Karlsruher Institut für Technologie

Transparent Memory Extension for Shared GPUs

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

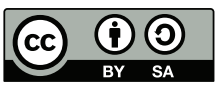
von

Dipl.-Inform. Jens Kehne

aus Frankfurt am Main

Tag der mündlichen Prüfung: 07.02.2019

Hauptreferent:	Prof. Dr. Frank Bellosa Karlsruher Institut für Technologie
Koreferent:	Prof. Dr. Jörg Nolte BTU Cottbus



Abstract

Over the last few years, graphics processing units (GPUs) have become popular in computing. Consequently, all major cloud providers have included GPUs in their platforms. These platforms typically use virtualization to share physical resources between users, which increases the utilization of these resources. Utilization can be increased even further through oversubscription: Since users tend to buy more resources than are actually needed, providers can offer more resources than physically available to their customers, hoping that the customers will not fully utilize the resources that were promised all the time. In case customers do fully utilize their resources, however, the provider must be prepared to keep the customers' applications running even if the customers' resource demands exceed the capacity of the physical resources.

The memory of modern GPUs can be oversubscribed easily since these GPUs support virtual memory not unlike that found in CPUs. Cloud providers can thus grant large virtual address spaces to their customers, only allocating physical memory if a customer actually uses that memory. Shortages of GPU memory can be mitigated by evicting data from GPU memory into the system's main memory. However, evicting data from the GPU is complicated by the asynchronous nature of today's GPUs: Users can submit kernels directly into the command queues of these GPUs, with the GPU handling scheduling and dispatching autonomously. In addition, GPUs assume that all data allocated in GPU memory is accessible at any time, forcefully terminating any GPU kernel that tries to access unavailable data.

Previous work typically circumvented this problem by introducing a software scheduler for GPU kernels which selects the next kernel to execute in software whenever a previous kernel finishes execution. If data from the next kernel's address space has been evicted, the scheduler returns that data to GPU memory before launching the next kernel, evicting data from other applications in the process. The main disadvantage of this approach is that scheduling GPU kernels in software bypasses the GPU's own, highly efficient scheduling and context

switching and therefore induces significant overhead in applications even in the absence of memory pressure.

In this thesis, we present GPUswap, a novel approach to oversubscription of GPU memory which does not require software scheduling of GPU kernels. In contrast to previous work, GPUswap evicts data on memory allocation requests instead of kernel launches: When an application attempts to allocate memory, but there is insufficient GPU memory available, GPUswap evicts data from the GPU into the system's main memory to make room for the allocation request. GPUswap then uses the GPU's virtual memory to map the evicted data directly into the address space of the application owning the data. Since evicted data is thus directly accessible to the application at any time, GPUswap can allow applications to submit kernels directly to the GPU without the need for software scheduling. Consequently, GPUswap does not induce any overhead as long as sufficient GPU memory is available. In addition, GPUswap eliminates unnecessary copying of data: Only evicted data that is actually accessed by a GPU kernel is transferred over the PCIe bus, while previous work indiscriminately copied all data a kernel might access prior to kernel launch. Overall, GPUswap thus delivers consistently higher performance than previous work, regardless of whether or not a sufficient amount of GPU memory is available.

Since accessing evicted data over the PCIe bus nonetheless induces non-trivial overhead, GPUswap should ideally evict rarely-accessed pages first. However, the hardware features commonly used to identify such rarely-accessed pages on the CPU – such as reference bits – are not available in current GPUs. Therefore, we rely on off-line profiling to identify such rarely-accessed pages. In contrast to previous work on GPU memory profiling, which was based on compiler modification, our own profiling uses the GPU's performance monitoring counters to profile the application's GPU kernels transparently. Our profiler is therefore not limited to specific types of application and does not require recompiling of third-party code such as shared libraries. Experiments with our profiler have shown that the number of accesses per page varies mostly between an application's memory buffers, while pages within the same buffer tend to exhibit a similar number of accesses.

Based on the results of our profiling, we examine several possible eviction policies and their viability on current GPUs. We then design a prototype policy which allows application developers to assign a priority to each buffer allocated by the application. Based on these priorities, our policy subsequently decides which buffer's contents to evict first. Our policy does not require hardware features not present in current GPUs, and our evaluation shows that the policy is able to relocate significant amounts of application data to system RAM with minimal overhead.

