99)	(4, 3, 939, 2)	
2	(3, 53, 4, 8)	(33, 588, 34, 1)		
...				

Figure 5.3: An execution ID correlation table.

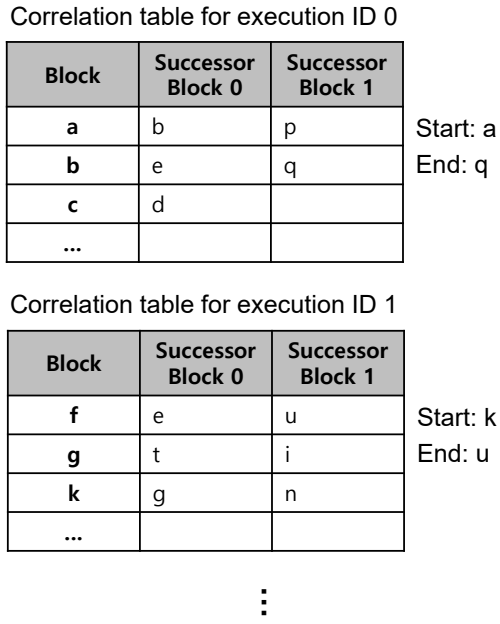


Figure 5.4: UM block correlation tables.

Execution ID correlation table. The execution ID correlation table (the execution table in short) records the execution history of the execution IDs of CUDA kernels, and only a single table exists. Execution IDs come from the DeepUM runtime. Figure 5.3 shows an example of an execution table. Each entry of the execution table holds sets of correlated execution IDs. Each set consists of four execution IDs. The first three IDs represent the previously executed kernels right before the last kernel (currently executing kernel when updating the table). Thus, the last ID represents the next kernel to execute when prefetching occurs. For example, the entry for the execution ID *0* has a record of (*7*, *9*, *92*, *75*). This record in the entry for the execution ID *0* implies that the kernels with execution IDs *7*, *9*, and *92* have been executed, and the kernel with execution ID *0* is currently executing. It predicts the execution ID of the kernel to be executed next as *75*.

The number of records each entry contains is variable, i.e., the number of the successor kernels of a kernel is variable. Thus, each entry can hold all history of successor kernels' execution IDs. DeepUM chooses this scheme to predict the next kernel to be executed as accurately as possible. Even though the incorrect result of predicting the next UM block to be accessed is not that costly, the inaccurate result of predicting the next kernel to be executed is expensive.

UM block correlation table. The NVIDIA driver manages the pages in the CUDA UM space by grouping them into multiple UM blocks. A maximum of 512 contiguous pages are grouped into a UM block. Rather than recording the history of faulted page addresses, a UM block correlation table (a block table in short) records the history at the UM block level. There are two reasons for choosing this granularity. One is that there are too many page addresses to manage for large-scale DNNs. The other is that making DeepUM has the same page management granularity as the NVIDIA driver is more efficient. Note that the NVIDIA driver manages the pages in the granularity of a UM block.

Figure 5.4 shows an example of block tables. A block table exists for each execution ID and records a history of UM block accesses within the corresponding CUDA kernel. It is similar to the table used in the original correlation prefetching. However, a block table contains the address of the *end* UM block that points to the last UM block prefetched, and the *start* UM block that points to the first faulted UM block in the corresponding kernel execution. These two pointers are used to implement chaining. Both the start UM block and end UM block are captured during the transition of the currently executing kernel. End UM block is the UM block where the page resides that lastly faulted right before execution ID transition. Start UM block is the UM block where the first faulted page resides that occurred right after the execution ID transition.

Prefetching mechanisms and chaining. When a page fault occurs, the DeepUM driver prefetches all pages in the UM blocks correlated to the faulted UM block by looking up the UM block correlation table of the currently executing kernel.

When the prefetching thread in DeepUM meets the UM block that is the same as the end block in the UM block correlation table, it ends prefetching for the kernel and predicts the kernel that will execute next by looking up the execution ID table. Then, it starts prefetching for the predicted kernel, beginning with the start UM block in the UM block correlation table of the predicted kernel.

Consider the block tables in Figure 5.4, suppose that DeepUM is prefetching the UM block b for the kernel with execution ID *0*. Also suppose that the kernel with execution ID *1* will be executed right after the kernel with execution ID *0*. The successor UM blocks for b are e and q. Since block q is the same as the end UM block for execution ID *0*, the prefetching thread stops prefetching for the kernel with execution ID *0*. Then it starts prefetching block k (the start UM block of the block table for execution ID *1*).

This process is called chaining. It is the process of continuously calculating UM block addresses for prefetching after a page fault has occurred. The chaining ends when a new page fault interrupt signal is raised, or the prefetching thread fails to predict the next kernel to execute. The chaining pauses when the prefetching thread has enqueued all prefetch commands for the next N kernels. The prefetching thread resumes after the currently executing kernel finishes.

5.3 Optimizations for GPU Page Fault Handling

In this section, we describe the optimization techniques for GPU page fault handling.

5.3.1 Page Pre-eviction

Page eviction occurs when the driver fails to allocate GPU memory space for migrating faulted pages. Figure 5.5 shows a scenario when page eviction occurs. Page eviction occurs when there is no GPU memory space available for the faulted pages to handle the faulted pages. Page eviction takes significant time, implying that the page eviction logic lies on the critical path of page fault handling. Thus, fault handling time increases when no more space is available on the GPU.

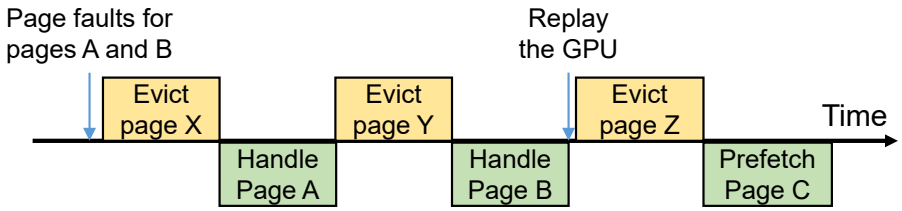


Figure 5.5: Page eviction scenario.

To minimize the fault handling time, the DeepUM driver pre-evicts pages when no free GPU memory space is left. A similar idea is proposed by Kim et al.[78]. The major difference is that DeepUM uses a different policy to select victim pages. The policy used by Kim et al.[78] is the same as the policy implemented in the NVIDIA driver. It evicts pages that are least recently migrated to the GPU. The DeepUM driver evicts pages that satisfy the following two conditions:

- Least recently migrated.

- Not expected to be accessed by the currently executing kernel and the next N kernels predicted to execute.

Since the NVIDIA driver tracks the free spaces on the GPU side, the DeepUM obtains the available space information from the NVIDIA driver. It obtains the next executing kernel information from the execution ID correlation table.

5.3.2 Invalidating UM Blocks of Inactive PyTorch Blocks

PyTorch has different memory allocators for CPUs and GPUs. PyTorch's GPU memory allocator manages device memory pools to minimize memory allocation/free time and to reduce memory fragmentation. Two types of memory pools are managed by the GPU memory allocator: *large* and *small*.

A memory object in PyTorch is called a block. In this thesis, we call it the *PT block* to distinguish it from a UM block. The large pool consists of PT blocks larger than 1MB, and the small pool consists of PT blocks less than or equal to 1MB. When a memory allocation request comes in, and the requested size is larger than 1MB, the memory allocator finds a PT block from the large pool. Otherwise, it finds a PT block from the small pool. When multiple PT blocks in the pool match the requested size, the allocator returns the smallest available PT block. In addition, the PT block is split when its size is much larger than the requested size. The selected PT block is removed from the memory pool and marked active. However, when no PT block is available in the memory pool, the GPU memory allocator allocates a new PT block by requesting a device memory space to the CUDA runtime.

After the PT block has been used by the DNN model and returned to the GPU memory allocator, the allocator inserts the PT block into an appropriate memory pool and marks it inactive. Inactive PT blocks, i.e., PT blocks in the memory pools, are freed to produce a new memory space only when no available

memory space is left in the pool.

The problem arises when we use the PyTorch memory allocator with UM. When inactive PT blocks on the GPU memory are evicted to the CPU memory, unnecessary heavy data traffic occurs. In addition, they occupy CPU memory space. The problem worsens when the inactive PT blocks are marked as active and used by DNN models again. Since the pages in the PT blocks have been evicted to the CPU memory, they should be migrated to the GPU again, resulting in heavy data traffic.

To solve this problem, we add a few lines of code to the PyTorch memory allocator to tell the DeepUM driver when a PT block is marked inactive. If a victim page belongs to an inactive PT block, the DeepUM driver simply invalidates the corresponding UM block in the GPU memory.

Chapter 6

Framework for Virtualizing a Single Device Image for a GPU Cluster

In this chapter, we describe the design and implementation of SnuRHAC.

6.1 Overall Structure of SnuRHAC

Figure 6.1 shows the overview of SnuRHAC architecture. SnuRHAC targets a cluster where each node is equipped with multiple NVIDIA GPUs. Also, each node is connected with a network that supports remote direct memory access (RDMA)[90]. The operating system kernel has the NVIDIA device driver[82] and the SnuRHAC driver. The NVIDIA driver allows CUDA runtime to control GPUs. It also handles various interrupt signals from GPUs, including page-fault interrupt signals. The SnuRHAC driver extends CUDA UM to the cluster by hooking the NVIDIA driver's page fault handler. The SnuRHAC driver is installed in every node and communicates with each other through RDMA. In

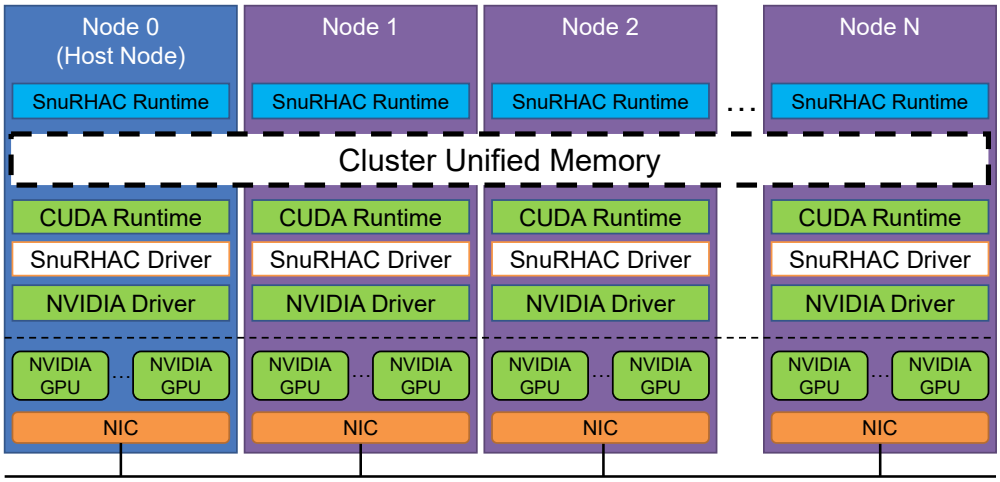


Figure 6.1: Overview of SnuRHAC architecture.

the user space, the CUDA runtime and the SnuRHAC runtime run. In node 0, the host node, a CUDA program runs on top of the SnuRHAC runtime. SnuRHAC runtime provides wrapper functions of CUDA API functions, and the CUDA program calls these wrapper functions.

To run a CUDA program using SnuRHAC, the user needs two steps. First, install the SnuRHAC driver in each node of the cluster. Second, run the CUDA program through `mpirun` command with `LD_PRELOAD` environment variable set. The `mpirun` command runs the SnuRHAC runtime in each node. Every MPI library provides this command. Setting `LD_PRELOAD` environment variable loads the SnuRHAC runtime library before loading the CUDA runtime library. This makes all CUDA API function calls from the user CUDA program directed to the SnuRHAC runtime, not the CUDA runtime. As a result, no user code modification is required, and very minimum effort is required to run the CUDA program using SnuRHAC.

When a user runs a CUDA program under SnuRHAC, the SnuRHAC run-

time shows a single virtual GPU to the program. The program calls wrapper functions provided by the SnuRHAC runtime. When the program calls a wrapper function in the host node, the SnuRHAC runtime in the host node sends appropriate commands to the other nodes. The SnuRHAC runtime in the other nodes receives the commands and handles them. Notable wrapper functions are the function that allocates device memory(e.g., `cudaMalloc()`), the function that copies to/from device memory(e.g., `cudaMemcpy()`) and the function that launches CUDA kernels(e.g., `<<<...>>>()`).

`cudaMalloc()`. When the host program calls `cudaMalloc()` to allocate a device memory space, the SnuRHAC runtime in every node allocates a UM space by calling `cudaMallocManaged()`. With SnuRHAC driver, multiple GPUs in the cluster see the same address space and share data through UM. Details are discussed in Section 6.3.

