GDM: Device Memory Management for GPGPU Computing

Kaibo Wang
The Ohio State University
wangka@cse.ohio-state.edu

Xiaoning Ding
New Jersey Institute of Technology
xiaoning.ding@njit.edu

Rubao Lee
The Ohio State University
liru@cse.ohio-state.edu

Shinpei Kato
Nagoya University
shinpei@is.nagoya-u.ac.jp

Xiaodong Zhang
The Ohio State University
zhang@cse.ohio-state.edu

ABSTRACT

GPGPUs are evolving from dedicated accelerators towards mainstream commodity computing resources. During the transition, the lack of system management of device memory space on GPGPUs has become a major hurdle. In existing GPGPU systems, device memory space is still managed explicitly by individual applications, which not only increases the burden of programmers but can also cause application crashes, hangs, or low performance.

In this paper, we present the design and implementation of GDM, a fully functional GPGPU device memory manager to address the above problems and unleash the computing power of GPGPUs in general-purpose environments. To effectively coordinate the device memory usage of different applications, GDM takes control over device memory allocations and data transfers to and from device memory, leveraging a buffer allocated in each application's virtual memory. GDM utilizes the unique features of GPGPU systems and relies on several effective optimization techniques to guarantee the efficient usage of device memory space and to achieve high performance.

We have evaluated GDM and compared it against state-of-the-art GPGPU system software on a range of workloads. The results show that GDM can prevent applications from crashes, including those induced by device memory leaks, and improve system performance by up to 43%.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems; D.4.8 [**Operating Systems**]: Performance—*Memory management*

Keywords

GPU; Memory Management; Operating System

SIGMETRICS'14, June 16–20, 2014, Austin, Texas, USA.

http://dx.doi.org/10.1145/2591971.2592002.

1. INTRODUCTION

General-purpose GPUs (Graphics Processing Units), a.k.a. GPGPUs[1], are quickly evolving from conventional, dedicated accelerators towards mainstream commodity computing devices, which is driven by the demands for cost-effective high performance from new application domains and supported by GPU hardware and system software advancement [35, 37, 39, 16]. During the transition, system software plays an increasingly important role on managing GPUs. System software relieves application developers from explicit resource management in their programs. It must also coordinate the utilization of GPU resources, ensuring that applications can make continuous progress and no application can be deprived of resource usage indefinitely [22, 23].

Recent research and improvements on GPGPU resource management have mainly focused on supporting GPU abstractions [41], GPU file system [42], and the management of GPU computing units [30, 26, 31]. These system enhancements improve the usability and performance of GPGPU computing. However, despite these improvements, with state-of-the-art GPGPU system software an application still can easily crash, hang, or lose performance (§2.3). The major reason behind this problem is the lack of GPU device memory management at the operating system (OS) level, which has become a major hurdle of GPGPUs as truly general-purpose mainstream computing facilities. This paper identifies these problems and systematically studies the essentiality and design of GPGPU device memory management.

1.1 Problems with Application-Level Device Memory Management

Device memory is the primary onboard DRAM storage for the computation performed on GPU. Unlike system memory where the OS controls space allocation and reclamation, GPU device memory is still directly controlled by individual applications in current systems, which complicates GPGPU application design. In large applications, managing the usage of device memory space is a heavy burden for programmers. There have been numerous reports on application and system crashes [1, 2, 3, 4, 5, 6] caused by application's failure to manage device memory correctly.

Managing device memory space at application level becomes even more difficult when there are multiple applica-

[1]We use GPU and GPGPU interchangeably, with the latter emphasizing more on general-purpose computing.

tions or application components (e.g., multiple worker threads in a server) with conflicting demands for device memory. Due to the lack of an arbitrator to coordinate the conflicts, applications can crash or hang on unexpected shortage of device memory space. Even if an application may manage to survive by using smaller device memory space or shifting computation back to the CPU, its performance can suffer dramatically.

For instance, in Matlab, each worker thread can offload its computation tasks to GPUs for acceleration. However, if their working sets cannot totally fit into the device memory, some workers can easily fail or encounter severe performance degradation [1]. Device memory conflicts will become increasingly common, when GPGPUs are more prevalently adopted in large-scale applications (e.g., Matlab, AutoCAD, relational databases, etc.), or in the cloud where resources are shared by virtual machines [26].

We will discuss the problems with existing GPU system design in more details in §2.3 and illustrate their consequences in §6.

1.2 GDM: OS Device Memory Management

As a critical system resource, device memory space must be managed by the OS to effectively coordinate conflicting demands and to guarantee efficiency. In this paper, we present the design and implementation of GDM(**G**PGPU **D**evice-memory **M**anager) in the OS. With experiments, we show that such a device memory management component in the OS is indispensable for unleashing the high computing power of GPUs in general-purpose systems.

Without requiring modifications to existing APIs, GDM transparently takes control over device memory allocations and data transfers to and from device memory. Instead of letting applications directly allocate device memory space and exchange data with GPUs, GDM sits between applications and GPU devices, acting as an agent and coordinator for carrying out these operations, leveraging a *staging area* created in each application's virtual memory space. It monitors the utilization of device memory space allocated to each application, and dynamically reclaims under-utilized space by swapping out the content to staging areas. In this way, GDM controls and coordinates the actual device memory consumption of different applications, or different phases of a single application, so as to achieve system-wide benefits of performance and service quality. With support of GDM, even multiple applications with conflicting memory requirements can efficiently share the same GPU and make progress concurrently. As we will demonstrate in §6, GDM also enhances the capability of GPGPU systems to tolerate device memory leaks and defend against malicious device memory usage.

The above benefits, however, do not come without any overhead, which is mainly from the extra data movements incurred by GDM management. Several unique characteristics of the GPGPU system make it especially challenging to reduce the overhead. Firstly, GPGPU applications are usually data-intensive. Thus, GDM must handle large sets of data that potentially incur high cost. Secondly, the data-driven nature of GPGPU computing involves synchronizations at various stages, which hinder the overlapping between data transfer and the computation over the data. This makes the performance of GPGPU applications sensitive to the delay caused by data movement. Finally, GPU devices may lack necessary hardware support for efficiently minimizing the overhead, which makes the solution even more challenging. To address these challenges, we have developed a series of optimization techniques in GDM, such as object-level access pattern inference, hashing-based dirty block detection, and cost-aware data replacement policy. These techniques can effectively reduce unnecessary data movement to achieve high performance.

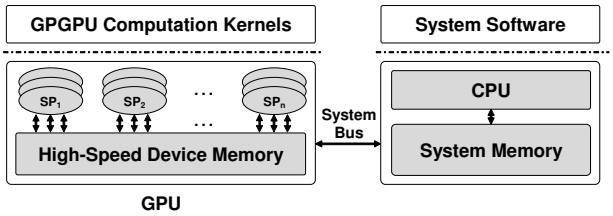


Figure 1: GPGPU system organization.

1.3 Contributions

This paper systematically studies the essentiality and design of GPGPU device memory management. It makes the following main contributions: (1) We have identified and analyzed the serious problems caused by the lack of OS management of device memory space on existing GPGPU systems. (2) We have explored the design space of managing device memory at system level. (3) We have implemented a prototype of GDM in an open-source GPGPU driver and on commonly used hardware to best utilize device memory resource for general-purpose systems, including a set of optimization techniques and principles that are crucial to the performance of device memory management for GPUs. (4) We have conducted extensive performance evaluation on GPU systems with insights. The experiments show that GDM can effectively prevent applications from crashing or stalling due to unexpected shortage of free device memory space. The experiments also show that the optimization techniques can increase system throughput by up to 46%.

2. DEMAND FOR SYSTEM-LEVEL DEVICE MEMORY MANAGEMENT

To deliver high performance, GPGPU computing not only relies on vectorized GPU processors to process data in parallel but also requires high-speed memory to guarantee fast data accesses. Thus, a common practice is to integrate GPU processors with device memory on the same GPU board, which is connected with the system bus to accept data-parallel tasks. This section introduces the system organization, which this paper mainly focuses on, and validates the indispensability of efficient device memory management.