Contents

Chapter 1

Introduction

Graphics processing units (GPUs) have become increasingly popular in computing. The freely-programmable nature of modern GPUs combined with their unprecedented levels of performance and low power consumption make these GPUs a perfect fit for applications like machine learning [1], computer vision [17], cryptography [32], network packet processing [22] and even processing of general web requests [5]. However, despite their obvious benefits, GPU adoption is often hindered by hardware costs: The price for a high-end GPU model can reach five-digit numbers[1].

An increasingly popular alternative to buying dedicated hardware is to integrate these high-end GPUs into shared environments, such as clouds. Sharing a GPU in this manner allows the cloud provider to increase the GPU's utilization – and thus offer GPU computation time at a lower price – for two reasons: First, applications tend to have a finite execution time. After an application finishes, a customer with exclusive access to a GPU may take a while to start the next application, which forces the GPU to idle. In contrast, a shared GPU can execute multiple applications concurrently and thus does not fall idle if a single application exits. Second, running GPU applications may not fully utilize the GPU's resources: Most applications perform I/O or CPU computation from time to time, during which a dedicated GPU would idle. In contrast, a shared GPU can simply execute code from another application during these periods, which increases utilization. Furthermore, GPU applications may not launch a sufficient number of threads to fully utilize all of the GPU's cores, thus leaving some of these cores unused. In that situation, some GPUs have the ability to execute threads from more than one application in parallel to increase utilization.

Integrating a GPU into a cloud environment, however, requires the cloud provider to efficiently virtualize the GPU, which creates interesting new challenges. GPUs

[1] At the time of this writing, the prices for an Nvidia Tesla V100 in German web shops ranged from €8,780.66 to €18,194.99.

were designed to be used exclusively by a single application, and modern-day GPUs inherited the basic design principles of their ancestors. As a result, GPUs are difficult to share between applications – especially if these applications do not cooperate with each other, which is the common case in a cloud environment. While some support for sharing – such as GPU virtual address spaces – has been added in recent years, hardware support for GPU virtualization is still in its infancy.

Virtualizing a GPU in software has been a major focus of research in recent years. These efforts, however, have largely focused on the efficiency of virtualization [18, 21, 56, 62] – i.e., on reducing the virtualization overhead – or on expanding the functionality of virtual GPUs [13, 58, 61, 64, 67] – e.g., on enabling full virtualization. More recently, some research efforts have also considered the problem of achieving fairness between multiple, mutually untrusted GPU applications [38, 44, 54, 58, 61].

A topic that has been mostly overlooked in previous research, however, is how to deal with the memory of a virtualized GPU. As for the GPU's computational power, it is desirable for a cloud provider to oversubscribe that memory to increase its utilization. While this oversubscription can be achieved simply by promising customers a larger amount of memory than is actually available, doing so can easily lead to a shortage of GPU memory if customers actually do allocate all the memory promised to them. On the CPU, this situation is typically handled by swapping data to disk; however, this technique does not apply to current GPUs since these GPUs lack important features – most notably page fault support – that are necessary to implement classical swapping.

1.1 Extending the GPU Memory

One approach to alleviating shortages of GPU memory is to use system RAM in its place. Current GPUs typically have the ability to access system RAM directly. This access is transparent to the applications running on the GPU: The GPU driver can map system RAM into the GPU address space of the application, which the application can then read and write as usual. When the available GPU memory is running low, the operating system can thus prevent applications from failing by transparently allocating system RAM instead of GPU memory. However, each access to system RAM translates to a transaction on the PCIe-bus. Accessing system RAM is therefore significantly slower than accessing GPU RAM: The GDDR5X memory of current GPUs reaches transfer rates of up to 448 GiB/s and latencies of about 10 ns [33], whereas the PCIe-bus limits transfers to and from system RAM to a bandwidth of 16 GiB/s and a latency of several hundred nanoseconds [42]. Therefore, any application using system RAM instead of GPU memory will suffer