`cudaMemcpy()`. When the host program calls `cudaMemcpy()`, the SnuRHAC runtime copies data to/from the host UM space at the host node. To copy data, the SnuRHAC runtime exploits a maximum of eight host threads. The multiple host threads simultaneously copy different parts of the source host memory to the host UM space. When the program requests to copy data from the device memory to the host memory, the SnuRHAC runtime in the host node copies data from the host UM space to the host memory space. Pages that do not reside on the host UM space will be automatically migrated by the page fault mechanism. On the other hand, when the program requests to copy data from the host memory to the device memory, the SnuRHAC runtime in the host node copies data from the host memory space to the host UM space. When each GPU in the cluster executes the CUDA kernel and if a page accessed is not resident on the GPU, the SnuRHAC UM system automatically migrates

faulted pages from the host UM space to the UM space of the faulted GPU. The details are discussed in Section 6.3.

Kernel launches. Finally, when the host program request to launch a CUDA kernel, the SnuRHAC runtime in the host node partitions the workloads and sends kernel launch commands to all nodes. It also issues prefetch commands before each kernel executes. The details are discussed in Section 6.2 and Section 6.5.

6.2 Workload Distribution

Partitioning the workload. The SnuRHAC runtime automatically partitions the workload and distributes it to multiple GPUs in a cluster. In other words, it partitions the CUDA grid specified in the execution configuration of the kernel launch command and obtains multiple chunks of thread blocks. Then, it distributes the chunks to multiple GPUs. There are many ways to partition the grid in the unit of a thread block. Figure 6.2 shows many ways of partitioning a two-dimensional grid for four GPUs.

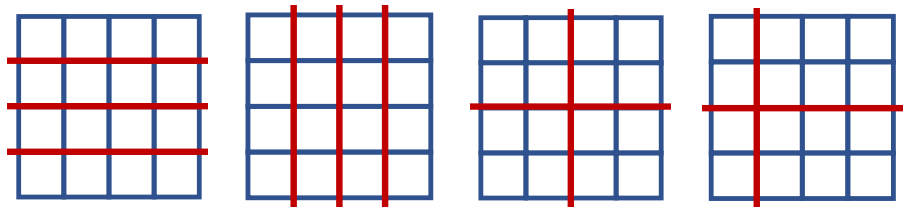


Figure 6.2: Partitioning a two-dimensional grid for four GPUs.

Each CUDA thread has its unique thread ID. The GPU groups consecutive 32 threads in a block based on their thread IDs and execute the threads in

this group simultaneously. This group of threads is called a *warp*. To maximize DRAM bandwidth utilization, the GPU coalesces global memory loads and stores issued by the threads in a warp into as few transactions as possible. Most of the CUDA program is designed to coalesce global memory access in the x-dimension of the thread block because of the thread ID calculation rule in CUDA. Moreover, adjacent blocks have a high possibility to access the same page. Thus, it is desirable to have contiguous thread blocks in x-dimension in a partitioned chunk of blocks.

Based on the above observation, SnuRHAC has the following grid partitioning algorithm:

1. If the number of blocks in z-dimension is greater than or equal to the number of GPUs in the cluster, partition the grid in z-dimension.
2. Otherwise, if the number of blocks in y-dimension is greater than or equal to the number of GPUs in the cluster, partition the grid in y-dimension.
3. Otherwise, partition the grid in x-dimension.

When the grid of a kernel does not have enough blocks in a certain dimension, SnuRHAC looks for other dimensions with lower priority. Here, "enough blocks" means that the number of blocks is greater than or equal to the number of GPUs in the cluster. Also, we assume that the computing power of GPUs is all the same. Thus, it is best to distribute the workload on the GPUs equally. Distributing the workload to the GPUs with different computing powers is beyond the scope of this paper.

In Figure 6.2, the SnuRHAC runtime will choose (a) as a final partition for the two-dimensional grid. Since the grid is two-dimensional, the first step of the partitioning algorithm does not apply. Since there are enough blocks in the y-dimension of Figure 6.2(a), the SnuRHAC runtime will partition the grid in the

y dimension in the second step of the partitioning algorithm. The SnuRHAC runtime will partition the grid in Figure 6.2(b) in the x dimension even though there are enough blocks in the y dimension. The partitions in Figure 6.2(c) and Figure 6.2(d) cannot happen by following the partitioning algorithm. The SnuRHAC runtime partitions the grid in only one dimension.

Kernels for the distributed workload. SnuRHAC generates a kernel to make each GPU execute the partitioned grid, not the original grid. To create the kernel, it modifies the original kernel at the PTX level. PTX is a low-level intermediate code of CUDA kernels. It is compiled to the machine code at run time and runs on NVIDIA GPUs. Unless the programmer specifies not to embed the PTX code, the NVIDIA CUDA compiler always embeds the PTX code to the output CUDA executable. SnuRHAC extracts the PTX code from the CUDA executable and modifies the kernels to receive additional kernel arguments from the SnuRHAC runtime. The kernel filters out the thread blocks that will be executed on other GPUs.

6.3 Cluster Unified Memory

SnuRHAC provides an additional Linux kernel module called the SnuRHAC driver to support UM for a cluster. It keeps track of the location and status of all pages in the UM space. Also, it transfers the pages across the nodes through RDMA if necessary.

Managing pages across the nodes. The NVIDIA driver manages the pages in the CUDA UM space by grouping them into multiple UM blocks. Maximum 512 contiguous pages are grouped into a UM block. The maximum size of a UM block is 2MB because the page size is 4KB. Each UM block object contains all

Table 6.1: Structure of Block Descriptor.

Type	Entry	Description
uint512_t[NUM_NODES]	read_masks	Read protection status of pages
uint512_t[NUM_NODES]	write_masks	Write protection status of pages
uint32_t	flags	Bit 0: Lock flag Bit 1: read-only flag Bit 2-31: Reserved
struct list_head	lock_waitlist	List of nodes waiting for lock

information of pages in the UM block, such as which processor has the pages and whether the pages are mapped with read protection or write protection. To record the location and status of UM pages across the nodes, the SnuRHAC driver manages an additional data structure called *block descriptor* for each UM block. A block descriptor manages the pages' location and protection status across the nodes for the corresponding UM block. Table 6.1 shows the structure of the block descriptor.

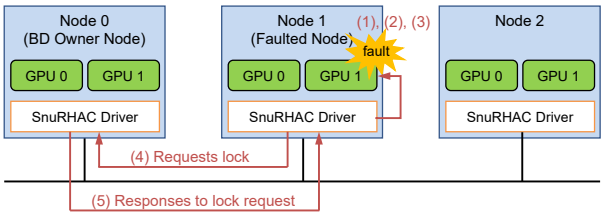
The first entry of the block descriptor is an array of 512-bit bitmaps. The size of the array is equal to the number of nodes in the cluster. The Mth bit of the Nth element of the array represents whether the Mth page in the UM block is mapped to processors in the Nth node with read protection. The second entry is the same as the first entry except that it represents write protection status. When any processor in the Nth node of the cluster has the Mth page with write protection, the Mth bit of the Nth element of the array is set. The third entry of the block descriptor represents the flag status of the UM block object. The first bit is set when a node is modifying the block descriptor. The second bit is set when the SnuRHAC runtime sets a read-only flag for the corresponding UM block. The last entry is used to list the ID of nodes waiting to modify the

block descriptor.

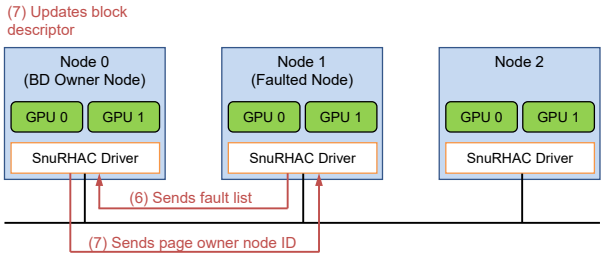
Distributed block descriptors. The SnuRHAC runtime allocates a UM space in every node when the CUDA program requests device memory allocation. It passes the information of the allocated UM space to the SnuRHAC driver. Then, SnuRHAC driver creates a block descriptor for each UM block in the UM space. To minimize interconnection network contention, the SnuRHAC driver distributes block descriptors across the nodes in the cluster. It distributes every 32 block descriptors in a round-robin manner to each node. For example, node 0 manages block descriptors for UM blocks 0 to 31. Node 1 manages block descriptors for UM blocks 32 to 63, and so on.

Handling page faults. The SnuRHAC driver's page fault handler hooks the NVIDIA driver's page fault handler. That is, the SnuRHAC's handler handles the page fault interrupt signal. However, it calls some functions implemented in the NVIDIA driver's page fault handler, such as functions for fault buffer accesses and sending replay signals to the GPU. The NVIDIA GPU has a hardware fault buffer that is a circular queue. It accumulates the information for faulted accesses. Suppose that a GPU page fault has occurred in a node P. The page fault handler in the SnuRHAC driver of P services the fault in the following manner (please, see Figure 6.3):

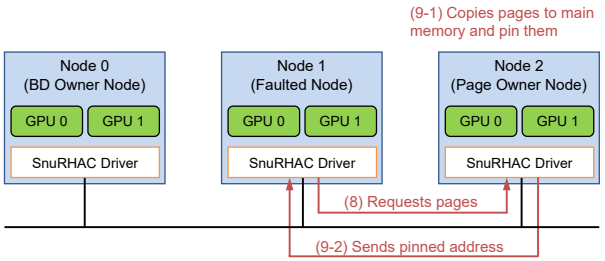
1. The SnuRHAC page fault handler fetches the page addresses and access type of the faulted accesses from the fault buffer in the GPU. (Figure 6.3(a))
2. The handler sorts faulted accesses by page addresses and groups them by UM blocks.
3. For each faulted UM block UMB, The handler finds the node that owns



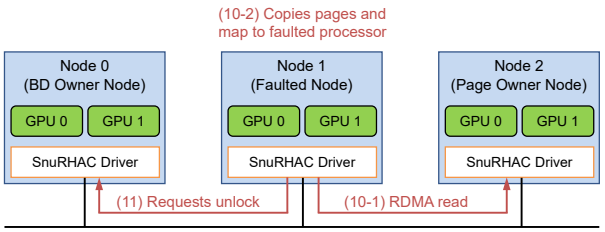
(a) Steps (1) to (5) of GPU page-fault handling.



(b) Steps (6) to (7) of GPU fault handling.



(c) Steps (8) to (9) of GPU page-fault handling.



(d) Steps (10) to (11) of GPU page-fault handling.

Figure 6.3: Page-fault handling.

the corresponding block descriptor. We call this node BD owner node.

4. It sends a lock request message for UMB to the BD owner node and waits for the response.

5. The BD owner node sends a response to P if no other node has locked UMB. This checking is done by looking at the lock flag of the block descriptor of UMB. If another node has already locked the block descriptor, the BD owner node adds P to the waitlist in the block descriptor.

6. If P receives the response from the BD owner node, P sends the list of faulted accesses obtained from step 1 to the BD owner node. (Figure 6.3(b))

7. When the BD owner node receives the faulted access list, it looks up the list and finds the nodes where the faulted pages are resident. We call these nodes page owner nodes. Then, it sends the IDs of the discovered nodes to P. Simultaneously, the BD owner node updates the read_masks and write_masks of the block descriptor based on the information from the faulted access list.

8. P receives the list of nodes that have faulted pages. It requests pages to the page owner nodes. (Figure 6.3(c))

9. The page owner nodes receive the page requests from P. They copy the faulted pages to their main memory. After copying, they pin the memory space and send the pinned pages to P. They update the page mappings if necessary.

10. P reads the faulted pages through RDMA and copies the pages to the UM space of the faulted GPU and maps the pages to the faulted GPU. (Figure 6.3(d))

11. When P finishes copying and mapping all the faulted pages, it sends an unlock request of UMB to the BD owner node.
12. The SnuRHAC driver in P sends a replay signal to the faulted GPU so that the GPU can replay the faulted page accesses.

A similar fault handling process is followed for the CPU page fault. When a page fault occurs in the CPU, it raises a page fault interrupt signal and the SnuRHAC page fault handler services the fault.

SnuRHAC pipelines the page fault handling mechanism so that multiple faulted UM blocks can be serviced simultaneously. It increases the throughput of page fault handling.

Managing page mappings. Page mapping determines which page is mapped to which processors with read or write protection. The SnuRHAC driver in each node maintains this information. For intra-node page mapping, the SnuRHAC driver follows the page mapping rule of the NVIDIA driver. For inter-node page mapping, the SnuRHAC driver allows multiple nodes to have the same page simultaneously with read protection. However, only one node can have the page with write protection. When a read page fault occurs from a processor (*e.g.*, a CPU or GPU) in the Nth node for the Mth page in a UM block:

- The SnuRHAC driver sets the Mth bit of the Nth element in the read_masks of the block descriptor.
- The SnuRHAC driver maps the faulted page to the faulted processor with read protection.