2.1 GPGPU Computing Architecture

GPUs are suitable for performing data-parallel computation. They are often used together with the CPUs to form a hybrid computing system, as shown in Figure 1.

For high performance, a GPU usually has tens of hundreds of stream processors (SPs). Each SP is a many-lane SIMD (Single Instruction Multiple Data) engine. To satisfy data accesses from such a large number of SPs, a wide and fast memory interface must be employed. The device memory designed for GPUs is therefore optimized for high bandwidth and integrated close to the SPs.

Generally, the bandwidth of GPU device memory is several times higher than the bandwidth of system memory accessed by CPUs, which emphasizes more on low latency. For example, a server-class NVIDIA Tesla K10 GPU provides over 300 GB/s device memory access. In contrast, the maximum memory bandwidth of a similar-level Intel Xeon E5-4650 CPU can only reach about 50 GB/s. Compared with system memory, the capacity of device memory, however, is much more limited, due to the pincount and power constraints suffered by the memory technology (e.g. GDDR) used for GPUs [33]. For example, a high-end GPU card is usually equipped with only a few gigabytes of device memory, while tens of gigabytes of system memory has been common on a modern server for years.

GPUs are connected to the system bus to accept data-parallel tasks, which are often called GPGPU kernels (or *kernels* for brevity), and the data to be processed. GPGPU system software is responsible for task scheduling, initiating data transfers, and handling task exceptions. The operations performed by system software are mostly control-intensive, and thus can only be executed efficiently on CPUs.

In the paper, we mainly target the mainstream GPGPU computing architecture described above, in which dedicated device memory modules are used by GPUs to maximize throughput. Another GPGPU architecture, represented by AMD's APU [7], fuses graphics units and CPU cores on the same die and lets them share system memory. It cannot provide the same high computing power as a GPU with dedicated device memory does. Processors with the fused architecture are mostly used in mobile and low-end desktop systems to handle graphics workload at a low cost. The performance is bottlenecked by the number of graphics units that can be integrated on the same CPU chip and by the narrow system memory bandwidth contended by both CPU and graphics cores. To alleviate memory bandwidth bottleneck, there are proposals to integrate fast memory modules (e.g. stacked memory [18] or eDRAM [40]) into this architecture. These memory modules will play an important role to improve the performance of computation on the graphics cores. The principles and techniques developed in this paper can be adapted to manage these memory modules and other accelerators (e.g., DSP) with similar memory structures as well.

2.2 Device Memory: A Critical Resource

Device memory provides a high-speed data storage for GPGPU computing, and must be well managed in order to achieve high performance. Despite its limited capacity, applications have high demands for device memory space. On one hand, as applications become increasingly data-intensive, the data sets handled by a GPGPU task also grows rapidly, requiring larger device memory space. On the other hand, GPGPU applications tend to keep their working sets on the device memory for future reuses to minimize data transfers.

As an example, when GPUs are used to process database queries in data warehousing applications, main accelerator structures such as hash tables have to be loaded into device memory [45]. These data structures can be very large, especially for big-data problems [19]. Meanwhile, these data structures are usually used by different queries repeatedly. Keeping them in the device memory helps improving application performance. The small capacity and the high demands from applications make GPU device memory a critical but limited system resource.

2.3 Issues with Existing System Designs

Despite the cruciality of device memory it has not been well managed by the system. In a general-purpose computing environment, applications are still forced to manually manage device memory on their own. Before a task can be offloaded to GPU, the application must ensure that enough space has been reserved on the device memory and the working set of the task has been transferred to the reserved space. After the task finishes, it also has to decide whether the data sets should continue staying on the device memory in case of reuses by other tasks, or can be transferred back to the system memory to make room for other data to be processed.

The above design used to be more or less acceptable in the early era of GPGPU computing when GPUs were dedicated to applications with clear, static demands for device memory space. However, as both the scale and scope of GPGPU applications expand, it has become an increasingly heavy burden, or impossible, for programmers to correctly keep track of the demands for device memory space and manage the consumption accordingly.

For example, some applications consist of GPU-accelerated modules developed by different groups of developers, or third-party GPGPU libraries and runtimes (e.g., CULA [15], PyCUDA [8], and Theano [17]). It is difficult to monitor and coordinate the device memory space consumption of different components. Applications such as Matlab, Boinc, and GPU databases may also launch multiple workers, whose activities and demands for device memory space depend on user requests and are affected by the OS scheduling. It is laborious and inefficient to deal with such dynamics at the programming stage. When GPGPUs are shared by multiple applications (e.g. in the cloud), managing device memory space inside each individual application also leads to uncoordinated contention for the space.

Due to the complexity of managing device memory, applications may frequently experience shortage of free device memory space. For example, one worker thread may not be able to obtain enough device memory space if other worker threads have occupied too much of it. The application may crash or hang if it cannot handle the situation correctly. There have been an increasing number of device memory-related crash reports in both open-source and commercial GPGPU software such as Matlab [1], Boinc [2], and Theano [5]. An application may survive by reducing the granularity of GPGPU tasks or shifting computation back to the CPU dynamically. But either method can significantly reduce application performance. Please note that the shortage of free device memory may happen even when the allocated device memory space is not being actively used, which leads to resource underutilization.

The absence of system management of device memory also causes other system issues. For instance, device memory leaks are a common type of software bugs that exist in many real-world GPGPU systems, including key computation libraries [9, 3], popular language runtimes [5, 6], and widely-deployed applications [10, 11]. Without system management, the leaked memory space cannot be reclaimed until the leaking application crashes or is terminated. This shrinks the device memory space available to applications and significantly degrades system performance. Even worse,

without system management, a malicious program can reserve most device memory space without releasing it, causing the whole GPGPU system unusable.

2.4 Demand for System Management

The above issues cannot be effectively addressed at application level due to the lack of system-wide information and the authority required for managing a shared resource. For example, a library that implements device memory management functionalities can relieve the burden of application programmers. However, a library can only provide local management within each individual application or application component adopting the library. The conflicting demands between applications or application components still cannot be addressed.

To address the above issues, GPGPU system software must be enhanced to control device memory management. This will not only relieve application developers from this tedious obligation, but also present an arbitrator to coordinate the contention for device memory space. With the new improvements of GPU hardware and firmware, especially those to support multitasking [38, 25], the application domains and environments of GPGPU computing will continue expanding. The demand for system management of device memory space is also becoming more imperative.

The demand for system management of device memory is analogous to that for the OS to manage the physical space of system memory [21]. Before virtual memory was introduced, the large efforts spent by programmers to incorporate memory overlaying procedures into their programs proved inefficient and unrealistic as applications became increasingly complex. Nowadays, in almost all modern systems, the physical space of system memory is managed by the OS; applications just need to allocate and deallocate objects in their virtual spaces to use system memory.

However, unlike system memory management, because of the special characteristics of GPGPU systems and applications, the management of device memory must address a set of unique challenges to achieve high performance. We will introduce these challenges and the design of GDM in the next two sections.

3. GDM OVERVIEW

The objective of GDM is to take over the control of device memory space from applications without changing current APIs for device memory operations. For this purpose, GDM creates a ***Staging Area*** in each GPGPU application's virtual memory space. This staging area effectively serves as the device memory extensions for the GPU kernels launched in the application. Thus, device memory operations from the application can be redirected to the corresponding staging area, while the actual control of device memory space is released to GDM.