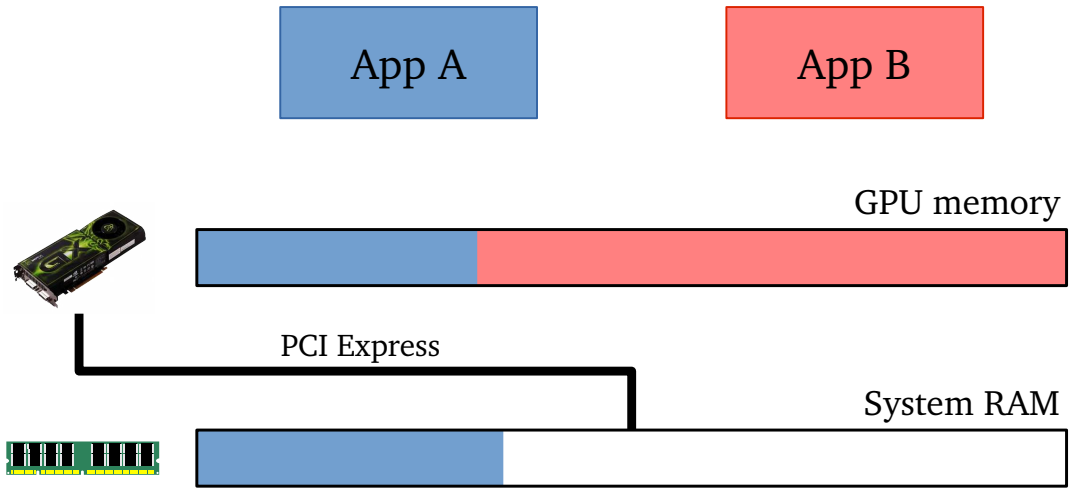


Figure 1.1: Unfairness with first-come-first-served-allocation. Both applications have allocated the same amount of memory. However, application B receives a larger amount of GPU memory simply because it allocated its memory first.

a severe performance hit. However, this performance degradation is arguably preferable to applications failing altogether due to lack of memory.

While a performance penalty is unavoidable when extending GPU memory with system RAM, it is the operating system's task to at least minimize that performance penalty. Specifically, this leads to two main goals for such a system:

1. **Utilization:** The system should maintain high utilization of the available GPU memory. Specifically, applications should not have to use system RAM until all GPU memory is exhausted.

2. **Fairness:** If using system RAM is unavoidable, applications should still benefit equally from the GPU memory available. Thus, each application should be guaranteed an equal share of the available GPU memory.

The allocation strategies for GPU memory found in most current research projects on GPU virtualization typically fall short on at least one of these goals. In fact, current research largely uses rather simple allocation strategies like first-come-first-served (FCFS) or static partitioning. In the following, we will examine these allocation strategies in more detail.

First-Come-First-Served Allocation

FCFS allocation is probably the simplest strategy imaginable: Allocation requests are fulfilled immediately on arrival, without considering any other requests, past or future. Requests are fulfilled from GPU memory when sufficient GPU memory

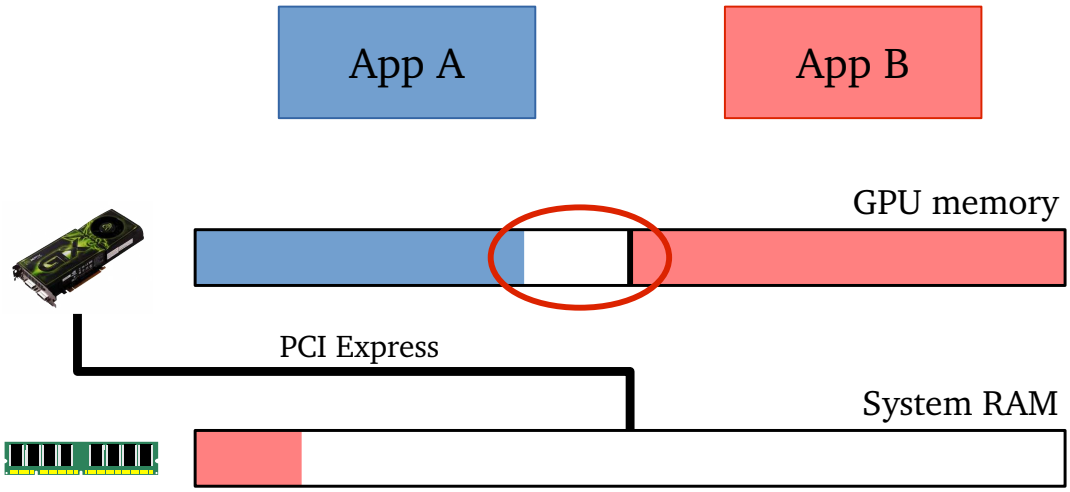


Figure 1.2: Poor utilization with static partitioning. Some GPU memory is reserved for application A. Therefore, application B is forced to use slower system RAM, even though there would be enough free GPU memory available.