Even though there exists a processor to which the faulted page is mapped with write protection, mapping the faulted page to the faulted processor with read protection does not cause any data consistency problem during the kernel

execution. According to the CUDA documentation, updates to global memory by a thread block does not need to be visible to other thread blocks in the same kernel grid during the kernel execution[91]. Since different GPUs always execute different thread blocks for the same kernel, it is safe. The SnuRHAC driver synchronizes the page mappings after the kernel execution has finished in all GPUs in the cluster to ensure data consistency.

When a write page fault occurs from a processor in the Nth node for the Mth page in a UM block:

- The SnuRHAC driver sets the Mth bit of the Nth element in the read_masks and the write_masks of the block descriptor.
- The SnuRHAC driver clears the Mth bit of all the other elements in the write_masks.
- The SnuRHAC driver degrades the write protection of the faulted page to read protection in the processor to which the faulted page was previously mapped.
- The SnuRHAC driver maps the faulted page to the faulted processor with write protection.

When a write page fault occurs, the SnuRHAC driver does not unmap the faulted page to the processor to which the faulted page was previously mapped with write protection or read protection. Instead, it just degrades the write protection to read protection. This implies that a processor may have an outdated page. However, similar to the read page fault, it does not cause any data consistency problem during the kernel execution.

Synchronization across the nodes. When all GPUs in the cluster finish a kernel execution, the SnuRHAC driver performs synchronization for the kernel

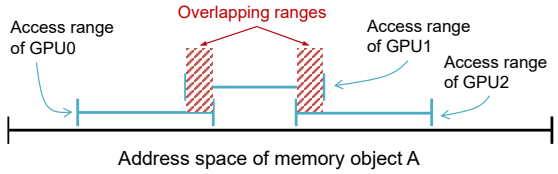


Figure 6.4: Checking overlapping access ranges.

to maintain data consistency. At this point, a page may be mapped to many processors (*e.g.*, CPUs or GPUs) in the cluster. However, only one GPU can have the page with write protection, and the others have it with read protection if there is any. If a page is mapped to any processor with write protection, the SnuRHAC driver unmaps the page from the processors who have the page with read protection. Also, it clears the bit of the page in the read_masks in the block descriptor. Thus, the processors who have the page with read protection will fetch the most up-to-data page later when it executes a kernel that accesses the page.

6.4 Additional Optimizations

Handling read-only pages. Basically, a page cannot be mapped to multiple processors (*e.g.*, GPUs) at the same time. However, SnuRHAC allows some pages in the UM space to be mapped to multiple processors. When a CUDA program launches a kernel, if all memory operations for a memory object are read-only, SnuRHAC sets the read-only flag for the UM blocks that belong to the memory object. Then the pages in the UM blocks can be mapped to multiple processors. It significantly reduces frequent page faults due to page sharing.

Avoiding multiple GPUs. Some applications are not suitable for multiple

GPUs. An example is that uses atomic operations. The SnuRHAC runtime inspects kernel code if there is an atomic operation. If so, it executes the kernel using only a single GPU. Another example is an application that has many pages shared between GPUs. When a read-only flag is set for the memory object, it does not cause any performance degradation because its pages do not move once they are copied to the target GPUs. However, the problem occurs when write sharing occurs. When writes to the shared pages are frequent, it introduces significant performance degradation because write-shared pages are migrated frequently. To avoid this, SnuRHAC runtime calculates the access range of write operations from each GPU. Then, it calculates the amount of overlapping in the access ranges. If the ratio of the overlapping amount to the total amount of access ranges exceeds a predefined threshold, SnuRHAC executes the CUDA kernel using only a single GPU. Figure 6.4 shows how to check the overlapping access ranges.

6.5 Prefetching

Since processing page faults is costly and it is desirable to reduce page faults as much as possible. As a solution to this problem, SnuRHAC exploits page prefetching techniques. In this section, we describe the two prefetching techniques used in SnuRHAC: *static* and *dynamic*.

6.5.1 Static Prefetching

MAPA. To prefetch pages to GPUs when a kernel executes, SnuRHAC needs to know the kernel's memory access patterns. SnuRHAC's static prefetching technique is based on MAPA[30]. MAPA is a static memory access pattern analyzer for OpenCL. SnuRHAC extends it to CUDA kernels. MAPA relies on

both a source-level compiler analysis technique derived from traditional symbolic analyses and run-time information (*e.g.*, kernel arguments). It extracts memory access patterns of a kernel in symbolic expressions and classifies the access patterns into ten categories: constant, affine, indirect, complex, etc. Each variable in the symbolic expressions represents a determined value at run time, either before or during the kernel execution.

For example, for memory access patterns that fall in the affine category, MAPA provides affine function $f(x_1, x_2, ..., x_k) = a_1x_1+a_2x_2+...+a_kx_k+c$ that represents memory access locations within a memory object, where $a_i \neq 0$ and $0 \leq x_i < N_i$ for all $1 \leq i \leq k$. N_i indicates the constant upper bound of variable x_i. For example, if x_i indicates the thread index variable in x-dimension, N_i becomes the size of a thread block in x-dimension. If x_i indicates the induction variable `i` in `for(i=0;i<1000;i++)`, then N_i becomes 1000.

SnuRHAC exploits memory access patterns that fall in the affine and constant categories from MAPA. If possible, MAPA outputs every memory access expressions in the affine function of block index variables ($blockIdx.x, blockIdx.y, blockIdx.z$), thread index variables ($threadIdx.x, threadIdx.y, threadIdx.z$) and induction variables (if the memory operations is contained in a loop).

Figure 6.5 shows how SnuRHAC's static prefetching works. When a CUDA program launches a kernel ((1) in Figure 6.5), the SnuRHAC runtime in the host node loads memory access patterns from MAPA ((2) in Figure 6.5). Then, it partitions the workload and distributes the workload with the access patterns ((3) in Figure 6.5). The SnuRHAC runtime in each node calculates each thread block's memory access ranges using the symbolic access patterns and run-time information.

The most simple way to calculate the memory access range is calculating the lower bound and upper bound of addresses accessed by each thread in the

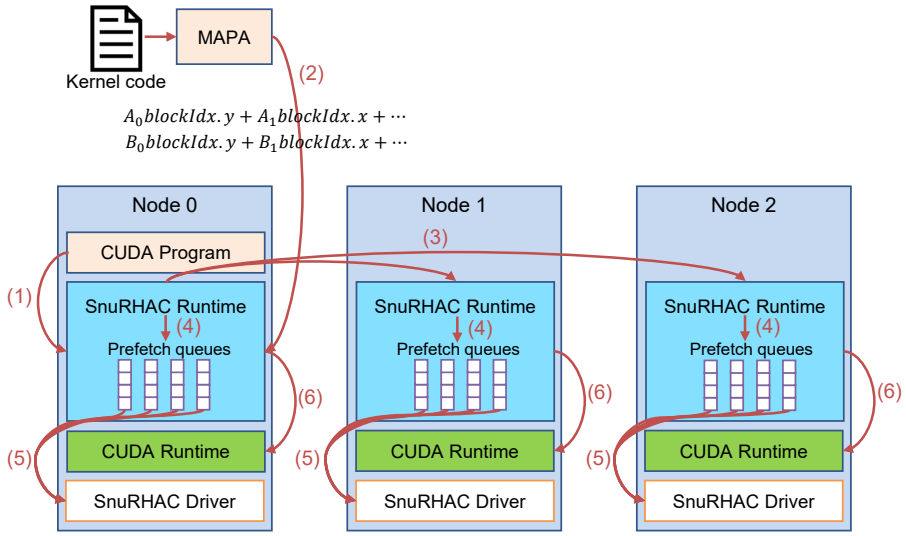


Figure 6.5: How static prefetching works.

thread block. However, calculating the addresses accessed by all threads can be very time-consuming. Instead, SnuRHAC selects some representative threads in the thread block and calculates the lower bound and the upper bound of addresses accessed by the thread block. For a one-dimensional block of size (B_x), SnuRHAC selects thread indices (0) and $(B_x - 1)$ as representative threads. For a two-dimensional block of size (B_x, B_y), SnuRHAC selects thread indices $(0, 0)$, $(0, B_y - 1)$, $(B_x - 1, 0)$, and $(B_x - 1, B_y - 1)$ as representative threads. For a three-dimensional block of size (B_x, B_y, B_z), SnuRHAC selects thread indices $(0, 0, 0)$, $(B_x - 1, 0, 0)$, $(0, B_y - 1, 0)$, $(B_x - 1, B_y - 1, 0)$, $(0, 0, B_z - 1)$, $(B_x - 1, 0, B_z - 1)$, $(0, B_y - 1, B_z - 1)$, and $(B_x - 1, B_y - 1, B_z - 1)$ as representative threads. After obtaining the access ranges for each GPU, the SnuRHAC runtime enqueues prefetch commands to appropriate prefetch queues ((4) in Figure 6.5).

The SnuRHAC runtime maintains a prefetch queue for each GPU. Also, there is a CPU thread called *prefetch scheduler* for each GPU in the SnuRHAC

runtime. The prefetch scheduler thread dequeues the prefetch commands and issues them to the SnuRHAC driver ((5) in Figure 6.5). While the SnuRHAC driver in a node prefetches the requested pages, the SnuRHAC runtime in the same node executes the partitioned kernel ((6) in Figure 6.5). The way the SnuRHAC driver prefetches pages is very similar to how it handles page faults because the SnuRHAC driver treats a prefetch command as page faults.

6.5.2 Dynamic Prefetching

It is impossible to analyze all different types of memory access patterns statically. One example of this is the indirect memory access pattern. To complement static prefetching, SnuRHAC uses dynamic prefetching techniques: adjacent page prefetching and correlation prefetching[89].

Adjacent page prefetching. When the read-only flag is set for a UM block, the SnuRHAC driver uses adjacent page prefetching. It is similar to the adjacent cache line prefetching in a CPU. When a page fault occurs in the GPU, the SnuRHAC driver prefetches all pages in the faulted page's UM block.

Correlation prefetching. When the read-only flag is not set for a UM block, the SnuRHAC driver uses correlation prefetching described in Section 5.2. SnuRHAC applies the correlation prefetching technique to the page level and implements a pair-based scheme[89]. The SnuRHAC driver in a node has a correlation table for each GPU in the node and keeps track of faulted pages from each GPU with the correlation table. When a page fault occurs in the GPU, all pages that are correlated to the faulted page (according to the correlation table) are prefetched.

Chapter 7

Evaluation

7.1 Framework for Maximizing the Performance of Traditional CUDA Program

In this section, we evaluate HUM with various GPU applications and analyze the results. We compare the performance of HUM with that of manual optimizations.

Table 7.1: System configuration.

CPU	2 × Intel 2.10 Ghz 16-core Xeon Gold 6130
Main memory	256GB DDR4
OS	CentOS 7.6.1810 (kernel 3.10.0-957)
GPU	4 × NVIDIA Tesla V100 PCIe (16GB device memory for a GPU)
GPU driver	NVIDIA display driver 410.48
CUDA version	10.0

Suite	No.	Name	Sync or async	CPU/H2D overlap	CPU/H2D overlap (HUM)
Parboil	1	bfs	S	N	Y
	2	cutcp	S	N	Y
	3	histo	S	N	Y
	4	lbm	S	N	N
	5	mri-gridding	S	N	Y
	6	mri-q	S	N	Y
	7	sad	S	N	Y
	8	sgemm	S	N	Y
	9	spmv	S	N	Y
	10	stencil	S	N	Y
	11	tpacf	S	N	N
	12	backprop	S	N	Y
	13	b+tree	S	N	Y
	14	cfd	S	N	N
	15	dwt2d	S	N	Y
	16	gaussian	S	N	Y
	17	heartwall	S	N	Y

Suite	No.	Name	Sync or async	CPU/H2D overlap	CPU/H2D overlap (HUM)
Rodinia	18	hotspot	S	N	Y
	19	hotspot3D	S	N	Y
	20	huffman	S and A	N	Y
	21	hybridsort	S	N	Y
	22	kmeans	S	N	Y
	23	lavaMD	S	N	Y
	24	leukocyte	S	N	N
	25	lud	S	N	Y
	26	mummergpu	S	N	Y
	27	myocyte	S	N	Y
	28	nn	S	N	Y
	29	nw	S	N	Y
	30	particlefilter	S	N	Y
	31	pathfinder	S	N	Y
	32	srad	S	N	Y
	33	streamcluster	S	N	Y
	34	alignedTypes	S	N	Y

Suite	No.	Name	Sync or async	CPU/H2D overlap	CPU/H2D overlap (HUM)
CUDA Code Samples	35	BlackScholes	S	N	Y
	36	eigenvalues	S	N	Y
	37	fastWalshTransform	S	N	Y
	38	matrixMul	S	N	Y
	39	MC_SingleAsianOptionP	S	N	Y
	40	mergeSort	S	N	Y
	41	MonteCarloMultiGPU	A	Y	Y
	42	nbody	S	N	Y
	43	reduction	S	N	Y
	44	scalarProd	S	N	Y
	45	scan	S	N	Y
	46	SobolQRNG	S	N	Y
	47	sortingNetworks	S	N	Y
	48	threadFenceReduction	S	N	Y
	49	transpose	S	N	Y
	50	vectorAdd	S	N	Y
	51	warpAggregatedAtomicsCG	S	N	Y

Figure 7.1: Characteristics of applications.