Figure 2 illustrates the positions of GDM and GDM staging areas in the system and how GDM interacts with other system components. GDM is built as part of the GPGPU driver in the OS. It intercepts and handles device memory related operations from GPGPU applications. GDM handles an allocation operation (e.g., *cuMemAlloc* in CUDA [39]) by allocating the required space in the staging area. Data to be transferred to device memory is first copied to the staging area, and is later transferred to the device memory when the kernel accessing the data is launched. This is shown

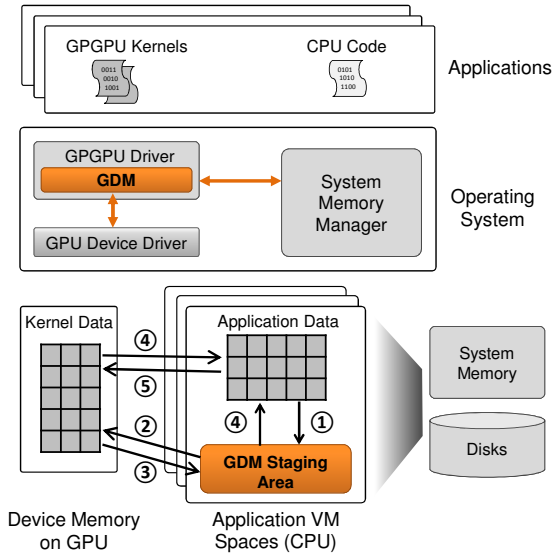


Figure 2: Overall Architecture of GDM.

with the arrows ① and ② respectively. After the kernel finishes, the data may stay in the device memory. When the device memory is short of free space, GDM transfers some data back to the corresponding staging area and reclaims the space, as shown by arrow ③. When an application calls the function to copy some data (e.g. computation results) from the device memory, GDM locates the latest version of the data (either in the staging area or in the device memory) and copies the data to the user buffer designated by the application. The arrows marked by ④ show the data transfers. To handle a deallocation operation (e.g. *cuMemFree* in CUDA), GDM frees and reclaims the corresponding space in both the staging area and the device memory.

3.1 Minimizing Overhead: A Major Challenge

GDM relieves applications from the burden of directly managing device memory space. While it avoids the problems due to uncoordinated usage of device memory space, the benefit does not come without cost. The main overhead of GDM is from the extra data copying (to and from staging areas) and data transferring (to and from device memory).

Some unique features of the GPU hardware and GPGPU application execution model make minimizing the overhead particularly challenging. Firstly, kernels running on GPU devices are usually data-intensive. Transferring large data sets over system bus may incur high overhead[2]. As shown from previous studies [41, 24] and our own measurement (§6.1), data transfers already account for a considerable portion of GPU operation time for many applications. If the amount of data movement incurred by GDM cannot be effectively controlled, the benefits of device memory management can be easily out-weighted by the potential high cost, diminishing the usefulness of the whole system.

Secondly, the performance of GPGPU applications is sensitive to the delays caused by bulky data transfers. The data-driven nature of GPGPU computing requires synchronizations at various stages. For example, a kernel cannot be launched before the transfer of its input data to the device memory finishes. These synchronization barriers reduce

[2]Data transfer rate via system bus is about one order of magnitude lower than device memory bandwidth.

the opportunities of overlapping operations before and after the synchronization points, making application performance sensitive to the delays on these operations. Among these operations, most are related to data transfers between the CPU and the device. Thus, the extra data transfers incurred by GDM management may degrade application performance if not handled properly. Meanwhile, system bus transactions are usually non-preemptive. A data transfer through PCIe, for example, cannot be interrupted once the DMA command is sent to the GPU copy engine. This exacerbates the problem caused by GDM-initiated data movement.

Finally, some hardware facilities for minimizing the cost have not been or cannot be efficiently implemented on GPUs. For example, in current GPU designs, there is no support for page reference bits to track fine-grained data access patterns. This poses great challenges to identifying inactive device memory areas. Hardware setting page dirty bits, a convenient feature for detecting data modifications, is also missing. On GPUs, page faults usually incur prohibitive costs [36]. On some GPGPU systems, page faults even cause application crashes. As far as we know, in the foreseeable future, there have not been clear plans on improving these facilities in GPU hardware.

To address these challenges, GDM minimizes the cost following two directions. One is to minimize data movements. This is achieved mainly through lazy copying, exploiting data locality, and careful classification of the data. The other direction is to reduce the latency incurred by data movements. GDM reduces the latency in two ways. Without compromising the correctness of program executions, GDM implicitly makes the handling of some heavy synchronous operations asynchronous to user programs, allowing programs to proceed while the operation is delegated to GDM for processing. GDM also internally breaks some bulky synchronous operations into several smaller pieces so that the processing of one piece can be overlapped with another to reduce costs. This also practically makes a long, bulky operation interruptible.

3.2 Guidelines for GDM Design

There are a few guidelines that have greatly influenced the design of GDM. One important guideline regards the choice between ***demand loading and anticipatory loading***. In the paper, we classify the methods of loading data to memory into two categories, namely *demand loading* and *anticipatory loading*. Demand loading refers to the method in which data loading is triggered by data accesses. It is usually achieved by hardware-supported exception mechanisms. For example, in demand paging, page fault handler is triggered automatically by hardware when a page being accessed is not present in the memory. The page fault handler loads the missing page from the disk and may prefetch a few more pages that it predicts to be accessed soon.

Anticipatory loading refers to the method in which the working set of a task is loaded into memory before the task is scheduled to run. It is the mechanism used in current GPGPU computing systems. An application reserves device memory space and transfers the working set of a GPU kernel to the device memory before it launches the kernel.

Demand loading is motivated by the high cost of loading data into memory. It pays the cost of handling page faults to load only the data that is demanded by the application and minimize the cost incurred by loading extra data. Anticipatory loading is more advantageous when the data sets handled by an application can be accurately determined before its execution. Though there are proposals to provide hardware support for demand paging on GPUs [36, 32], we argue that anticipatory loading will continue playing an important role in device memory management on future GPU devices. This is based on the following two observations.

In contrast with the data sets handled by CPUs, the data sets handled by GPUs are usually more predictable before the kernel starts execution. For example, some data sets to be referenced can be inferred from the data transfer APIs (e.g. *cuMemcpyHtoD* in CUDA) before launching a kernel; some are specified in the parameters of the GPU kernel. This makes anticipatory loading a viable approach in practice.

Handling page faults on a GPU incurs much higher overhead than doing so on a CPU, because it stalls a *faster* processor for a *longer* time. A GPU kernel is usually executed by hundreds of thousands of threads on a GPU, with the running state of each thread maintained in large register files, shared memory, and hardware caches which are often virtually addressed [34]. Saving the state of a GPU kernel on page faults, flushing caches, and restoring kernel state to resume execution thus take much longer time than that on the CPU. As the numbers of GPU cores and threads launched by GPU kernels keep increasing on future GPUs, the cost of handling page faults will also escalate significantly. Moreover, handling GPU page faults requires the involvement of system software running on the CPU (e.g., to carry out the corresponding memory management and re-scheduling operations). The extra delays and operations on the critical path of page fault handling further prolongs GPU stall time. The large overhead associated with page fault handling on a GPU thus may not be justified.

In GDM design, we use anticipatory loading for the data sets that are predicted to be accessed. To minimize the overhead incurred by handling page faults, demand loading is only used to handle unexpected data accesses if the GPU device supports page faults.

The second guideline regards the ***the granularities of device memory management*** to match the data-parallel nature of GPGPU computing. The granularities determine the units in which data in the staging areas and device memory space should be managed. In system memory management, memory page of a few kilobytes is a commonly used granularity by the OS. But this granularity is too small for GPGPU computing. The data-parallel feature of GPGPU computing determines that the data sets handled in GPGPU programs are usually very large (even the register file sizes on GPUs are at least hundreds of kilobytes). Using small granularities increases the overhead of managing metadata. More importantly, data transfers to and from the device memory in small units cannot amortize the start-up latency of memory controller, incurring prohibitive costs.

Memory regions have been used in several studies. A device memory region is allocated by the user program through the memory reservation API call, and can be as large as hundreds of megabytes or even gigabytes. Thus, data transfers in units of regions can cause high synchronization and data movement overhead. Moreover, managing data at the region level is incapable of capturing the distinct access patterns of user data structures created within a single region, which are important information for the management of device memory space.

Ideal granularities are those that can balance the latency and throughput of data movement and can preserve program-level data structure information to minimize overhead. GDM manages device memory space with both *block* and *object*-level information. We will introduce these concepts and how GDM utilizes them for management in the next section.