is available; if not, system RAM is used instead. This strategy does maintain good utilization – system RAM is never used while GPU memory is available. However, the fact that FCFS only considers requests individually easily leads to unfairness, as illustrated in Figure 1.1: Two applications, A and B, both allocated the same amount of memory. However, application B allocated its memory first, and therefore received only GPU memory, which was abundantly available at the time. When application A allocated the same amount of memory later on, system RAM was used for part of the allocation request because there was not enough GPU memory available. The result is an unfair distribution of GPU memory: Application A must keep some of its data in system RAM and thus suffers performance degradation, whereas application B can keep all its data in the much faster GPU memory.

Static Partitioning

A seemingly obvious solution for the unfairness problem of FCFS is to statically partition the available GPU memory between applications: For n applications, $1/n$ of the available memory is reserved for each application exclusively. While this scheme achieves good fairness – each application is guaranteed to receive the same share of GPU memory – it also requires the number of applications to be known beforehand. In addition, this scheme can lead to poor utilization if an application does not actually use all of its share, as illustrated in Figure 1.2: Two applications are using the GPU concurrently, with half of the GPU's memory reserved for each application. However, application A uses slightly less memory

than it would be entitled to, while application B uses slightly more. As a result, application B is forced to use system RAM – and thus suffers from degraded performance – even though there is still unused GPU memory available.

Swapping

Besides FCFS and static partitioning, there are some recent research projects taking more sophisticated approaches. Most notably, Gdev [39] and its extension GDM [63] include a swapping mechanism integrated with a software scheduler for GPU kernels: The scheduler decides in software which GPU kernel to launch next, and the swapping mechanism subsequently moves all data that this kernel might need to the GPU – potentially evicting data from other applications in the process – before the kernel is actually launched. While this approach is successful in enabling oversubscription of GPU memory, it also suffers from two fundamental drawbacks: First, scheduling GPU kernels in software can induce considerable overhead in GPU applications even in the absence of memory pressure since it implicitly disables the GPU's internal, highly efficient scheduling and context switching [38]. Second, copying all data a GPU kernel might need prior to kernel launch may lead to unnecessary data transfer if the kernel does not actually need all the copied data.

1.2 Contributions

The goal of the work presented in this thesis is to extend GPU memory with system RAM with minimal overhead while maintaining both fairness and high utilization of GPU memory. Specifically, this thesis makes the following contributions:

Memory extension mechanism for GPUs We present an extension mechanism for GPU memory, called GPUswap, which transparently extends GPU memory with system RAM without relying on software scheduling of GPU kernels. GPUswap evicts application data from the GPU to system RAM whenever an allocation request cannot be satisfied due to insufficient GPU memory. In contrast to previous work, GPUswap's operation is triggered by allocation requests rather than GPU kernel launches, which has two main advantages: First, GPUswap does not add any overhead to GPU kernel launches. Therefore, GPUswap's overhead is virtually zero in the absence of actual memory pressure. Second, GPUswap keeps the evicted data directly accessible to the application. Therefore, during GPU computation, only data that is actually touched by the application is transferred over the PCIe bus.

Profiling mechanism for GPU memory accesses The eviction policy accompanying GPUswap requires information about the applications' memory accesses to operate. Since this information cannot be collected at runtime on current GPUs due to lack of hardware support, we instead develop a memory profiling mechanism which is based on the GPU's performance monitoring counters and GPUswap. Our mechanism offers two main advantages over previous work: First, our mechanism does not assume a specific type of application, but can instead count memory accesses from arbitrary applications. Second, our mechanism is able to gather information about memory regions outside the application's direct control, such as those allocated by the GPU runtime library.

Using our mechanism, we observe that page-level eviction, as done on the CPU, is often not necessary on the GPU since GPU applications display a higher degree of uniformity in their memory accesses due to their data-parallel nature.

Eviction policy We present a proof-of-concept policy based on hints generated from profiling of GPU applications to demonstrate the general benefit of an eviction policy for GPUs. In addition, we discuss other possible eviction policies for GPUswap and their applicability to current GPUs. Some of these policies are not actually viable on current GPUs since these GPUs lack many hardware features commonly found in CPUs, such as page faults or reference bits. However, as GPUs are growing ever closer to CPUs in terms of features [59], one of these strategies may well become state of the art in the future. Therefore, we also discuss how the hardware of current GPUs would have to change to enable more efficient eviction policies for GPUs.