7.1.1 Methodology

System configuration. We use NVIDIA Tesla V100 (Volta architecture[92]) GPU for our experiment. Detailed system configuration is summarized in Table 7.9.

Benchmark applications. We use 51 applications from various sources: 11 applications from Parboil[31], 22 applications from Rodinia[32], and 18 applications from CUDA Code Samples[33]. While we use all the applications from Parboil and Rodinia, we choose only 18 out of 170 applications in CUDA Code Samples. We exclude 152 applications in CUDA Code Samples because of the following reasons:

- They use CUDA graphics or driver API,
- They have neither CUDA kernel execution nor H2Dmemcpy,
- They use additional CUDA libraries (cuBLAS, cuFFT, cuSPARSE, cuSOLVER, nvGRAPH),
- They appear in Parboil or Rodinia, or

- Their kernel execution times are too small (less than 1ms) to see the effect of overlapping H2Dmemcpy and CUDA kernel computation.

We exclude applications that use additional CUDA libraries because the HUM runtime has to provide wrapper functions for all library functions used in the applications. These applications mainly focus on showing the functionality of the libraries. While there is no technical difficulty to implement the wrapper functions, we decide not to include those applications because it is too time-consuming.

Figure 7.1 shows the characteristics of applications in each benchmark suites. The column Sync or async shows the type of H2Dmemcpy commands each application uses. The column CPU/H2D overlap shows if the application is designed to overlap CPU computation and H2Dmemcpy. The column CPU/H2D overlap (HUM) shows if HUM can overlap the CPU computation and the H2Dmemcpy.

Most of the applications use synchronous H2Dmemcpy and hence, they are unable to overlap CPU computation and H2Dmemcpy when running under normal CUDA. On the other hand, HUM can overlap the CPU computation and the H2Dmemcpy in most of the cases except some applications that modify the contents of the source host memory object of the H2Dmemcpy or frees it after the H2Dmemcpy (lbm and tpacf in Parboil, cfd and leukocyte in Rodinia).

We use the largest dataset that fits in the GPU memory for each application, hence, most of the datasets used for the experiment are hundreds of megabytes to a few gigabytes. As the goal of HUM is performance improvement without any code modification, no source code of the applications is modified.

7.1.2 Results

Speedup on V100. Figure 7.2 shows the speedup of each application with various optimization schemes on a single V100 GPU. The speedup is obtained

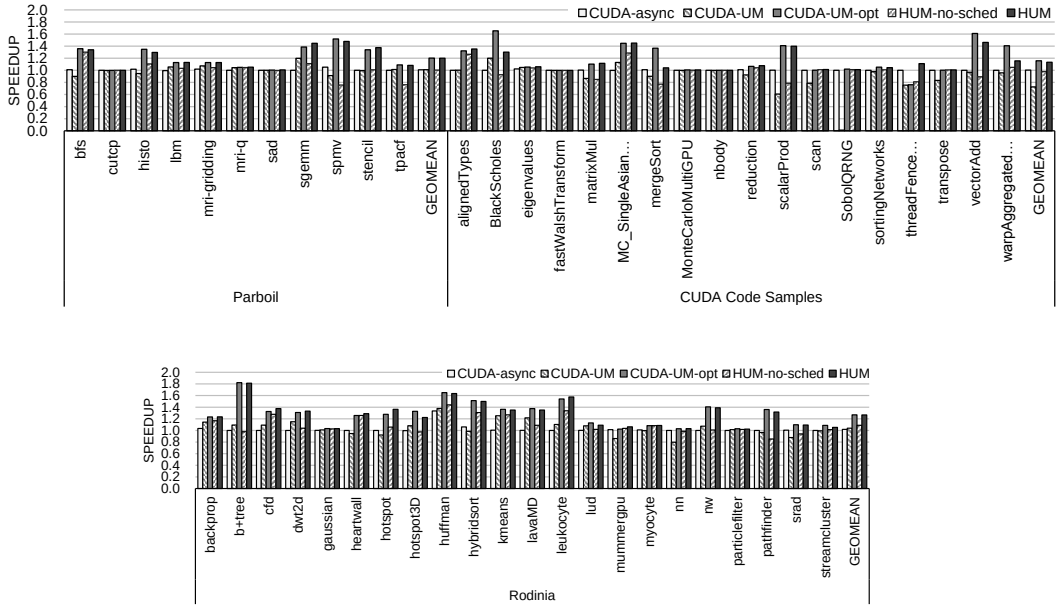


Figure 7.2: Speedup of each application with a single V100 GPU.

over running the original version of each application (this setup is called CUDA hereafter). The optimization schemes are described as follows:

- CUDA-async is a manually optimized version where synchronous memory copy functions in the original application is transformed to corresponding asynchronous ones when the transformation is safe.
- CUDA-UM is a naive UM implementation of each application. We change all `cudaMalloc()` functions to `cudaMallocManaged()`. Then, we remove all CUDA memory copy functions, such as `cudaMemcpy()` and `cudaMemcpyAsync()`, because data will be automatically transferred between the host and the device by CUDA UM.
- CUDA-UM-opt is a manually optimized version of CUDA-UM using user hints (*e.g.*, `cudaMemPrefetchAsync()` and `cudaMemAdvise()`).
  We add `cudaMemPrefetchAsync()` as early as possible before the CUDA

kernel launch so that memory copy and kernel computation can be overlapped. `cudaMemPrefetchAsync()` is also used to map blank pages to the GPU if the pages are first accessed for write by the GPU. This prevents excessive write page faults in the GPU side. We add `cudaMemAdvise()` to avoid page migration if the pages are read by both the CPU and the GPU without any write (*i.e.*, read-only accesses).

- HUM-no-sched runs the applications under HUM without any H2Dmemcpy command scheduling described in Section 4.7.
- HUM runs the applications under HUM with all the HUM techniques described in Chapter 4.

The number of memory-copy threads mentioned in Section 4.7 is set to eight in both HUM-no-sched and HUM. For execution time measurement, we use the timing routines that are already placed in the original benchmark source code. These routines exclude data initialization and result verification from the execution time measurement. In addition, we exclude file I/O from the time measurement to clearly see the effect of overlapping.

For all applications, CUDA-UM-opt and HUM outperform CUDA, CUDA-async, and CUDA-UM. Some applications show marginal speedup under HUM and CUDA-UM-opt. This happens when the H2Dmemcpy time takes a very little portion of the total execution time. For example, the host computation time dominates the total execution time of cutcp in Parboil. The kernel execution time dominates the total execution time of mri-q in Parboil, gaussian and particlefilter in Rodinia, MonteCarloMultiGPU, nbody, scan, and transpose in CUDA Code Samples. The D2Hmemcpy time dominates the total execution time of sad in Parboil, SobolQRNG in CUDA Code Samples.

On the other hand, some applications show very good speedup under HUM

and CUDA-UM-opt. The applications sgemm and spmv in Parboil, b+tree, hybridsort, and leukocyte in Rodinia have enough kernel computation time to hide the H2Dmemcpy time. The application huffman in Rodinia mainly benefits from overlapping the H2Dmemcpy and the host computation.

On average, CUDA-UM-opt achieves the speedup of 1.22x for all applications and HUM achieves the speedup of 1.21x (1.20x for Parboil, 1.26x for Rodinia, and 1.13x for CUDA Code Samples). While CUDA-UM-opt does not use any H2Dmemcpy command scheduling like HUM, it shows comparable performance to HUM. This is because HUM spends more time on memory copying than CUDA-UM-opt.

CUDA-UM-opt is much better than HUM in BlackScholes, vectorAdd, and warpAggergatedgAtomicsCG in CUDA Code Samples. This happens when the kernel computation time is much shorter than H2Dmemcpy time. Since the memory transfer time dominates the entire application execution time, CUDA-UM-opt shows better performance.

This is due to the prefetching heuristics used in the NVIDIA driver for page migration. When a GPU page fault occurs, the NVIDIA driver actively prefetches some pages around the faulted page from the host UM space to the GPU UM space according to the prefetching heuristics (note that the heuristics are not publicly known).

CUDA-async is a little bit better than CUDA for Parboil on average, but there is no difference between CUDA-async and CUDA for Rodinia and CUDA Code Samples on average. This is because few applications in Rodinia and CUDA Code Samples have some host computation to hide between the H2Dmemcpy command and the kernel launch command.

CUDA-UM is a little bit better, on average, than CUDA-async for Parboil and Rodinia because of the prefetching heuristics used in the NVIDIA

driver for the Unified Memory. CUDA-UM is much worse than CUDA-async for CUDA Code Samples on average because of SobolQRNG. In SobolQRNG, CUDA-UM is 88 times slower than CUDA-async. The 4GB write-only data accessed by the kernel in SobolQRNG incur a lot of page faults in the GPU side for CUDA-UM. This does not happen for CUDA-UM-opt because to avoid the write page faults, CUDA-UM-opt maps the data pages to the GPU using `cudaMemPrefetchAsync()` before the kernel execution is initiated.

Effect of H2Dmemcpy command scheduling. HUM-no-sched is slower than HUM consistently. Even HUM-no-sched is slower than CUDA for some applications. One such a case is when the memory-copy time dominates the execution time. When the kernel computation time is not large enough, overlapping the H2Dmemcpy and the kernel computation cannot fully amortize the slowdown in H2Dmemcpy due to copying the memory object twice from the source host memory space to the host UM space, and then from the host UM space to the device UM space. tpacf in Parboil, nn, pathfinder, and srad in Rodinia, BlackScholes, mergeSort, scalarProd, and threadFenceReduction in CUDA Code Samples fall in this category.

Another case is when the CUDA kernel launch is not scheduled as early as possible. spmv in Parboil, b+tree in Rodinia, matrixMul and vectorAdd in CUDA Code Samples fall in this category. For example, as mentioned in Section 4.7, in vectorAdd, there are two memory objects to transfer from the host to the device (vector A and vector B). Without the memory-copy command scheduling, the kernel execution cannot be scheduled until the entire vector A has been copied to the device.

The number of memory-copy threads. As mentioned in Section 4.6, HUM uses multiple threads to copy the source host memory object to the host UM

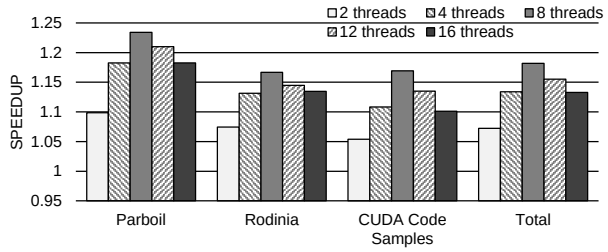


Figure 7.3: Average speedup obtained by varying the number of memory-copy threads.

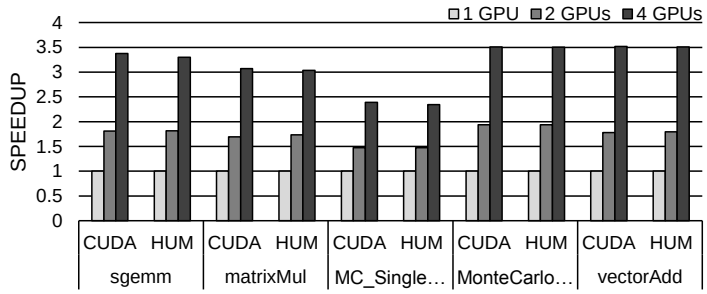


Figure 7.4: Speedup on multiple V100 GPUs.

space to execute a H2Dmemcpy command. To find the optimal number of threads, we vary the number of memory-copy threads from 1 to 16 and measure the overall performance. Here, we divide the source host memory object in N chunks when there are N memory-copy threads. Each thread takes care of a chunk of the source host memory object and copies the chunk simultaneously. Figure 7.3 shows the average speedup obtained over one thread for each benchmark suite. We see that, on average, eight is the optimal number of memory-copy threads.