The third guideline regards the ***generality*** of GDM design. We realize that GPU hardware design is still evolving towards mature, general-purpose computing device. Thus, in our design, we do not exclude possible new features in future GPU hardware that may help with device memory management. At the same time, we try to keep the GDM design as general as possible. We explore the techniques that can minimize its reliance on the uncertainties of future GPU hardware features. This also helps it be adopted, starting with current GPU hardware.

4. GDM DESIGN

This section presents the details of GDM, focusing on the design tradeoffs and optimization techniques for minimizing the overhead of device memory management.

4.1 Staging Areas

GDM creates a staging area for each GPGPU application in its virtual memory space. Instead of using a large chunk of space with continuous virtual addresses, a staging area consists of a set of virtual memory areas. These areas are dynamically allocated when GDM handles the requests from applications for device memory reservations. Because these areas are located in the application's virtual memory space, physical memory is not allocated until they are populated.

Staging areas first serve as a temporary storage for the data to be transferred onto device memory. With staging areas, data transfers to the device memory can be fulfilled asynchronously. Specifically, when an application calls the API function to transfer some data from a user source buffer to the device memory, this function returns after GDM marks the source buffer copy-on-write. The data is transferred to the device memory later from the source buffer if it has not been changed (arrow ⑤ in Figure 2). Otherwise, the data is copied to the staging area when it is about to be changed in the source buffer, and is later transferred to the device memory from the staging area. In many GPGPU applications, a source buffer is often not modified before the data transferred from it is used in a kernel. Thus, copy-on-write can effectively reduce the memory consumption of staging areas and the cost incurred by data copying.

Staging areas also serve as the swap space for the data that can no longer stay on the device memory due to space contentions. When an application needs more free device memory space to launch kernels, GDM evicts some data from device memory to the staging area and reclaims the space for its own data sets. The data swapped to the staging area may later be loaded back to the device memory when the kernel referencing the data is to be issued.

Creating staging areas inside the virtual address spaces of applications provides a few benefits. The low-level management of staging areas, from space allocation/deallocation to data swapping between system memory and disks when the system memory is under pressure, relies on the existing virtual memory manager in the operating system. This, on one hand, simplifies the design of GDM. On the other hand, it puts the system memory space occupied by staging areas under the unified management with other system components and applications. This helps the operating system balance system memory usage for the overall benefit of system performance.

```
cuMemAlloc(&region1, 800MB);
dest_buf_1 = region1;
cuMemcpyHtoD(dest_buf_1, src_buf_1, 400MB);
dest_buf_2 = dest_buf_1 + 400MB;
cuMemcpyHtoD(dest_buf_2, src_buf_2, 200MB);
dest_buf_3 = dest_buf_2 + 200MB;
cuMemcpyHtoD(dest_buf_3, src_buf_3, 10KB);
```

Figure 3: Device memory region and object.

4.2 Device Memory Regions, Objects, Blocks

A fundamental design decision to make is the granularity at which the device memory space should be managed. One natural choice is device memory region. A device memory ***region*** is allocated/deallocated by the user program through the device memory reservation/release API calls. Applications may reserve different regions for different data sets to be handled by GPU kernels. In these cases, data in the same memory region may show good access uniformity; managing data based on regions can thus be an efficient choice. Regions have been used as the units of device memory management in some existing studies [31, 29].

However, in some important GPGPU applications [28, 44], we do see cases in which a memory region includes multiple data sets with distinct access patterns (e.g. data structures with different read/write properties or being referenced by different kernels). Data sets can also be shared among different GPU contexts easily with a single IPC call, which makes the program structure much clearer to maintain. For these applications, managing data with regions fails to classify fine-grained data access characteristics and increases both space and data movement overhead.

GDM identifies this demand and adds an object-based memory management layer below regions to differentiate data sets with different access patterns in each memory region. In GDM, an ***object*** is a data set handled by a data transfer operation. This is based on the observation that programs usually invoke separate data transfer API calls to pack multiple data sets into the same device memory region. The region area modified by each data transfer API call corresponds to an object in GDM. For efficiency, GDM merges small objects with their neighboring objects in the same device memory region. As an example, based on the pseudo code snippet in Figure 3, GDM creates one region (i.e. *region1*) and three objects, one for *dest_buf_1*, one for *dest_buf_2* , and one for the rest part of the region.

Objects can still be very large and cumbersome to manage. Meanwhile, object sizes usually vary widely in GPGPU programs, which introduces unnecessary complexity and overhead in memory management. For example, transferring large objects leads to high synchronization cost; evicting a whole object lowers the utilization of device memory if the required space is smaller than the object size. To address these problems, GDM further breaks objects into fixed-size ***blocks***. Then, it allocates/reclaims device memory space and transfers data in units of blocks. The block size is selected to effectively amortize the start-up latency of data transfers.

As will be explained in the following subsections, this hierarchical layout of regions, objects, and blocks makes the management of device memory space especially efficient.

4.3 Loading Data to Device Memory

For the correct and efficient execution of a kernel, GDM must load the working set of the kernel onto device memory. Basically, two key questions must be addressed: which data sets should be loaded, and when should they be loaded.

To address the first question, GDM uses different techniques for different types of GPU devices. If page faults are correctly supported on the GPU device, GDM monitors and analyzes the parameters used to launch a GPU kernel and the data transfer API calls made before the kernel launch. It extracts the objects involved in the parameters and API calls. Usually these objects are the data sets to be handled by the kernel. For these objects, GDM transfers the data block by block into the device memory before the kernel is issued (i.e. anticipatory loading). Other data sets, if accessed, will be loaded on demand on page faults.

If page faults are not supported, GDM by default loads the whole context to the device memory. To reduce data transferring, GDM provides interfaces for programs to specify objects needed by a kernel, with which advanced programmers can direct GDM to only load the specified objects.

For anticipatory loading, another key question is what is the good time to transfer the data sets to device memory. If a data set is transferred to the device memory too early, it may be evicted prematurely before the kernel referencing it is issued. This incurs extra data transfers. If a data set is transferred to the device memory too late, the execution of the kernel will be delayed.

In a busy system, where kernels queue up waiting to be issued, the system throughput depends on how quickly these kernels can be issued. Thus, the most efficient way is to transfer the data sets used by the kernels according to the order of the kernels in the queue. When GDM finishes transferring the data sets used by a kernel, it can start to transfer the data sets used by the next kernel in the queue. In this way, GDM can overlap data transfers with GPU computation to a great extent and minimize the time that kernels must wait for their working sets.

When all the launched kernels have been issued, if the device memory still has free space and the GPU copy engine is idle, GDM will trace back recent application requests of data transfers to the device memory for unfulfilled requests. As explained in Section 4.1, with staging areas, GDM handles data transfer requests to the device memory asynchronously. Thus, there may be some data sets in staging areas that have not been transferred to the device memory even the applications have requested to do so (e.g. by calling *cuMemcpyHtoD*). GDM takes the opportunity and loads these data sets onto device memory because they are more likely to be accessed in kernels soon. To prevent performance loss, GDM stops loading the data when the device memory is filled. GDM also stops loading the data when a kernel is launched, so that the GPU copy engine can be quickly released to transfer the data sets of the newly launched kernel.

4.4 Management of Device Memory Space

GDM makes every effort to satisfy the device memory demand of the kernel to be issued. If the device memory is short of free space, GDM must evict some data of finished kernels and reclaim the space. Thus, a core issue with the management of device memory space is data replacement, i.e. the policy that determines which data sets should be evicted when the free device memory space is insufficient.

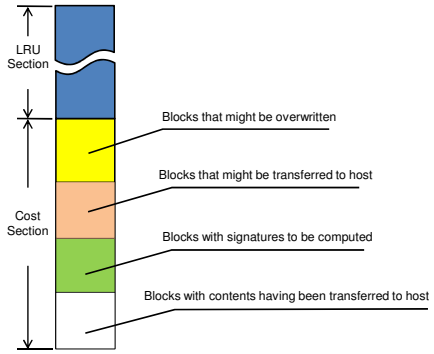


Figure 4: An LRU stack is structured for the LRU-COST replacement policy.

A large number of replacement policies have been proposed in previous studies of system memory and buffer management. The goal of these policies is mainly to maximize hit ratios, i.e. reuses of data in the memory. Every time when a replacement decision has to be made, these policies try to select an item that is least possible to be reused in the future.