Performance considerations Finally, we evaluate the overhead induced by both GPUswap and the use of system RAM in general. In case GPU memory is scarce, we analyze the overhead induced by GPUswap's evictions. Using GPUswap, there are two main sources of overhead: i) Copying data between CPU and GPU in response to memory pressure, and ii) frequent accesses to system RAM after data has been swapped out. We quantify both types of overhead, and assess to what extent these overheads can be alleviated by an eviction policy.

1.3 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 – Background and Literature Review describes the hardware of current GPUs, and how the design of that hardware affects our goal of extending GPU memory with system RAM. In addition, we review related work on GPU virtualization and GPU memory management in this chapter.

Chapter 3 – An Eviction Mechanism for GPUs introduces GPUswap, our novel memory extension mechanism for GPUs, which is the main contribution of this thesis. We describe both GPUswap's design and implementation in this chapter.

Chapter 4 – Profiling Memory Access Patterns of GPU Applications presents our method for profiling the memory access patterns of GPU applications. We first describe our methodology for measuring GPU memory accesses of GPU applications, which is based on GPUswap. Then, we describe the memory access patterns we observed in several GPU applications, and the implications of these patterns for eviction policies.

Chapter 5 – Potential Eviction Policies for GPUs discusses possible eviction strategies and their viability on current GPUs. We also assess how the hardware of current GPUs should be extended for better memory management – i.e., what additional features GPUs would need to enable more efficient eviction decisions. Finally, we present a proof-of-concept policy which works on GPUs that are in use today.

Chapter 6 – Performance Evaluation of GPU Memory Extension quantifies the overhead that eviction has on GPU applications. In this chapter, we also use our proof-of-concept policy to assess to what extent the overhead associated with using system RAM in place of GPU memory can be alleviated through an eviction policy. Finally, we show that GPUswap induces no overhead unless actual memory pressure is present.

Chapter 7 – Conclusion summarizes the main points of the thesis, as well as the contributions and limitations of the presented work. Finally, we also discuss possible future research directions in this chapter.

Chapter 2

Background and Literature Review

Modern GPUs support paged virtual memory similar to that used on the CPU: Applications are confined to virtual address spaces, and virtual addresses from these spaces are translated to physical addresses by a dedicated MMU via a page table. However, current GPUs are trailing behind CPUs in terms of features: For example, GPUs are only starting to support page faults, and the page tables of current GPUs do not include reference or dirty bits. Therefore, well-known techniques for memory management on the CPU are typically not applicable to GPUs.

2.1 Virtual Memory Systems

In the first computer systems, programs were loaded directly into physical memory. With the advent of multiprogramming, however, this approach turned out to be insufficient: Since each physical address can be used by only one program at a time, programmers had to manually ensure that each program – and even multiple concurrent instances of the same program – used different parts of the available memory, which proved to be a cumbersome and error-prone task.

2.1.1 Virtual Address Spaces

As many other problems in computer science, the problem of multiplexing memory addresses among applications was eventually solved by adding another level of indirection. On today's computers, programs operate exclusively on **virtual addresses**. Whenever a program accesses a virtual address, a dedicated coprocessor, called the **memory management unit (MMU)**, transparently translates this virtual address into a physical address using a translation table, as shown in Figure 2.1. Since each application has its own translation table, different