Multi-GPU environments. To show that HUM works well with multi-GPU environments, we choose the applications whose speedup under HUM with a single GPU is greater than 1.10 and whose workload can be easily distributed

across multiple GPUs. These applications include sgemm in Parboil, and matrixMul, MC_SingleAsianOptionP, and vectorAdd in CUDA Code Samples. We implement the multi-GPU version of them. In addition, we choose MonteCarloMultiGPU in CUDA Code Samples because it is originally designed to support multiple GPUs.

Figure 7.4 shows the speedup obtained by varying the number of GPUs for these applications. We do not vary the workload for multiple GPUs, hence Figure 7.4 shows the result of strong scaling for both CUDA and HUM. The speedup is obtained over the case of a single GPU for each of CUDA and HUM. The result indicates that HUM achieves scalable performance in the multi-GPU environment. The major reason for this strong scaling is that page faults occurred in different GPUs are handled by different host threads.

7.2 Framework for Running Large-scale DNNs on a Single GPU

In this section, we compare the performance of DeepUM with previous approaches: LMS[25], vDNN[23], AutoTM[27], SwapAdvisor[28], Capuchin[26], and Sentinel[29].

7.2.1 Methodology

LMS (Large Model Support)[25] is an open-source project developed by IBM. It supports the PyTorch framework and automatic GPU memory swapping. We directly compare the performance of DeepUM and LMS by actually executing LMS. Since the other five approaches are based on TensorFlow[93] or other frameworks and are mostly closed-source, we indirectly compare their performance with DeepUM. Ren et al.[29] implement these approaches and measure

the speedup of their training throughput over NVIDIA UM. Thus, we take the number from Ren et al. and compare them with the speedup of DeepUM over UM. We also obtain these approaches' maximum available batch sizes from the same paper. We use the same models, datasets, and GPU for a fair comparison.

System configuration. We use NVIDIA Tesla V100 GPUs[92] with different device memory sizes. Table 7.2 shows the detailed system configuration.

Table 7.2: System configuration for evaluation of DeepUM.

CPU	2 × AMD 1.5Ghz 32-core EPYC 7452
Main memory	512GB DDR4 for each node
OS	Ubuntu 18.04.4 LTS (kernel 4.15.0-72)
GPU	NVIDIA Tesla V100 PCIe 32GB NVIDIA Tesla V100 PCIe 16GB
GPU driver	NVIDIA display driver 460.32
CUDA version	11.2

Table 7.3: DNN models and dataset used for evaluation.

Model	Source	Dataset
GPT-2 XL[94]	Hugging Face[36]	Wikitext
GPT-2 L[94]	Hugging Face[36]	Wikitext
BERT Large[95]	Hugging Face[36]	Wikitext, GLUE CoLA
BERT Base[95]	Hugging Face[36]	Wikitext
DLRM[96]	MLPerf[34]	Cirteo Kaggle
ResNet200[97]	PyTorch examples[35]	ImageNet, CIFAR-10
ResNet152[97]	PyTorch examples[35]	ImageNet
DCGAN[98]	PyTorch examples[35]	celebA
MobileNet[99]	PyTorch examples[35]	CIFAR-100

DNN models and datasets. We use six DNN models from various sources: Hugging Face[36], MLPerf[34], and PyTorch examples[35]. BERT and ResNet in Hugging Face and PyTorch examples are also included in MLPerf. The list of models and datasets are summarized in Table 7.3.

7.2.2 Comparison with Naïve UM and IBM LMS

Speedup. Figure 7.5 shows the speedup of training throughput for each DNN model with various batch sizes. We use a single V100 32GB GPU. The speedup is obtained over UM that runs each DNN model using NVIDIA UM without prefetching. LMS shows the performance of the original LMS, and LMS-mod shows the performance of LMS that is modified to periodically free cached PT blocks in the PyTorch memory pool. By cleaning up cached memory objects periodically, we can reduce the occurrence of out-of-memory (OOM) errors caused by memory fragmentation. DeepUM uses UM block correlation tables, each containing 2048 rows, two-way associative, and four successors. Also, we select the batch sizes close to the maximum batch size of LMS to enable the maximum GPU memory oversubscription. Some numbers of LMS and LMS-mod are missing because they fail to run the model in some batch sizes because of the OOM error.

Like other memory prefetching strategies, the performance depends on the application's memory access patterns and the ratio between the memory transfer time and the computation time. For example, DLRM shows almost no speedup over UM for both LMS and DeepUM. DLRM is a recommendation model introduced by Facebook, and its input data consist of users' preferences, buy lists, etc. The model looks up the embedding table for each input data item and transforms the input data item with an appropriate embedding vector. In DLRM, most of the memory space is used to store embedding tables. In

addition, its memory access pattern is irregular because the embedding table lookups highly depend on the input data. This is why prefetching strategies of both LMS and DeepUM do not work well. On the other hand, ResNet consists of multiple building blocks called residual blocks. Once a memory object is prefetched for a residual block, the computation time dominates the processing of a residual block.

Note that LMS moves data at the whole tensor level while DeepUM moves data at the UM block level. While LMS is faster than LMS-mod on average, it fails to run with batch size that LMS-mod can run. DeepUM can run a larger batch size than LMS and LMS-mod because of the virtual memory system supported by DeepUM runtime.

On average, DeepUM shows the best performance. DeepUM is 3.06× faster than UM and 1.11× faster than LMS.

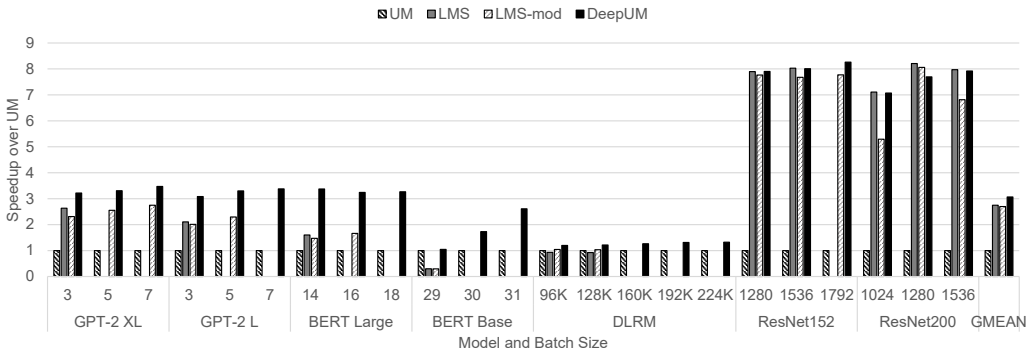


Figure 7.5: The speedup of IBM LMS and DeepUM over a naïve UM implementation.

Maximum possible batch sizes. Table 7.4 shows the maximum possible batch sizes of IBM LMS and DeepUM. DeepUM can run the models with the batch size that requires the peak memory usage to be almost the same as the

Table 7.4: Maximum possible batch sizes.

Model	Dataset	LMS	DeepUM
GPT-2 XL	Wikitext	3	16
GPT-2 L	Wikitext	3	24
BERT Large	Wikitext	14	192
BERT Base	Wikitext	29	256
DLRM	Criteo Kaggle	128k	512k
ResNet200	ImageNet	1536	2304
ResNet152	ImageNet	1536	1792

total CPU memory size. The numbers in the table indicate that exploiting UM in DeepUM suffers fewer memory fragmentation issues and has a higher chance of running large DNN models without any memory problems.

Correlation table size. Table 7.5 shows the size of memory space used for storing correlation tables. Since DeepUM dynamically allocates a UM block correlation table when it finds a kernel with a new execution ID, the memory size for storing correlation tables differs for each DNN model and batch size. Note that the correlation tables are stored on the CPU side.

Number of page faults. Original correlation prefetching is one method of cache-line prefetching for CPUs. In this case, the cache hit rate is typically used to show the effectiveness of a prefetching technique. However, the GPU we used does not have such a performance counter, and it is very hard to know whether the GPU has accessed the prefetched pages or not at a certain point. Even though there is an instrumentation tool, such as NVBit[100], it causes lots of instrumentation overhead, resulting in the prefetch timing difference.

Table 7.5: Correlation table size.

Model	Batch size	Table size (MB)
GPT-2 XL	3	308
	5	344
	7	348
GPT-2 L	3	169
	5	213
	7	232
BERT Large	3	78
	5	75
	7	74
BERT Base	3	19
	5	27
	7	33
DLRM	96k	13
	128k	19
	160k	30
	192k	31
	224k	35
ResNet152	1280	115
	1536	128
	1792	130
ResNet200	1024	144
	1280	151
	1536	169

Table 7.6: Average number of page faults per training iteration.

Model	Batch size	Fault count of UM	Fault count of DeepUM	Ratio
GPT-2 XL	3	7437122	687	< 0.1%
	5	12395173	7612	< 0.1%
	7	17210705	2549	< 0.1%
GPT-2 L	3	2948920	235	< 0.1%
	5	6055304	476	< 0.1%
	7	8974631	884	< 0.1%
BERT Large	3	1171717	2913	0.2%
	5	1777710	84	< 0.1%
	7	1834746	1355	< 0.1%
BERT Base	3	88459	1595	1.8%
	5	349106	4536	1.3%
	7	1077223	5531	0.5%
DLRM	96k	1263865	3706	0.2%
	128k	1712886	6912	0.4%
	160k	2583610	22624	0.8%
	192k	3471958	32139	0.9%
	224k	4278593	38437	0.9%
ResNet152	1280	121380940	34323	< 0.1%
	1536	144893625	72598	< 0.1%
	1792	182230994	144455	< 0.1%
ResNet200	1024	126734315	107093	< 0.1%
	1280	173517031	68039	< 0.1%
	1536	207933814	118472	< 0.1%

Therefore, we use the number of page faults to measure the accuracy of DeepUM prefetching technique.

Table 7.6 shows the average number of page faults per training iteration for each model and different batch sizes. The result indicates that DeepUM prefetches pages quite accurately and can significantly reduce page faults.

Effects of prefetching and optimizations. Figure 7.6 shows the effects of prefetching and optimizations. Prefetching shows the effect of the correlation prefetching only. Prefetching+Preeviction shows the effect of correlation prefetching and page pre-eviction described in Section 5.3.1. Finally, Prefetching+Preeviction+Invalidate shows the effect of all the optimization techniques mentioned in Section 5.3.1 and Section 5.3.2. Prefetching, Prefetching+Preeviction, and Prefetching+Preeviction+Invalidate reduce 45.6%, 63.7%, and 66.7% of the execution time on average.

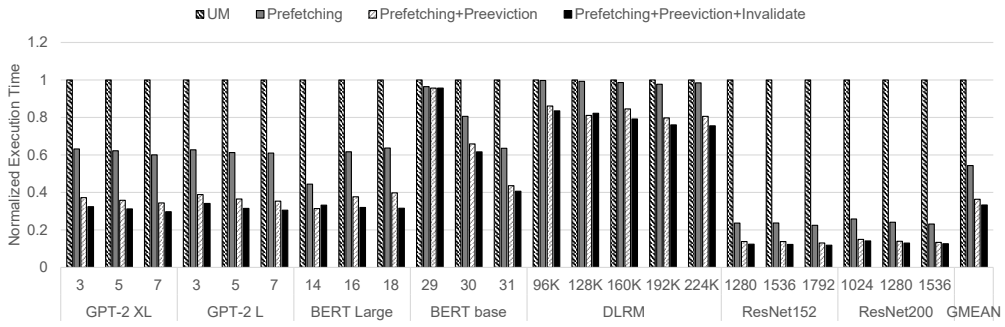


Figure 7.6: Effect of prefetching and optimizations.

As mentioned before, DLRM gets no benefit from prefetching due to its irregular memory access patterns. When the batch size is 29 in BERT-base, the effect of prefetching is very small. This is because a tiny portion of the total memory usage is oversubscribed (approximately 3% of the total memory usage).

Overall, the result indicates that the prefetching and optimization techniques in DeepUM are very effective.

7.2.3 Parameters of the UM Block Correlation Table

There are several configuration parameters of the UM block correlation table: the number of immediate successor blocks (NumSuccs), the number of rows in the table (NumRows), and the associativity of the table (Assoc). To find the optimal configuration, we perform a sensitivity analysis.

Table 7.7: Effect of parameters of the UM block correlation table.