However, conventional replacement policies are usually designed for systems where small amounts of data (e.g. pages or blocks) are loaded on demand. Directly adopting them will lead to sub-optimal performance in GPGPU systems, where usually a large amount of data (e.g. that can fill two thirds of device memory capacity) must be loaded before the corresponding kernel can start execution. Therefore, in GDM, other than maximizing hit ratios, the design of a replacement policy must achieve an additional goal — minimizing the time to spare the space for loading the data sets of the incoming kernel. The latency of readying the required space has direct impact on application performance.

GDM enhances the LRU replacement policy to maximize data reuses in the device memory and to minimize the latency of data eviction. The replacement policy with the enhancement is named *LRU-COST* [3]. LRU-COST uses a stack to manage the data sets loaded into the device memory. When a kernel is issued, all the data sets it will operate are put on the top of the LRU stack, pushing existing data sets in the stack down towards the bottom. As shown in Figure 4, LRU-COST partitions the stack into two sections. The part on the top is named *LRU section*, and the part at the bottom is named *COST section*. The size of COST section is from 0 to $selection\_factor \times len\_stack$, where $selection\_factor$ is an adjustable parameter with a default value of 0.2 and len_stack is the size of the whole stack. Data sets in the COST section are classified and sorted to minimize the eviction cost, as we will explain later. The data sets with the lowest costs are put at the bottom. When more free space is needed, LRU-COST selects the data sets at the bottom of COST section to evict. When COST section is depleted, it is refilled with the data sets in the LRU section that belong to the working sets of finished kernels. It preferentially selects the data sets at the bottom of the LRU section until it reaches its maximum size.

The cost of evicting a data set is determined by its status. Evicting a *clean* data set incurs lower cost than evicting a

[3]While other replacement policies can also be enhanced with similar approach, we select LRU because it is widely used and easy to implement.

dirty one. A data set is clean if it has not been changed since it is loaded to the device memory. Otherwise, the data set is dirty. The cost of evicting a clean data set is minimal, because there is a copy of the data set in the virtual memory space of the application, either in the staging area or in the corresponding source buffer. To evict a dirty data set, some cost has to be paid to transfer the data back to the staging area to preserve the changes.

In real-world applications, usually a considerable portion of GPU data is read-only during kernel executions. For example, based on our analysis of 16 benchmarks in the Rodinia benchmark suite [20], on average, 61% of all the GPU data referenced during kernel executions are not modified. Thus, there is a good potential to improve performance by preferentially evicting clean data sets.

In the future, the clean/dirty status of data sets may be traced by hardware automatically. However, as mentioned earlier, existing GPU hardware does not provide such support. Thus, after a kernel finishes execution, GDM has no immediate information to determine whether a data set has been modified. If GDM cannot find a way to differentiate clean data from dirty data, it must write back the data sets being evicted indiscriminately to the corresponding staging areas. This may significantly increase wasteful system bus traffic and cause delay of kernel execution.

To address this problem, GDM computes a signature for each data block in the COST section, which is a set of MD5 hash values of the data in the block (please refer to §5.2 for details). When the COST section is refilled, GDM immediately issues a maintenance kernel to compute the new signatures for the data blocks in the section from bottom to top. If the new signature of a block is different from its previous signature or if its previous signature does not exist, the block is marked dirty. For a 4MB block, its signature can be kept with only 4KB and is transferred along with the block. Computing the signature of a data block on a GPU can be made over an order of magnitude faster than transferring the data back to the staging area.

Although computing signatures can significantly reduce the data traffic on system bus, it increases the workload of GPU processors. When there are other application kernels that have been issued on the device, the maintenance kernel and the application kernels may compete for GPU processors. This reduces system throughput (because the execution of application kernels is delayed) and/or increase the latency incurred by data eviction (because computing signatures is delayed). To address the above problem, GDM uses three heuristics to reduce the amount of computation.

Uniformity Heuristic: In the same object, if the first data block, the data block in the middle of the object, and the last data block are dirty blocks, GDM assumes that other blocks are also dirty blocks. This is based on the observation that kernels usually carry out similar operations on the data in the same object, because of the data-parallel nature of GPGPU computing. This is to reduce signature computation for write-mostly data.

Overwrite Elimination: A data block in the device memory is invalidated and its space can be reclaimed when the application overwrites its content via a data transfer API call. This usually takes place when an application performs computation on a data set larger than device memory capacity. In a loop, the application repeatedly updates its data in the device memory and launches a kernel to process each partition of the data set. Thus, if GDM expects a block may be invalidated based on its overwriting history, it gives a low priority to computing its signature by putting it on the top of the COST section (see Figure 4). This heuristic can be used to reduce signature computation for both read-mostly and write-mostly data.

Double-Transfer Avoidance: A data block becomes a clean block if the application calls an API to copy its content out from the device memory. Thus, if GDM expects that a data block may be transferred to system memory upon application requests, it gives a low priority to computing its signature by putting it far away from the bottom of the COST section (refer to Figure 4). When the content has been transferred and the block is still in the COST section, GDM moves the block to the bottom of the stack.

To apply the last two heuristics, GDM keeps track of the blocks that changed status to "clean" or were overwritten, and use the information as hints to predict whether the behaviors will repeat.

5. IMPLEMENTATION

We have implemented a prototype of GDM in the GPGPU driver, Gdev [31], on Linux. We choose Gdev because it is open-source and has been shown in previous research [31] to perform comparably with the proprietary commercial CUDA system. Our prototype system targets discrete GPU cards, which are usually connected to the host CPU system through PCIe bus. In this section, we highlight some of the implementation details that deserve articulation.

5.1 Regions, Blocks, and Objects

When an application reserves a device memory region, GDM allocates two virtual memory areas for it: one from the CPU program's virtual memory space, which is used as the region's staging area; the other from the GPU context's virtual device address space, which is used by the GPU kernels to access data in the region. The starting address of the virtual device memory area is returned to the application as the identifier of the data region allocated. We set block size to 4MB in our prototype system, which is small enough compared with common object sizes in GPGPU programs and meanwhile preserves over 98% of PCIe efficiency. Nouveau [12], the GPU device driver Gdev relies on, currently does not allow users to allocate/deallocate virtual and physical device memory areas separately, nor does it support dynamic mapping/unmapping between them. We have thus modified the source code of Nouveau (less than 400 lines of changes) to expose such functionalities to GDM.

Objects are maintained by GDM to infer user data structures and improve data replacement efficiency. Initially, every data region is a single object. A host-to-device data transfer, if larger than block size, can split an object into two or three smaller objects and/or merge several objects into a larger one. In our implementation, objects are aligned at the block boundary.

5.2 Signature Computing

By definition, computing the MD5 hash value of a given data block is inherently sequential. A block has to be logically broken into 64-byte chunks. For each chunk, a 16-byte hash value is computed based on the data in this chunk and the hash value computed from the previous chunk. The hash value computed for the last chunk is used as the MD5 hash

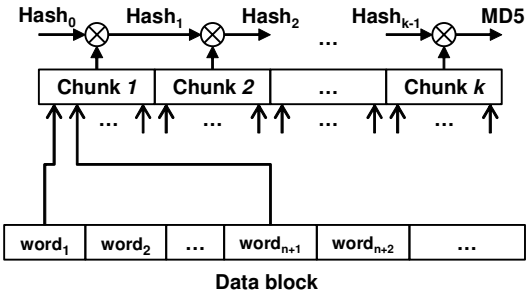


Figure 5: The Computation of data signatures.

value of the whole data block. To enable efficient parallelization on GPUs, GDM computes an array of MD5 hash values, instead of a single one, for a data block, and uses them together as the signature of the block.

This is illustrated in Figure 5. The data in a block are equally partitioned among GPU threads; each thread computes a MD5 hash value for the data assigned to it. Consecutive 4-byte words are allocated to different threads, so that device memory accesses can be coalesced for maximal kernel efficiency. Figure 5 shows how a data block is partitioned and the MD5 computed by one thread in an n-thread kernel. The signature of the data block comprises the MD5 values computed by all GPU threads. Optimal number of threads for computing the signature of a data block is determined by the block size and GPU hardware parameters. In our implementation, we set the number of threads per 4MB data block to 256, which achieves very good performance. To maximize GPU core utilization, GDM computes the signatures of multiple blocks in one kernel, which further improves performance.