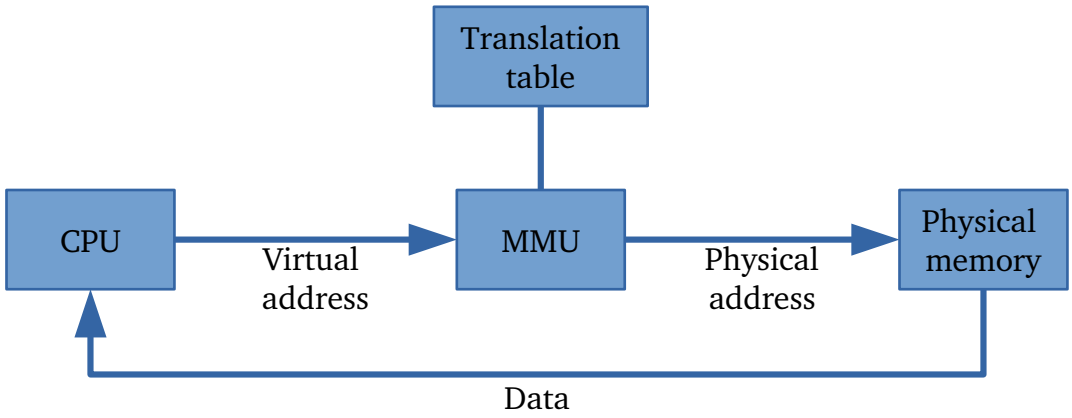


Figure 2.1: Address translation on current CPUs. Programs running on the CPU access virtual addresses, which the MMU translates into physical addresses using a translation table.

applications are free to use the same virtual addresses. It is the operating system's task to configure each application's translation table appropriately so that each application's data actually resides in a different physical location.

The set of all virtual addresses available to a program forms the program's **virtual address space**. This virtual address space is not conceptually different from the physical address space; its addresses can be used in machine instructions directly just like physical addresses, which any necessary translation being performed transparently by the MMU. However, since the virtual address space is an abstraction of the physical memory, it provides three key benefits over accessing physical memory directly:

1. Applications are given the illusion of owning the entire physical memory. Since each application has a dedicated virtual address space, each application is free to access any virtual address, without any danger of overwriting data other than its own.

2. Applications are protected from each other. Since each memory access has to go through the MMU and translation table, applications are unable to access physical memory for which no translation exists in their virtual address space. As a result, the operating system can guarantee that applications cannot access each other's data by ensuring that each physical address is mapped in at most one address space. Note that it is also possible to relax this protection by establishing mappings to the same physical address in multiple address spaces, but this is typically done only if explicitly requested by the application.

3. Applications need not care about the amount of physical memory available. Instead, they are free to use their entire virtual address space as they see fit; it is the operating system's task to provide appropriate translations

into physical memory. If the applications' demand for memory exceeds the capacity of the physical memory, the operating system must decide how to respond to that condition, for example by terminating one or more applications to free physical memory for use by other applications.

The virtual address space is typically larger than the physical address space. As a consequence, not every virtual address has a corresponding translation to a physical address. Normally, addresses without a translation are not a problem since applications typically do not use their entire virtual address space. However, the operating system must be able to handle accesses to virtual addresses without a translation – typically either by allocating memory at the address that was accessed and restarting the faulting instruction (lazy allocation) or by terminating the program if the operating system determines that the memory access was the result of faulty or malicious application behavior.

2.1.2 Paging

As address spaces can be quite large on current computer systems, providing virtual-to-physical translations for each individual address is infeasible. As a result, paged virtual memory [15] was introduced. With paged virtual memory, the virtual address space is composed of **pages**, which are contiguous and indivisible regions of memory. The size of a page ranges from a few kilobytes to several gigabytes, with the starting address of each page being aligned to the page's size. For example, the x86-64 CPUs in use today typically offer a page size of 4 KiB, but also support **huge pages** of 2 MiB or 1 GiB [2]. Other architectures, like PowerPC, support page sizes of up to 16 GiB [30].

With paged virtual memory, the translation table – which is called the **page table** in this context – specifies a translation for each page. To translate a virtual into a physical address, the MMU uses the page table to map the highest bits of the virtual address to the starting address of a physical page. The remainder of the virtual address is then added to the physical page's starting address to obtain the address of the specific byte to be read or written.

Even though page tables must only store one entry per page, the entire table can still be prohibitively large for large virtual address spaces. For example, a complete page table for a virtual and physical address space of 64 bit and a page size of 4 KiB would be 32 PiB in size. As a result, various schemes have been developed to reduce the size of the page table in memory [31]. On x86 CPUs, for example, page tables consist of multiple levels: The first level, called the **page directory**, contains addresses to second-level page tables, which in turn may hold physical page addresses or addresses of yet another level of page tables. This scheme can be extended to an arbitrary nesting depth – today, three to four levels are commonplace – with only the last level of page tables holding physical page