Name	Assoc	NumSuccs	NumRows
Config0	2	4	128
Config1	2	8	128
Config2	4	4	128
Config3	2	4	512
Config4	2	8	512
Config5	4	4	512
Config6	2	4	1024
Config7	2	8	1024
Config8	4	4	1024
Config9	2	4	2048
Config10	2	8	2048
Config11	4	4	2048
Config12	2	4	4096

Table 7.7 shows different configurations for the UM block correlation table, and Figure 7.7 shows the speedups of the different configurations over Con-

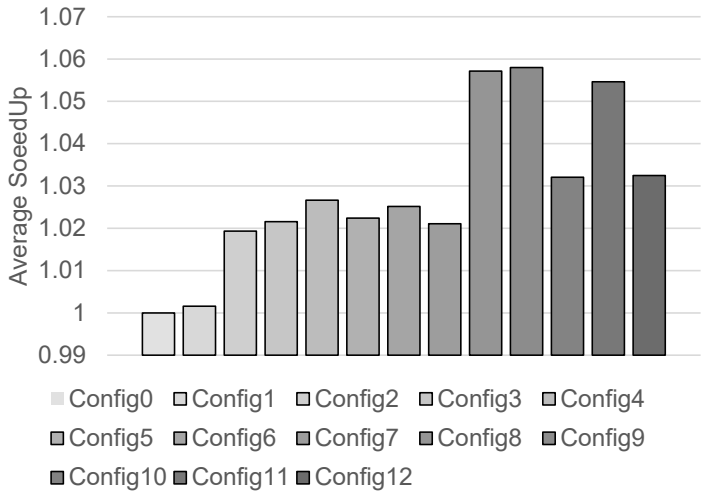


Figure 7.7: Performance when varying the parameters of UM block correlation table.

fig0 using V100 32GB GPU for each configuration. We see that, on average, configuration *9* shows the best performance.

7.2.4 Comparison with TensorFlow-based Approaches

We compare the performance of DeepUM with TensorFlow-based approaches: vDNN[23], AutoTM[27], SwapAdvisor[28], Capuchin[26], and Sentinel[29]. Figure 7.8 shows the speedup for each DNN model. The speedup is obtained on a V100 16GB GPU. Note that the numbers are obtained from Ren et al.[29], and they measure the speedup of the training throughput over NVIDIA UM without prefetching.

Table 7.8 shows the maximum possible batch size of the TensorFlow-based approaches and DeepUM. To measure the maximum possible batch sizes, we limit the total CPU memory usage of DeepUM to 128GB to match the system configuration with the TensorFlow-based approaches.

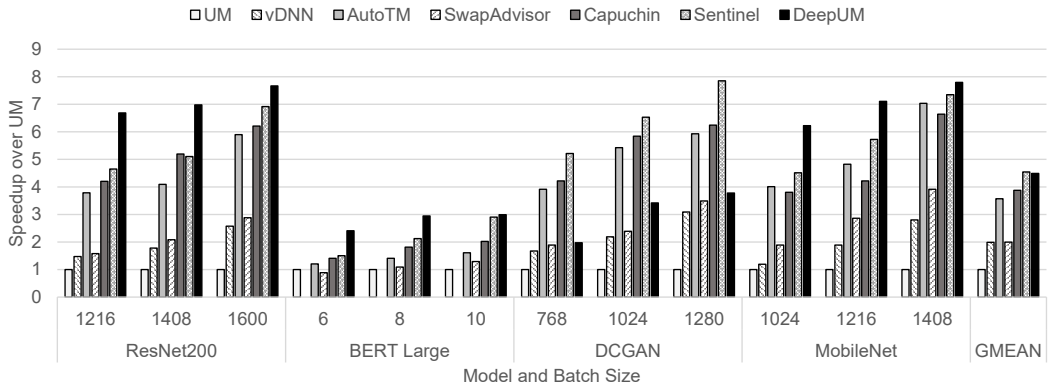


Figure 7.8: Comparison with TensorFlow-based approaches.

Table 7.8: Maximum possible batch sizes of TensorFlow-based approaches and DeepUM.

Model (Dataset)	ResNet200 (CIFAR-10)	BERT Large (CoLA)	DCGAN (celebA)	MobileNet (CIFAR-100)
vDNN	4.2K	not work	1.4K	1.2K
AutoTM	5.6K	27	2.5K	3.2K
SwapAdvisor	5.4K	25	2.4K	3.1K
Capuchin	5.9K	27	2.7K	3.2K
Sentinel	5.7K	28	2.5K	3.2K
DeepUM	6.4K	64	3.5K	5.1K

Overall, DeepUM is faster than IBM LMS and other TensorFlow-based approaches than Sentinel. DeepUM shows comparable performance to Sentinel. Note that Sentinel's swapping mechanism is not transparent to the user while DeepUM's is fully automatic. Moreover, DeepUM allows a much larger batch size than other previous approaches.

Note that previous approaches manage data at the DNN layer or tensor level. It implies that all data accessed in a layer or a tensor are moved together. The performance difference comes from the more fine-grained data movement of DeepUM, where memory objects are prefetched and evicted in a more fine-grained manner with accurate prediction through the correlation tables.

7.3 Framework for Virtualizing Single Device Image for a GPU Cluster

In this section, we evaluate SnuRHAC with various GPU applications and analyze the result. We also compare the performance of SnuRHAC with that of hand-written multi-GPU applications.

7.3.1 Methodology

System configuration. A total of eight nodes are used for the evaluation. Each node has four NVIDIA Tesla V100 GPUs[92]. The interconnection network is Mellanox InfiniBand EDR. Table 7.9 shows the detailed system configuration.

Benchmark applications. We use 18 applications from various sources: three applications from CUDA Code Samples[33], three applications from Parboil[31], eight applications from PolyBench[37] and four applications from Rodinia[32]. These applications are summarized in Table 7.10. We select these applications from the benchmark suites based on the following criteria:

Table 7.9: System configuration for evaluation of SnuRHAC.

Number of nodes	8
CPU	2 × Intel 2.10 Ghz 20-core Xeon Gold 6230 for each node
Main memory	768GB DDR4 for each node
OS	Ubuntu 18.04.4 LTS (kernel 4.15.0-101)
GPU	4 × NVIDIA Tesla V100 PCIe (32GB device memory for a GPU) for each node
GPU driver	NVIDIA display driver 440.64
CUDA version	10.1
Interconnect	Mellanox InfiniBand EDR
MPI version	OpenMPI 4.0.3

- The longest kernel's execution time on a single GPU should be longer than 0.1 seconds when the largest dataset that fits in the device memory is used. Kernels that have too short execution time are not suitable to run them using multiple GPUs. Kernel launch overhead may be more significant than the actual kernel execution time. We conclude that a kernel execution time should be at least a few milliseconds when using 32 GPUs. This leads to the 0.1-second limit for a single GPU.
- The application should not use CUDA graphics API or additional CUDA libraries (cuBLAS, cuFFT, cuSPARSE, cuSOLVER, nvGRAPH). There are two reasons for this: One is that our goal is to show how to provide an illusion of a single GPU on top of multiple GPU devices, not to implement wrappers for every possible case. The other is that CUDA kernel source code for each API function in those libraries is not available. In this case,

Table 7.10: Applications used for evaluation.

Suite	No.	Name	Device memory size
CUDA Code Samples	1	binomialOptions	1536MB
	2	MonteCarloMultiGPU	312MB
	3	Nbody	576MB
Parboil	4	mri-q	502MB
	5	sgemm	3224MB
	6	tpacf	609MB
PolyBench	7	2mm	1280MB
	8	3mm	1792MB
	9	correlation	128MB
	10	covariance	128MB
	11	gemm	3072MB
	12	gesummv	8192MB
	13	syr2k	768MB
	14	syrk	512MB
Rodinia	15	hotspot	12288MB
	16	lavaMD	7492MB
	17	myocyte	1MB
	18	particlefilter	3360MB

SnuRHAC cannot perform static prefetching. If the kernel source code of each API function is available, implementing wrappers for those libraries is trivial and does not take much time.

- The CUDA kernel should not use atomic operations that access the global memory. We exclude applications that use atomic operations because they are not suitable for multiple GPUs. Those applications will also be detected by the SnuRHAC runtime and will be directed to run using only a single GPU.

Before running each application using SnuRHAC, we run MAPA to perform static analysis on every kernel source code. MAPA writes the result to a file, and the SnuRHAC runtime reads the file at run time. Thus, the time taken to run MAPA is not included in the application execution time.

7.3.2 Results

Speedup. Figure 7.9 shows the speedup of each application by varying the number of GPUs from 1 to 32. As each node has 4 GPUs, the results of 1 GPU, 2GPUs, and 4GPUs use only a single node of the cluster. The speedup is obtained on a single GPU, which runs each application using the original CUDA runtime only. Note that the original CUDA runtime does not use UM. We measure the performance of SnuRHAC with three different schemes. SnuRHAC shows the performance of SnuRHAC without the static prefetching and dynamic prefetching techniques. SnuRHAC-SP shows the performance of SnuRHAC with the static prefetching technique and without the dynamic prefetching technique. SnuRHAC-SP-DP shows the performance of SnuRHAC using both techniques.

We use a correlation table with 64k rows, 4-way associative, 3 levels, and 4 successors for dynamic prefetching. Therefore, the size of a table is approxi-

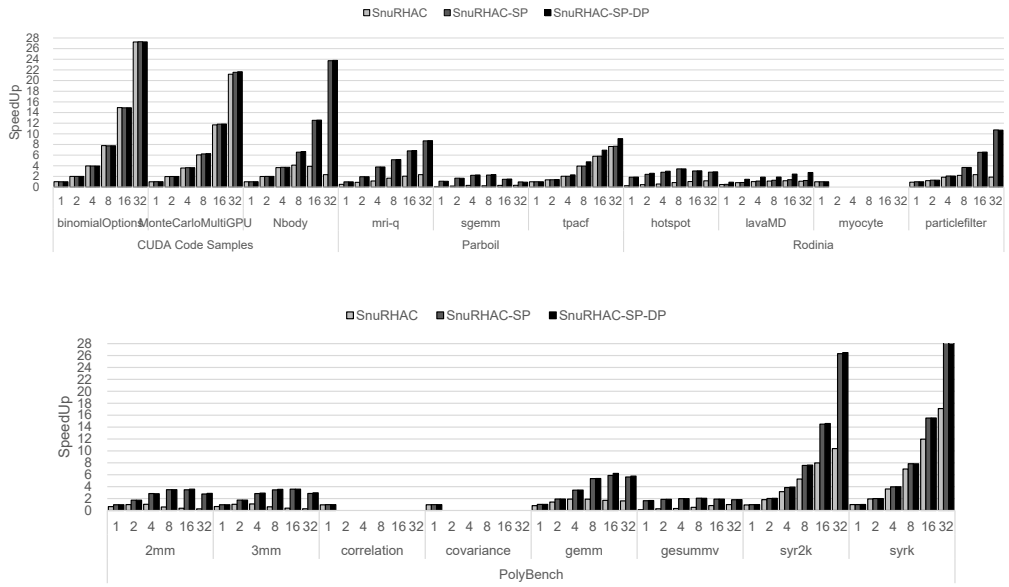


Figure 7.9: Speedup over a single GPU.

mately 12MB. We perform sensitivity analysis for the parameters of the correlation table later in this section. The threshold value for checking overlapping access ranges mentioned in Section 6.4 is set to 0.5. We empirically found that applications that show frequent page faults due to the page sharing between GPUs have a high ratio of the overlapping access ranges between GPUs. These applications include corrleation, covariance in PolyBench and myocyte in Rodinia. We also found that 0.5 is the proper value to enforce these applications to use a single GPU.

We use the timing routines that are already placed in the original benchmark source code for execution time measurement. Also, we exclude file I/O time, input data initialization time, and result verification time from the execution time measurement. These times are always the same in each application regardless of how many GPUs are used. We exclude them from the execution time measurement to clearly see how well the performance scales when the

number of GPUs is increased.

SnuRHAC-SP-DP shows the best performance for all applications. SnuRHAC-SP shows similar performance to SnuRHAC-SP-DP but shows notable differences for tpacf in Parboil and lavaMD in Rodinia. This is because MAPA cannot analyze most of the memory operations in the kernels, and thus static prefetching does not work well for these applications. SnuRHAC shows the poorest performance for most of the applications because pages are migrated on-demand. myocyte in Rodinia, correlation, and covariance in PolyBench show the result for a single GPU only because they are enforced to use a single GPU by SnuRHAC runtime because of the overlapping access ranges check.

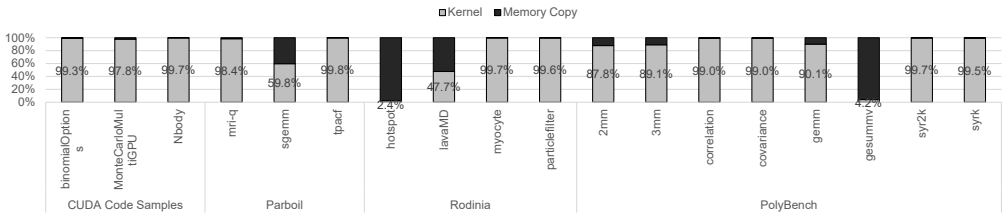


Figure 7.10: Breakdown of kernel execution time and memory copy time on a single GPU.