6. EVALUATION

This section evaluates GDM under various workloads. Before presenting the results, we first introduce the setup and methodology of our experiments.

6.1 Experiment Setup and Methodology

The experiments were carried out on a machine equipped with a 2.80GHz Intel Core i7-860 CPU, 8GB system memory, and an NVIDIA GTX 480 GPU card[4]. The operating system is Red Hat Enterprise Linux 6 running 3.3 kernel. The GPU device driver is Nouveau patched with our code to enable separate allocation of virtual and physical device memory areas and dynamic mapping between them. The GPGPU drivers are Gdev (i.e. the stock Gdev) and GDM (i.e. Gdev with GDM enhancement). The CUDA compiler is NVIDIA nvcc 4.0. Excluding the space reserved by Gdev and Nouveau, there is about 1400MB of device memory space available to user applications.

The benchmarks used in our experiments, as listed below, represent a variety of typical GPGPU applications and systems, including scientific computing, databases, machine learning, and image processing. Among them, four benchmarks, *backprop*, *hotspot*, *nn*, and *srad*, were selected from the Rodinia benchmark suite. Other benchmarks were extracted from real-world applications or existing open-source projects.

[4]We choose a GPU with NVIDIA GF100 architecture because Gdev supports it most reliably. Other GPUs (e.g. Kepler) are fundamentally the same with respect to memory management.

Name	Size (MB)
backprop	668
hj	2114
hotspot	1316
kmeans	120

Name	Size (MB)
mpe	658
nn	1138
srad	619
theano	4253

Table 1: Total size of device memory space allocated in each benchmark.

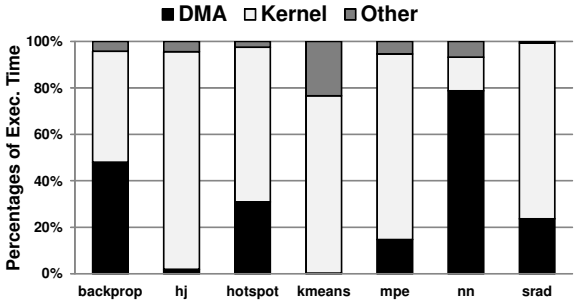
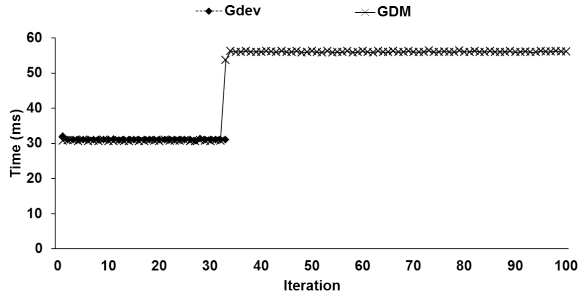


Figure 6: Decompositions of benchmark execution times.

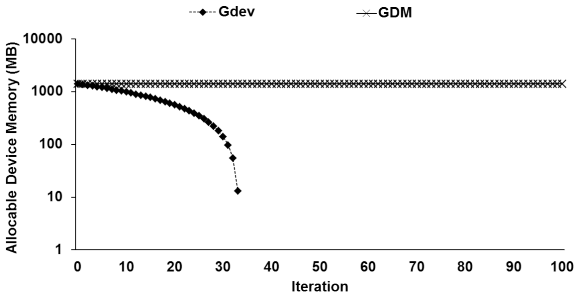
- **backprop** is an implementation of the backpropagation machine learning algorithm.
- **hj** performs a hash join operation over two database table columns. The data types of the two columns are both four-byte integers.
- **hotspot** solves a differential function extracted from a popular thermal modeling tool called HotSpot.
- **kmeans** implements the k-means clustering algorithm that partitions an array of multidimensional data elements into several clusters [13].
- **mpe** evaluates the value of expression $A \times B + C \times D$, in which A, B, C, D are four large matrices.
- **nn** computes the k-nearest neighbors of a given target within a large cloud of data points.
- **srad** is a diffusion method used mainly in ultrasonic and radar imaging applications to remove speckles.
- **theano** reproduces a real-world device memory leak bug in the Theano scientific Python library (the first commit in [5]). This bug is caused by incorrect increment of the reference counting value for a device memory region.

These benchmarks are highly optimized to maximize the utilization of GPU cores and have different demands for device memory space. The total size of device memory regions allocated in each benchmark is listed in Table 1. Figure 6 shows the percentages of execution time spent on DMA transfers, GPU kernels, and other operations (e.g., CPU computations and disk accesses) when each benchmark executes alone with stock Gdev. *theano* is DMA-bound and is not drawn because it crashes due to device memory leak and cannot finish execution with stock Gdev. As shown in Figure 6, all the benchmarks are GPGPU-intensive. At the same time, they incur different amount of data movement between the host and GPU device. Thus, the benchmarks can stress the design of GDM and test its components to minimize data movement overhead.

We evaluated GDM under two types of workloads: solo runs and combo runs. In a solo run, a benchmark executes alone. In a combo run, two benchmarks co-run with each other, and the benchmarks run multiple times to ensure the full overlap of their executions. Since our imple-



(a) Performance of theano over time.



(b) Size of allocable device memory over time.

Figure 7: The impact of device memory leaks with and without GDM management.

mentation of GDM is built into Gdev, we use the performance with the stock Gdev as baseline. The metrics we use to compare the performance are execution time for solo runs and weighted speedup for combo-runs. The weighted speedup of a combo run is the sum of the speedups of the participating benchmarks over their solo executions, i.e., $\sum_{i=1}^{n}(solo\_time_i/combo\_time_i)$ [43]. For brevity, we call the weighted speedup of a combo run its throughput.

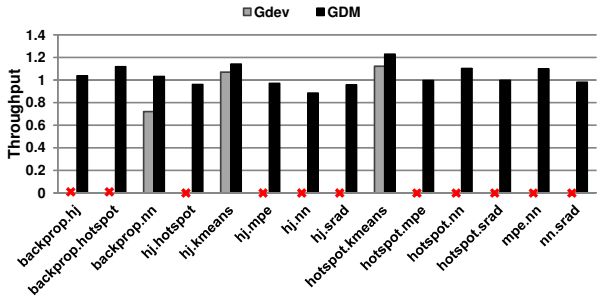
6.2 Tolerating Device Memory Leaks

Leveraging host memory and, subsequently, disks as swap space, GDM identifies and removes inactive blocks from the device memory. This greatly postpones device memory exhaustion and increases the capability of the system to tolerate device memory leaks.

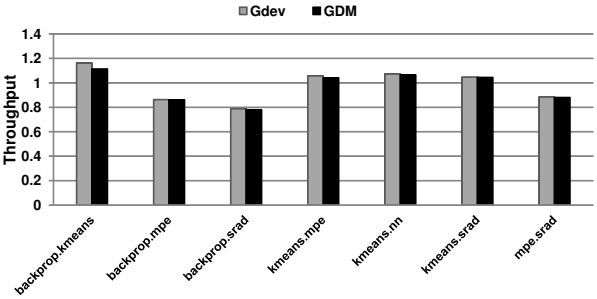
We use *theano* as a real-world example to demonstrate this advantage. *theano* is a Python script written using the Theano library. It invokes an in-place add operation for 100 times in a loop. In each iteration, the underlying Theano runtime creates an array on the device memory and launches a GPU kernel to update the elements in the array with an incremental value. However, there is a bug in Theano and the array is not released by Theano when the iteration finishes, adding a memory leak delta of about 40MB.

Figure 7(a) shows the per-iteration execution times of *theano*. Figure 7(b) illustrates the amount of allocable device memory (in logarithmic scale) after each iteration. Without GDM support, the leak quickly drains the device memory and causes *theano* to crash before one third of the iterations can be finished. Gdev cannot deal with device memory leaks in this case because data swapping is not fully supported.