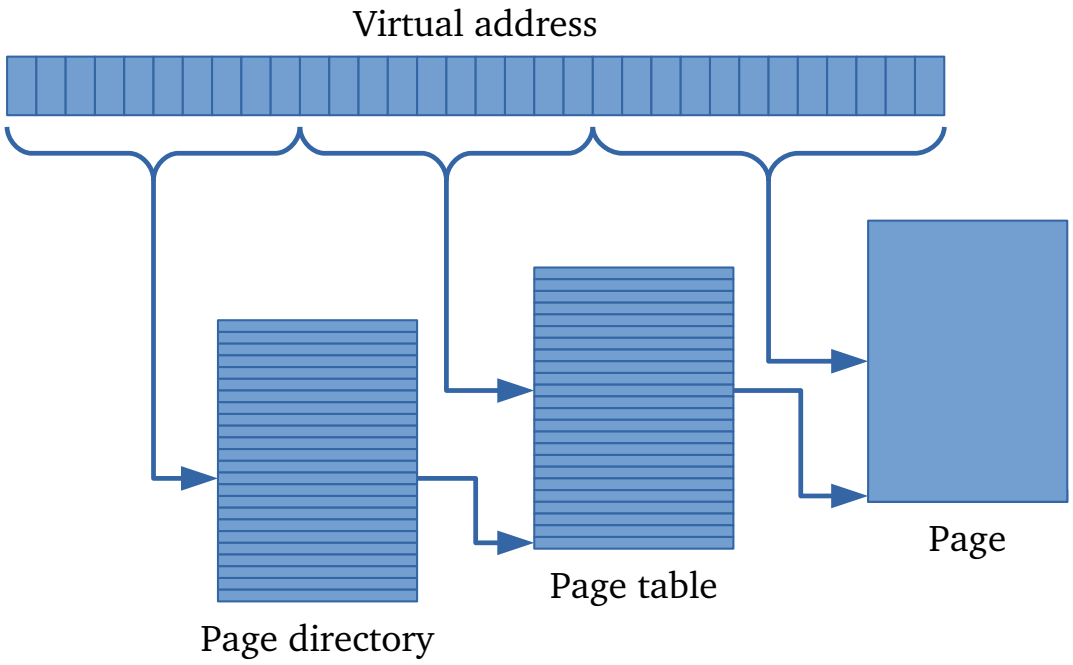


Figure 2.2: Virtual address translation for a two-level page table and 32-bit virtual addresses, as found in an x86 CPU without physical address extension. The first 10 bits of the virtual address serve as an index into the page directory, yielding the address of the second level page table. The second 10 bits of the virtual address are then used as an index into the second-level page table, which yields the starting address of the physical page. The remaining 12 bits of the virtual address then serve as an index into the physical page.

addresses. The advantage of this design is that only page tables actually in use must be held in memory. Other hardware architectures use different designs, such as **inverted page tables** holding physical-to-virtual address translations, thus scaling with the size of the physical memory rather than that of the virtual address space, **hashed page tables** using a hash of the virtual address to index the table or **software-walked page tables** allowing the operating system to define an arbitrary page table structure. To reduce the cost of resolving virtual-to-physical translations, most architectures also feature a **translation lookaside buffer (TLB)** which caches frequently-used translations.

Figure 2.2 shows the address translation process for a two-level page table and 32 bit addresses. In this example, the MMU uses the first 10 bits of the virtual address as an index into the page directory. The page directory entry at this index holds the starting address of a second-level page table. The next 10 bits of the virtual address are then used as an index into that second-level page table, with the referenced entry holding the starting address of a physical page. The

Figure 2.3: Page table entry on 32 bit x86

remaining 12 bits of the virtual address then serve as an index into the physical page, referencing a specific byte to be read or written.

Besides a physical address, a page table entry holds various other information about the physical page it references. As an example, Figure 2.3 shows the format used on 32 bit x86 for both page table and page directory entries. Since pages are 4 KiB in size with each page's starting address aligned to its size, each entry only needs to hold the upper 20 bits of the physical page- or page table address, which leaves 12 bits for additional information. These bits are used for various flags describing the page referenced by the entry:

- **Present (P)** indicates whether this page table entry is currently valid. If this bit is zero, the MMU ignores the remainder of the entry.
- **Read/Write (R)** determines whether the page can be written to. Somewhat counterintuitively, a value of one prevents the page from being written.
- **User (U)** controls if the page is accessible to unprivileged code. If this bit is not set, only the operating system kernel may access the page.
- **Write-through (W)** controls the cache's behavior for this page. If the bit is set, write-through caching is used for the page. Otherwise, write-back caching is used.
- **Cache (C)** disables caching for this page altogether if set.
- **Accessed (A)** is automatically set by the MMU if the page is read from or written to. This bit is also known as the **reference bit**.
- **Dirty (D)** behaves similar to accessed, but is only set when the page is written to.
- **Size (S)** is only present in page directory entries. If set, it indicates that this page directory entry does not hold the address of a second-level page table, but of a physical page of larger size (often called a huge page). For last-level page tables, this bit is always zero.
- **Global (G)** indicates that this entry should not be removed from the TLB when the TLB is flushed.