The scalability of the applications is significantly affected by the characteristics of the applications. Figure 7.10 shows the ratio of the kernel execution time to memory copy time measured on a single GPU using the CUDA runtime only. In general, applications that have a high ratio of the kernel execution time scale well in Figure 7.9. hotspot, lavaMD in Rodinia, and gesummv in PolyBench have very high ratio of memory copy time. They show poor scalability because they spend most of the execution time on copying data.

sgemm in Parboil, 2mm, 3mm, and gemm in PolyBench show relatively low memory copy time compared to the kernel execution time. However, they show

poor scalability because of the memory access patterns in the kernel. They all perform 2-D matrix multiplication. Suppose that we multiply matrix A with matrix B and then store the result to matrix C. To compute each row of matrix C, we need the corresponding row of matrix A and the entire matrix B. Each GPU in the cluster needs the full matrix B to compute some rows of matrix C. Thus, memory traffic over the network increases as the number of GPUs that participate in the computation increases.

mri-q in Parboil and particlefilter in Rodinia launches multiple kernels. Each kernel has different memory access patterns to the same memory object. It causes frequent data movement between GPUs on every kernel launch and degrades the performance.

Single GPU Performance. Table 7.11 shows the speedup of each application over the original CUDA runtime (CUDA) on a single GPU. Note that the original CUDA runtime does not use UM. SnuRHAC is much slower (0.628) than CUDA on average. SnuRHAC-SP and SnuRHAC-SP-DP are, on average, 1.023X and 1.059X faster than CUDA, respectively. Especially for hotspot and gesummv, SnuRHAC-SP-DP is 1.875X and 1.671X faster than CUDA. These two applications require relatively big device memory size (hotspot: 12GB and gesummv: 8GB in Table 7.10) and spend most of their execution time on memory copying (Figure 7.10). Thus prefetching is quite effective. The multi-threaded-copying mechanism in the SnuRHAC runtime from the host memory space to the host UM space boosts their performance, too.

Parameters for dynamic prefetching. As mentioned in Section 6.5.2, SnuRHAC uses correlation table for dynamic prefetching. There are several configuration parameters of the correlation table: the number of immediate successor pages

Table 7.11: Single GPU Performance

Application	SnuRHAC	SnuRHAC-SP	SnuRHAC-SP-DP
binomialOptions	1.005	1.002	1.002
MonteCarloMultiGPU	1.001	1.004	1.004
Nbody	1.002	1.002	1.003
mri-q	0.515	0.985	0.984
sgemm	0.114	1.095	1.080
tpacf	0.994	0.993	1.000
hotspot	0.260	1.856	1.875
lavaMD	0.483	0.511	0.920
myocyte	1.001	1.002	1.001
particlefilter	0.921	0.992	0.991
2mm	0.643	0.972	0.972
3mm	0.665	0.971	0.971
correlation	0.951	0.975	0.984
covariance	0.958	0.979	0.978
gemm	0.807	1.030	1.029
gesummv	0.117	1.635	1.671
syr2k	0.937	0.977	0.991
syrk	0.995	1.000	1.003
Geomean	0.628	1.023	1.059

per page fault (NumSuccs), the number of levels of successors to prefetch (NumLevels), the number of rows in the table (NumRows), and the associativity of the table (Assoc). To find the optimal configuration, we perform sensitivity analysis. Table 7.12 shows configurations for the correlation table and Figure 7.11 shows the speedup of SnuRHAC using 32 GPUs for each configuration. In Figure 7.11, we measure the geometric mean of speedup over the case that does not use dynamic prefetching. We see that, on average, configuration 5 shows the best performance.

Table 7.12: Configurations for sensitivity analysis

Name	Assoc	NumRows	NumSucc	NumLevels
Config1	2	32k	4	3
Config2	4	32k	4	3
Config3	8	32k	4	3
Config4	2	64k	4	3
Config5	4	64k	4	3
Config6	8	64k	4	3
Config7	2	128k	4	3
Config8	4	128k	4	3
Config9	8	128k	4	3

Multi-GPU applications. Some applications in CUDA Code Samples are designed to support multiple GPUs under a single operating system instance. We compare the performance of SnuRHAC with hand-written multi-GPU applications to show the effectiveness of SnuRHAC. Figure 7.12 shows the speedup obtained by varying the number of GPUs for these applications. CUDA is the

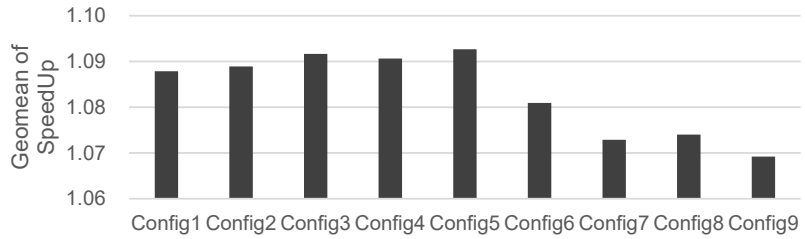


Figure 7.11: Varying the parameters of dynamic prefetching.

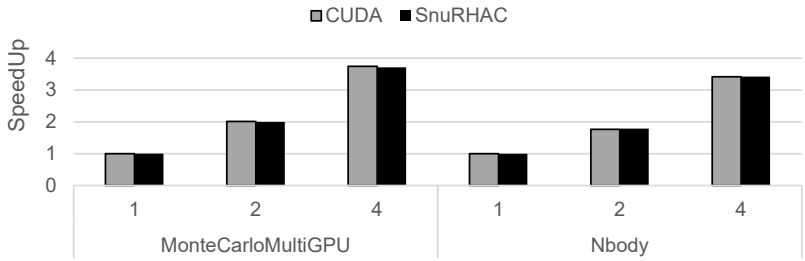


Figure 7.12: Speedup using multi-GPU applications.

result when running the applications under the original CUDA runtime, and SnuRHAC is the result when running them under SnuRHAC. The speedup is obtained over the case of a single GPU for CUDA. We do not vary the workload for multiple GPUs. Hence Figure 7.12 shows the result of strong scaling for both CUDA and SnuRHAC. The result indicates that SnuRHAC achieves almost the same scalability and performance as the hand-written multi-GPU applications.

Chapter 8

Discussions and Future Work

Supporting GPU clusters for large-scale DNNs. DeepUM mainly focuses on running large-scale DNNs on a single GPU. However, it can be extended to clusters easily. Since most of the DNN workloads do not need communication between GPUs except for all-reduce operation which occurs when calculating the average of the gradients, we can manage correlation tables for each GPU and prefetch pages for each GPU independently.

Enhance the workload distribution algorithm. SnuRHAC partitions the workload and distributes it to multiple GPUs using a simple algorithm. The algorithm tries to divide the CUDA grid in z-dimension first and then y-dimension second and then x-dimension last. However, we cannot guarantee that this is always the best way to partition the grid. We need to think about page migration between GPUs when there are multiple different CUDA kernel calls. To do this, the SnuRHAC runtime should carefully partition the grid using the memory access range information from MAPA to minimize the page migration

between different CUDA kernel calls.

Supporting multiple GPU architectures Frameworks introduced in this thesis supports CUDA-capable GPUs only. However, OpenCL also has a similar concept to Unified Memory which is called Shared Virtual Memory (SVM). We expect that the overall idea proposed in this thesis can be applied well for OpenCL-capable GPUs which support Shared Virtual Memory.

Chapter 9

Conclusion

In this thesis, we propose three frameworks that exploits UM to improve the ease-of-programming while maximizing the application performance.

HUM hides the host-to-device memory copy time by automatically overlapping it with the host computation or the kernel computation. It exploits Unified Memory and fault mechanisms of both the host and the GPU. HUM's Unified Memory is hidden to the programmer and there is no need to modify the source code. With 51 applications from Parboil, Rodinia, and CUDA Code Samples benchmark suites, we evaluate HUM. We compare their performance under HUM with that of their hand-optimized implementations. The evaluation result shows that HUM is quite effective and practical. On average, HUM achieves 1.20x for applications in Parboil, 1.26x for Rodinia, and 1.13x for CUDA Code Samples. The average speedup of all applications under HUM is 1.21, which is comparable to the average speedup 1.22 of the hand-optimized implementations for Unified Memory.

DeepUM allows GPU memory oversubscription for deep neural networks by

exploiting Unified Memory and using CPU memory as a backing store. While UM allows memory oversubscription using a page fault mechanism, it introduces enormous overhead. We use a correlation prefetching technique to hide its overhead. We also introduce several optimization techniques to minimize the GPU fault handling time. We use popular DNN models from various sources for evaluation. The evaluation result shows that DeepUM achieves comparable performance to the other state-of-the-art approaches. At the same time, DeepUM can handle larger models that other methods fail to run.

SnuRHAC extends Unified Memory across multiple nodes in a cluster. It is implemented with an additional Linux kernel module. It automatically distributes workload across multiple GPUs in a cluster, manages data across the nodes, and exploits static and dynamic page prefetching techniques to improve performance. These are all transparent to users, and no source code modification is required. Avoiding atomic operations and avoiding excessive write sharing optimizations filter applications that are not suitable for multiple GPUs and enforces those applications run on a single GPU only. The evaluation result of SnuRHAC with 18 applications from various sources indicates that SnuRHAC achieves scalable performance for the cluster environment depending on the application characteristics while significantly reducing the programmer's burden.

Bibliography

[1] J. Dongarra and P. Luszczek, *TOP500*, pp. 2055–2057. Boston, MA: Springer US, 2011.

[2] NVIDIA, "CUDA parallel computing platform." Website, 2021.

[3] Intel, "oneAPI programming model." Website, 2021.

[4] OpenACC-standard.org, "OpenACC." Website, 2021.

[5] K. group, "OpenCL overview - The Khronos Group Inc." Website, 2021.

[6] K. group, "SYCL overview - The Khronos Group Inc." Website, 2021.

[7] NVIDIA, "Unified Memory programming." Website, 2019.

[8] NVIDIA, "Pascal GPU architecture." Website, 2019.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[10] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the*

Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers) (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.

[11] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[12] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, (Cambridge, MA, USA), p. 1135–1143, MIT Press, 2015.

[13] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient data encoding for deep neural network training," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, p. 776–789, IEEE Press, 2018.

[14] 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 *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 926–939, 2020.

[15] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee, "FlashNeuron: SSD-Enabled Large-Batch training of very deep neural networks," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 387–401, USENIX Association, Feb. 2021.

[16] C. Li, R. Ausavarungnirun, C. J. Rossbach, Y. Zhang, O. Mutlu, Y. Guo, and J. Yang, "A framework for memory oversubscription management in graphics processing units," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 49–63, Association for Computing Machinery, 2019.

[17] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, (Cambridge, MA, USA), p. 3123–3131, MIT Press, 2015.

[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, p. 1737–1746, JMLR.org, 2015.

[19] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[20] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.

[21] A. Gruslys, R. Munos, I. Danihelka, M. Lanctot, and A. Graves, "Memory-efficient backpropagation through time," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*,

NIPS'16, (Red Hook, NY, USA), p. 4132–4140, Curran Associates Inc., 2016.

[22] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic gpu memory management for training deep neural networks," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, (New York, NY, USA), p. 41–53, Association for Computing Machinery, 2018.

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

[24] 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)*, pp. 143–152, 2018.

[25] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya, "TFLMS: Large model support in tensorflow by graph rewriting," *ArXiv*, vol. abs/1807.02037, 2018.

[26] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. 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 '20, (New York, NY, USA), p. 891–905, Association for Computing Machinery, 2020.

[27] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "AutoTM: Automatic tensor movement in heterogeneous memory systems using integer linear programming," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 875–890, Association for Computing Machinery, 2020.

[28] C.-C. Huang, G. Jin, and J. 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 '20, (New York, NY, USA), p. 1341–1355, Association for Computing Machinery, 2020.

[29] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 598–611, 2021.

[30] G. Jo, J. Jung, J. Park, and J. Lee, "Memory-access-pattern analysis techniques for OpenCL kernels," in *Languages and Compilers for Parallel Computing* (L. Rauchwerger, ed.), (Cham), pp. 109–126, Springer International Publishing, 2019.

[31] J. A. Stratton, C. Rodrigrues, I.-J. Sung, N. Obeid, L. Chang, G. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, Urbana, Mar. 2012.

[32] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing,"

in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Oct 2009.

[33] NVIDIA, "CUDA code samples." Website, 2019.

[34] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, and M. Zaharia, "MLPerf training benchmark," in *Proceedings of Machine Learning and Systems* (I. Dhillon, D. Papailiopoulos, and V. Sze, eds.), vol. 2, pp. 336–349, 2020.