In the experiment, we use the stock Gdev as a representative of exiting GPGPU systems without fully functional device memory management. However, the problem demonstrated with it is not limited to Gdev, but exists in similar



(a) Throughput of workloads with device memory contentions (A 'X' mark means the corresponding combo run crashed and the throughput could not be measured).



(b) Throughput of workloads without device memory contentions.

Figure 8: Performance of multitasking workloads with and without GDM management.

systems too. For example, we have also run *theano* with the commercial NVIDIA driver; it crashes after 36 iterations.

With GDM support, though the device memory may be filled, the space occupied by leaked regions can be reclaimed and is considered to be allocable. This allows *theano* to continue launching kernels and making progress continuously. Thus, *theano* is able to complete execution correctly. Since the leaked regions are modified during kernel executions, GDM incurs a constant overhead on data swapping after iteration 33. But, before GDM starts to swap out leaked regions, it does not slow down *theano*, as shown in Figure 7(a).

6.3 Multitasking Performance

In this subsection, we study the performance of GDM under multitasking workloads. We select all the possible combo runs consisting of two of the benchmarks excluding *theano*. We classify them into two groups based on their device memory demands. The first group have 14 combo runs with high demands. In each combo run, the total demand exceeds the device memory capacity, and the benchmarks contend for device memory space during the execution. The second group consists of the rest 7 combo runs. They have low demands for device memory space, and the benchmarks can share the device memory without contention. Before starting the combo-run experiments, we first measured the solo execution times of each benchmark with the stock Gdev and GDM respectively. Due to management activities, GDM performs slightly slower than the stock Gdev, but the difference is less than 2% on average.

Figure 8(a) shows the throughput of the combo runs in the first group with Gdev and GDM. With GDM, all the combo runs can finish their executions correctly. However, with Gdev, only 3 combo runs can finish without failures.

Eleven combo runs suffer program crashes, which happen to either one or both participating benchmarks. We have also performed the same experiment with commercial NVIDIA driver; all combo runs in the first group failed due to program crashes.

Most combo runs fail without GDM support because state-of-the-art approaches either do not support (e.g. CUDA) or use only primitive memory management policies (e.g. Gdev). For example, Gdev implements a simple data swapping mechanism based on its shared device memory support. With Gdev, when a region A in application P is to be loaded into the device memory short of free space, only a *single* region whose size is *larger* than A can be selected under the strict conditions that (1) it is not in application P, and (2) it has never replaced or been replaced by any other regions than A in P. If a region cannot be found to meet these constraints, the program may be blocked or crash due to insufficient device memory space for it to launch kernels. Unlike Gdev, GDM provides fully functional device memory management that allows the device memory space to be flexibly shared by any regions. This guarantees the successful executions of GPGPU applications on multitasking systems.

Figure 8(a) also shows that GDM can handle device memory contentions more efficiently than the stock Gdev. With Gdev, even though a few combo runs successfully finish their executions, they suffer substantial performance losses. For example, for the combo run of *backprop* and *nn*, the throughput achieved with the stock Gdev is only 70% of that with GDM. Due to the lack of necessary mechanisms and policies to reduce data movement, Gdev cannot support data swapping with low overhead. The optimization techniques in GDM can effectively minimize the overhead. Thus, GDM can improve the performance of these workloads by 20% on average (up to 43%).

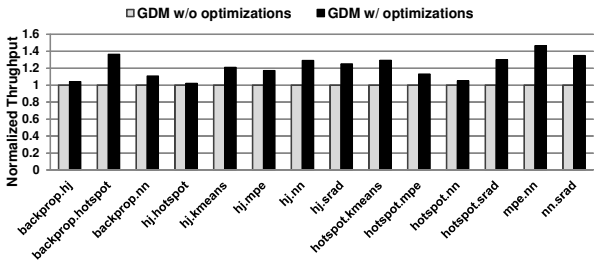
Figure 8(b) compares the performance of GDM and the stock Gdev under the workloads in the second group. For all the workloads except the co-running of *backprop* and *kmeans*, the performance difference between GDM and Gdev is barely observable. When *backprop* co-runs with *kmeans*, the throughput with GDM is slightly lower (by 4%) than that with Gdev. This shows the low overhead of GDM for multitasking workloads without device memory contentions.

6.4 Validation of Design Optimizations

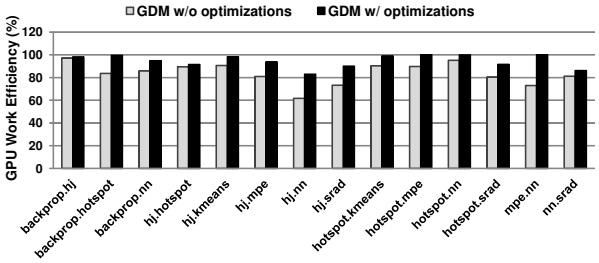
Throughout the design of GDM, optimization techniques are adopted to minimize data transfers and the associated cost. In this subsection, we validate the effectiveness of these optimization techniques through experiments. We compare the performance of the full-fledged GDM with a simplified version of GDM, named GDM-Base, which only provides basic management over device memory to guarantee the correct executions of the combo-run workloads.

Specifically, GDM-Base handles host-to-device data transfer eagerly. It carries out data transfers immediately upon application's requests (e.g. *cuMemcpyHtoD*). When the device memory is short of free space, traditional LRU algorithm is used to select a victim data set to replace. The victim data set is transferred back to the corresponding staging area. When a kernel is to be launched, GDM-Base examines the data sets in the context, and loads the data sets that are not resident in the device memory.

Since the overhead of device memory management is mainly incurred when device memory space is under pressure, we



(a) Normalized throughput with and without GDM optimizations.



(b) GPU work efficiency with and without GDM optimizations.

Figure 9: Effectiveness of GDM optimizations.

select the combo runs in the first group and compare the performance of GDM and GDM-Base under these workloads. As shown in Figure 9(a), with optimizations, GDM is able to consistently improve the throughput of the workloads by 21% on average and up to 46% (relative to GDM-Base).

To further understand how the optimization techniques improve performance, we have collected the *work efficiency* of GPU, which is defined as the percentage of total GPU time spent on kernel executions and *effective* data movement. A data movement is effective if it is carried out when the benchmark runs alone. For example, in a combo-run workload of program A and program B, x repetitions of program A fully overlap with y repetitions of program B. During the co-running of the programs, the total time spent by the GPU to execute kernels and move data is c. If the time used for kernel execution and data movement is a for program A and b for program B when each of program A and B executes alone, the GPU work efficiency for this combo run is $(ax + by)/c$. Work efficiency reflects the amount of overhead incurred by device memory management, with high efficiency indicating low overhead. Though the overhead is mainly from extra data transfers incurred by device memory management, kernel execution time is included in work efficiency measurement because we want to correlate overhead reduction with throughput increase.

Figure 9(b) shows that the optimization techniques improve GPU work efficiency by 14% on average and up to 37%. This explains the throughput improvement observed in Figure 9(a). Meanwhile, it also explains the varying degrees of performance enhancement for different combo runs. For example, the throughput of the co-running of *backprop* and *hj* only rises by 4%. This is because the GPU has already been working at almost full efficiency before and after optimizations are applied (97% vs. 98%), as shown by the first two bars in Figure 9(b).

For most workloads, with the optimization techniques in GDM, the GPU work efficiency is close to 100%. This shows that the optimization techniques work effectively on control-

ling the overhead. However, we notice that there are a few workloads with GPU work efficiency below 90%. This indicates that there is still potential to further improve the performance of GDM in future work.

6.5 Defending against DoS Attacks

GDM makes the system capable of thwarting denial-of-service (DoS) attacks that deplete the device memory space available to GPGPU applications. To demonstrate this capability of GDM, we have designed a malicious program which reserves a device memory region with the same size as the usable device memory capacity. The program repeatedly issues a GPU kernel that updates the data content in the reserved region so as to cause the largest performance degradation to GDM management.

We co-run each of the benchmarks (except *theano*) with the malicious program, and measure its execution time[5]. With GDM, all benchmarks successfully finish executions in spite of the presence of the malicious program. The performance of the benchmarks is lowered by the malicious program compared to their solo executions, but kept at an acceptable level (69% on average). The highest slowdown happens with *srad* (284%) because it launches kernels frequently and each kernel accesses a moderately large working set, causing more data evictions than other benchmarks. The lowest slowdown is observed with *kmeans* (9%) because it has the least demand on device memory space.

7. RELATED WORK

We are not the first to realize the problems caused by having GPU programmers directly and explicitly manage the device memory. Gdev [31] provides a data sharing mechanism for inter-process communication (IPC) and shows that this mechanism can be used to support device memory swapping. However, because it is based on an IPC mechanism and lacks generality, this proof-of-concept workaround barely works in practice and suffers serious performance issues as have been shown with our experiments. RSVM [29] provides an application-level device memory manager in a library. It relieves programmers from explicitly managing device memory, but programs must call the functions it provides to gain the benefits. Meanwhile, it suffers from the problems with application-level management. For example, it cannot address the contention between applications and does not allow an application to use other libraries that call CUDA APIs to allocate device memory or transfer data. Compared with these studies, GDM identifies the critical issues of device memory management at system level and provides a general and non-intrusive solution.

System management of GPGPU resources other than device memory has received attention in several recent studies. Pegasus [26] is a computation scheduling facility for virtualized, accelerator-based multiprocessor systems. It makes GPU a schedulable entity in the hypervisor and supports both high-throughput and low-latency scheduling among multiple guest OSes. TimeGraph [30] is a GPU command scheduler to support fair sharing of GPU computing resource for real-time, multitasking GPU applications. PTask [41] provides an OS abstraction for GPU computing resource and data transfer management. It presents a dataflow programming model that exposes information for OS kernel to provide performance isolation and to coordinate data movement between collaborative processes. GPUfs [42] proposes file system support for GPGPUs to allow a GPU program to access host files directly.

Some research projects in architecture and compiler areas improve the usability of GPUs as main-stream computing devices. iGPU [36] is a GPGPU architecture to support exceptions and speculative executions with compiler support. ADSM [24] is a data-centric programming model for heterogeneous computing that maintains an asymmetric shared memory space to achieve low cost. CGCM [27] is an automatic management and optimization system to reduce programmer's efforts for CPU-GPU data transfer. There are plans to provide unified and shared virtual spaces for CPU and GPU to access [14]. They do not provide or have not provided a solution to manage the physical space in the device memory. Instead, they pose a higher demand for operating system managing the device memory space, which is targeted by the research in this paper.

8. CONCLUSION AND FUTURE WORK

This paper identifies a crucial problem with existing GPGPU system software design. Namely, the lack of sophisticated device memory management causes application crashes, hangs, and inefficient utilization of GPGPU resources. This problem can seriously hinder the adoption of GPGPUs as mainstream computing devices in general-purpose systems.

The paper presents GDM, a fully functional device memory manager, to effectively address the problem. The design fully considers the unique features of GPGPU computing and GPGPU devices from the perspectives of both challenging problems and optimization opportunities. GDM manages device memory with both block and object-level information, and employs various optimization techniques to ensure system performance. Experiments verify the capabilities of GDM to tolerate device memory leaks, prevent program crashes, defend against malicious programs, and achieve high performance.

As future work, we plan to improve the management over GPU device memory following two directions. First, it is possible to further reduce the overhead by leveraging the information from compilers or applications. Through static analysis of kernel source code, the compiler can infer some information that may otherwise need to be obtained with extra cost. In many applications such as databases and in-memory big data engines, similar information can also be easily inferred from application-level semantics. Second, we also plan to investigate the collaboration between device memory manager and GPU kernel scheduler for more optimization opportunities. For example, when there is not enough free space in the device memory for the execution of a selected kernel, the system should balance the benefit of launching the kernel and the potential overhead. It may also decide whether to schedule another kernel with a smaller working set, or to wait for an issued kernel to finish execution and free up the space.

[5]On existing systems, a malicious program can also attack the system by issuing a non-terminating kernel (e.g. an infinite loop) or a large number of kernels. Thus, a thorough solution requires enhancements on GPU kernel scheduling, which is beyond the scope of the paper. This paper only focuses on the attacks through device memory space, and lets the system schedule GPU kernels in a round-robin manner in the experiment.

9. ACKNOWLEDGMENTS

We thank Yihong (Eric) Zhao from Yahoo! Inc. and the anonymous reviewers for their help and feedback. This work was partially supported by the National Science Foundation under grants CCF-0913050, OCI-1147522, and CNS-1162165.

10. REFERENCES

[1] http://mathworks.com/matlabcentral/newsreader/view_thread/324086.

[2] http://milkyway.cs.rpi.edu/milkyway/forum_thread.php?id=2780.

[3] http://culatools.com/blog/2012/03/12/3099.

[4] http://blenderartists.org/forum/showthread.php?269777.

[5] https://github.com/Theano/Theano (commit#: 5a755867f21b9a61, fe69a5a5b3a44695, 410016f9d6025064, 9bdeda96639e77af).

[6] http://mail-archive.com/pycuda@tiker.net/msg02432.html.

[7] http://amd.com/en-us/innovations/software-technologies/apu.

[8] http://documen.tician.de/pycuda/.

[9] https://devtalk.nvidia.com/default/topic/513370/cublas-problem.

[10] http://setiweb.ssl.berkeley.edu/beta/forum_thread.php?id=1441.

[11] http://mathworks.com/matlabcentral/answers/85601-unavoidable-memory-leaks-in-mex.

[12] http://nouveau.freedesktop.org.

[13] https://github.com/serban/kmeans.

[14] http://www.hsafoundation.com/.

[15] CULA linear algebra libraries. culatools.com.

[16] AMD. AMD accelerated parallel processing OpenCL programming guide, 2013.

[17] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In *SciPy*, 2010.

[18] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3d) microarchitecture. In *MICRO*, 2006.

[19] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[20] 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 *IISWC*, 2009.

[21] P. J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, 1970.

[22] P. J. Denning. Third generation computer systems. *ACM Comput. Surv.*, 3(4):175–216, Dec. 1971.

[23] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *HOTOS*, 1995.

[24] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *ASPLOS*, 2010.

[25] K. O. W. Group. The OpenCL specification 1.2, 2013.

[26] V. Gupta, K. Schwan, N. Tolia, V. Talwar, and P. Ranganathan. Pegasus: coordinated scheduling for virtualized accelerator-based systems. In *USENIX ATC*, 2011.

[27] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[28] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: cheap SSL acceleration with commodity processors. In *NSDI*, 2011.

[29] F. Ji, H. Lin, and X. Ma. RSVM: a region-based software virtual memory for GPU. In *PACT*, 2013.

[30] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

[31] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: first-class GPU resource management in the operating system. In *USENIX ATC*, 2012.

[32] H. Kim. Supporting virtual memory in GPGPU without supporting precise exceptions. In *MSPC*, 2012.

[33] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.

[34] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2), 2008.

[35] M. Macedonia. The GPU enters computing's mainstream. *Computer*, 36(10):106–108, 2003.

[36] J. Menon, M. De Kruijf, and K. Sankaralingam. iGPU: exception support and speculative execution on GPUs. In *ISCA*, 2012.

[37] T. Ni. Direct Compute: Bring GPU computing to the mainstream. In *GTC*, 2009.

[38] NVIDIA. NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012.

[39] NVIDIA. NVIDIA CUDA C programming guide, 2013.

[40] J. Poulton. An embedded DRAM for CMOS ASICs. In *ARVLSI*, 1997.

[41] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *SOSP*, 2011.

[42] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating a file system with GPUs. In *ASPLOS*, 2013.

[43] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

[44] K. Wang, Y. Huai, R. Lee, F. Wang, X. Zhang, and J. H. Saltz. Accelerating pathology image data cross-comparison on CPU-GPU hybrid systems. *Proc. VLDB Endow.*, 5(11), 2012.

[45] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.*, 6(10):817–828, 2013.