[35] PyTorch, "PyTorch examples." Website, 2022.

[36] T. Wolf, J. Chaumond, L. Debut, V. Sanh, C. Delangue, A. Moi, P. Cistac, M. Funtowicz, J. Davison, S. Shleifer, *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, 2020.

[37] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *2012 Innovative Parallel Computing (InPar)*, pp. 1–10, 2012.

[38] K.-H. Kim and Q.-H. Park, "Overlapping computation and communication of three-dimensional FDTD on a GPU cluster," *Computer Physics Communications*, vol. 183, no. 11, pp. 2364 – 2369, 2012.

[39] A. Khajeh-Saeed and J. B. Perot, "Computational fluid dynamics simulations using many graphics processors," *Computing in Science Engineering*, vol. 14, pp. 10–19, May 2012.

[40] S. Georgescu and H. Okuda, "Conjugate gradients on multiple GPUs," *International Journal for Numerical Methods in Fluids*, vol. 64, no. 10-12, pp. 1254–1273, 2010.

[41] M. Bernaschi, M. Bisson, and D. Rossetti, "Benchmarking of communication techniques for GPUs," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 250 – 255, 2013.

[42] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–9, Nov 2008.

[43] E. H. Phillips and M. Fatica, "Implementing the Himeno benchmark with CUDA on GPU clusters," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–10, April 2010.

[44] J. B. White III and J. J. Dongarra, "Overlapping computation and communication for advection on hybrid parallel computers," in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 59–67, May 2011.

[45] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pp. 58–58, Nov 2005.

[46] L. Fishgold, A. Danalis, L. Pollock, and M. Swany, "An automated approach to improve communication-computation overlap in clusters," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pp. 7 pp.–, April 2006.

[47] A. Danalis, L. Pollock, M. Swany, and J. Cavazos, "MPI-aware compiler optimizations for improving communication-computation overlap," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 316–325, ACM, 2009.

[48] T. Gysi, J. Bär, and T. Hoefler, "dCUDA: Hardware supported overlap of computation and communication," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, (Piscataway, NJ, USA), pp. 52:1–52:12, IEEE Press, 2016.

[49] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20, (New York, NY, USA), p. 3505–3506, Association for Computing Machinery, 2020.

[50] P. Markthub, M. E. Belviranli, S. Lee, J. S. Vetter, and S. Matsuoka, "DRAGON: Breaking GPU memory capacity limits with direct NVM access," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 414–426, 2018.

[51] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *2009 42nd Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 45–55, 2009.

[52] K. Spafford, J. Meredith, and J. Vetter, "Maestro: Data orchestration and tuning for OpenCL devices," pp. 275–286, 08 2010.

[53] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, (New York, NY, USA), p. 277–288, Association for Computing Machinery, 2011.

[54] C. S. de la Lama, P. Toharia, J. L. Bosque, and O. D. Robles, "Static multi-device load balancing for OpenCL," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, pp. 675–682, 2012.

[55] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, p. 245–256, IEEE Press, 2013.

[56] T. Lutz, C. Fensch, and M. Cole, "Partans: An autotuning framework for stencil computation on multi-GPU systems," *ACM Trans. Archit. Code Optim.*, vol. 9, Jan. 2013.

[57] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," ICS '13, (New York, NY, USA), p. 149–160, Association for Computing Machinery, 2013.

[58] R. Kaleem, R. Barik, T. Shpeisman, B. T. Lewis, C. Hu, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), p. 151–162, Association for Computing Machinery, 2014.

[59] P. Pandit and R. Govindarajan, "Fluidic Kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, (New York, NY, USA), p. 273–283, Association for Computing Machinery, 2014.

[60] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating multiple data-parallel kernels on multiple devices," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 355–366, 2015.

[61] R. Nozal, J. L. Bosque, and R. Beivide, "EngineCL: Usability and performance in heterogeneous computing," *Future Gener. Comput. Syst.*, vol. 107, p. 522–537, June 2020.

[62] M. Strengert, C. Müller, C. Dachsbacher, and T. Ertl, "CUDASA: Compute unified device and systems architecture," in *Proceedings of the 8th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '08, (Goslar, DEU), p. 49–56, Eurographics Association, 2008.

[63] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, pp. 224–231, 2010.

[64] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "DS-CUDA: A middleware to use many GPUs in the cloud environment," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pp. 1207–1214, 2012.

[65] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki, "Hybrid OpenCL: Enhancing OpenCL for distributed processing," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*, pp. 149–154, 2011.

[66] J. Kim, S. Seo, J. Lee, J. Nah, and G. Jo, "OpenCL as a programming model for GPU clusters," vol. 7146, pp. 76–90, 01 2013.

[67] A. Alves, J. Rufino, A. Pina, and L. P. Santos, "ClOpenCL: Supporting distributed heterogeneous computing in HPC clusters," Euro-Par'12, (Berlin, Heidelberg), p. 112–122, Springer-Verlag, 2012.

[68] P. Kegel, M. Steuwer, and S. Gorlatch, "DOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, (USA), p. 174–186, IEEE Computer Society, 2012.

[69] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), p. 341–352, Association for Computing Machinery, 2012.

[70] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger, "DistCL: A framework for the distributed execution of OpenCL kernels," in *2013 IEEE*

21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 556–566, 2013.

[71] J. Kim, G. Jo, J. Jung, J. Kim, and J. Lee, "A distributed OpenCL framework using redundant computation and data replication," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), p. 553–569, Association for Computing Machinery, 2016.

[72] L. Liao, K. Li, K. Li, C. Yang, and Q. Tian, "UHCL-Darknet: An OpenCL-based deep neural network framework for heterogeneous multi-/many-core clusters," in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, (New York, NY, USA), Association for Computing Machinery, 2018.

[73] K. Balhaf, M. A. Alsmirat, M. Al-Ayyoub, Y. Jararweh, and M. A. Shehab, "Accelerating Levenshtein and Damerau edit distance algorithms using GPU with unified memory," in *2017 8th International Conference on Information and Communication Systems (ICICS)*, pp. 7–11, 2017.

[74] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "CRUM: Checkpoint-restart support for CUDA's Unified Memory," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 302–313, 2018.

[75] D. Ganguly, Z. Zhang, J. Yang, and R. 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 '19, (New York, NY, USA), p. 224–235, Association for Computing Machinery, 2019.

[76] Q. Yu, B. Childers, L. Huang, C. Qian, and Z. Wang, "HPE: Hierarchical page eviction policy for Unified Memory in GPUs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2461–2474, 2020.

[77] T. Brokhman, P. Lifshits, and M. Silberstein, "GAIA: An os page cache for heterogeneous systems," in *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, (USA), p. 661–674, USENIX Association, 2019.

[78] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-aware unified memory management in GPUs for irregular workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), p. 1357–1370, Association for Computing Machinery, 2020.

[79] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader, "Traversing large graphs on GPUs with Unified Memory," *Proc. VLDB Endow.*, vol. 13, p. 1119–1133, Mar. 2020.

[80] M. Harris, "Unified Memory for CUDA beginners." Website, 2017.

[81] NVIDIA, "CUDA runtime API: Memory management." Website, 2019.

[82] NVIDIA, "NVIDIA driver downloads." Website, 2021.

[83] M. Harris, "How to overlap data transfers in CUDA C/C++." Website, 2012.

[84] F. S. Foundation, "mprotect(2) - Linux manual page." Website, 2019.

[85] T. Alexander and G. Kedem, "Distributed prefetch-buffer/cache design for high performance memory systems," in *Proceedings. Second International Symposium on High-Performance Computer Architecture*, pp. 254–263, 1996.

[86] An-Chow Lai, C. Fide, and B. Falsafi, "Dead-block prediction dead-block correlating prefetchers," in *Proceedings 28th Annual International Symposium on Computer Architecture*, pp. 144–154, 2001.

[87] T.-F. Chen and J.-L. Baer, "Reducing memory latency via non-blocking and prefetching caches," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, (New York, NY, USA), p. 51–61, Association for Computing Machinery, 1992.

[88] D. Joseph and D. Grunwald, "Prefetching using Markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[89] Y. Solihin, Jaejin Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *Proceedings 29th Annual International Symposium on Computer Architecture*, pp. 171–182, 2002.

[90] R. Consortium, "RDMA consortium." Website, 2021.

[91] NVIDIA, "CUDA toolkit documentation." Website, 2021.

[92] NVIDIA, "Artificial intelligence architecture — NVIDIA Volta." Website, 2019.

[93] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow,

A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[94] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[95] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *ArXiv*, vol. abs/1810.04805, 2019.

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

[97] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[98] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," in *4th International Conference on Learning Representations, ICLR 2016, San*

Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings (Y. Bengio and Y. LeCun, eds.), 2016.

[99] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *ArXiv*, vol. abs/1704.04861, 2017.

[100] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), p. 372–383, Association for Computing Machinery, 2019.

초록

Unified Memory (UM)는 CUDA 프로그래밍 모델에서 제공하는 기능 중 하나로 단일 메모리 주소 공간에 CPU와 GPU가 동시에 접근할 수 있도록 해준다. 이에 따라, UM을 사용할 경우 CUDA 프로그램에서 명시적으로 프로세서간에 데이터를 이동시켜주지 않아도 된다. 또한, CPU 메모리를 backing store로 사용하여 GPU의 메모리 크기 보다 더 많은 양의 메모리를 필요로 하는 프로그램을 실행할 수 있도록 해준다. 결과적으로, UM은 프로그래머의 부담을 크게 덜어주고 쉽게 프로그래밍 할 수 있도록 도와준다. 하지만, UM을 있는 그대로 사용하는 것은 성능 측면에서 좋지 않다. UM은 page fault mechanism을 통해 동작하는데 page fault를 처리하기 위해서는 많은 오버헤드가 발생하기 때문이다. UM을 사용하면서 최대의 성능을 얻기 위해서는 프로그래머가 소스 코드에 여러 힌트나 앞으로 CUDA 커널에서 사용될 메모리 영역에 대한 프리페치 명령을 삽입해주어야 한다.

본 논문은 UM을 사용하면서도 쉬운 프로그래밍과 최대의 성능이라는 두마리 토끼를 동시에 잡기 위한 방법들을 소개한다. 첫째로, HUM은 기존 CUDA 프로그램의 소스 코드를 수정하지 않고 호스트와 디바이스 간에 메모리 전송 시간을 최소화한다. 이를 위해, UM과 fault mechanism을 사용하여 호스트-디바이스 간 메모리 전송을 호스트 계산 혹은 CUDA 커널 실행과 중첩시킨다. 실험 결과를 통해 HUM을 통해 애플리케이션을 실행하는 것이 그렇지 않고 CUDA만을 사용하는 것에 비해 평균 1.21배 빠른 것을 확인하였다. 또한, Unified Memory를 기반으로 프로그래머가 소스 코드를 최적화한 것과 유사한 성능을 내는 것을 확인하였다.

두번째로, DeepUM은 UM을 활용하여 GPU의 메모리 크기 보다 더 많은 양의 메모리를 필요로 하는 딥 러닝 모델을 실행할 수 있게 한다. UM을 통해 GPU 메모리를 초과해서 사용할 경우 CPU와 GPU간에 페이지가 매우 빈번하게 이동하는데, 이때 많은 오버헤드가 발생한다. 두번째 방법에서는 correlation 프리페칭 기법을

통해 이 오버헤드를 최소화한다. 실험 결과를 통해 DeepUM은 기존에 연구된 결과들과 비슷한 성능을 보이면서 더 큰 배치 사이즈 혹은 더 큰 하이퍼파라미터를 사용하는 모델을 실행할 수 있음을 확인하였다.

마지막으로, SnuRHAC은 클러스터에 장착된 여러 GPU를 마치 하나의 통합된 GPU처럼 보여준다. 따라서, 프로그래머는 여러 GPU를 대상으로 프로그래밍하지 않고 하나의 가상 GPU를 대상으로 프로그래밍하면 클러스터에 장착된 모든 GPU를 활용할 수 있다. 이는 SnuRHAC이 Unified Memory를 클러스터 환경에서 동작하도록 확장하고, 필요한 데이터를 자동으로 GPU간에 전송하고 관리해주기 때문이다. 또한, UM을 사용하면서 발생할 수 있는 오버헤드를 최소화하기 위해 다양한 프리페칭 기법을 소개한다. 실험 결과를 통해 SnuRHAC이 쉽게 GPU 클러스터를 위한 프로그래밍을 할 수 있도록 도와줄 뿐만 아니라, 애플리케이션 특성에 따라 최적의 성능을 낼 수 있음을 보인다.

주요어: GPU, CUDA, Unified Memory, 프리페칭, 계산-통신 중첩, 거대 딥러닝 모델, 클러스터
학번: 2014-21775