When the MMU encounters an entry with the present bit set to zero while translating an address, raises an exception. This exception is called a **page fault**. Page faults are processed by the operating system, which usually handles them in one of two ways: If the faulting address should be valid, the operating system typically responds by making the accessed memory available to the application and then retrying the faulting instruction. This technique is often used in conjunction with

large application buffers to allocate only those parts of the buffer that are actually accessed – a technique called **demand paging**. If, however, the application is not supposed to access the faulting address, the operating system typically reacts by terminating the application since an access to an invalid address is typically caused by either a programming error or malicious application behavior.

2.1.3 Virtual Memory and DMA

Today's computers often include devices which transfer large amounts of data to and from the system's main memory, such as hard disks or network controllers. To speed up these large transfers, these devices typically support **direct memory access (DMA)**, which allows them to access data in the system's main memory autonomously, leaving the CPU free to perform other work. Traditionally, these DMA operations used to operate on physical memory addresses, which without remedy allowed each DMA-enabled device to access any data in memory, including that of other applications or the kernel.

To mitigate this issue, current CPUs include an **input/output memory management unit (IOMMU)** which applies virtual memory to DMA operations. DMA operations now target **bus addresses** instead of physical addresses, and the IOMMU translates these bus addresses to physical addresses using a set of per-device page tables maintained by the operating system. If a device accesses a bus address for which no page table entry exists, the DMA operation is typically aborted. By setting up the IOMMU's page tables appropriately, the operating system can thus prevent the device from accessing data it is not supposed to access without verifying the target address of each individual DMA request.

More recently, the IOMMUs in both Intel [29] and AMD [3] CPUs have begun to support direct access to I/O devices from user space. To allow user space applications to perform DMA in a safe way, these IOMMUs translate bus addresses to physical addresses in two steps. For **first-level translation**, which is optional, devices can attach an address space identifier to their DMA requests. The IOMMU uses this identifier to select one of several page tables attached to the device, and subsequently uses this page table to translate the bus address to an intermediate address. The page tables used in first-level translation conveniently share the same format as the regular MMU's page table. If the device tags each DMA request with an identifier for the application the request originated from, the IOMMU can thus use the regular MMU's page tables to allow each application to perform DMA using its own virtual addresses. If a device does not support address space identifiers, the IOMMU skips first-level translation altogether.

Once first-level translation is complete, **second-level translation** translates the intermediate address – or the bus address if first-level translation is skipped – to a physical address using another, dedicated page table global to the device.

This translation is transparent to the device, and is unconditionally applied to all DMA requests. Second-level translation is particularly useful in virtualized contexts: The hypervisor can use second-level translation to control which memory is accessible to an entire virtual machine, and subsequently allow that virtual machine to manage first-level translation itself.

2.1.4 Swapping

Since virtual address spaces make applications oblivious to the amount of physical memory available, it is possible for applications to allocate more memory than available. Modern operating systems handle this situation through **swapping**: If a new memory allocation request cannot otherwise be satisfied, the operating system pushes some application data out to secondary storage. To that end, the operating system copies a set of pages – not necessarily from the allocating application's address space – to another storage device – typically a hard disk or SSD. Once copying is complete, the operating system clears the present bit in the page table entries pointing to the swapped pages, and stores the location of each page in secondary storage – e.g., a block number on the hard disk – in the remainder of the page's page table entry. Once that process is complete, the operating system can use the swapped physical pages to fulfill the outstanding allocation request.

Since data that has been swapped to secondary storage is not directly accessible to the application, the operating system must be able to return swapped pages to physical memory if the application accesses those pages. When the application accesses a swapped page, the MMU raises a page fault since the present bit of swapped pages is set to zero. The page fault handler then reads the location of the swapped page from the page table entry – the rest of which the MMU conveniently ignores after reading the present bit – copies the content of the swapped page from secondary storage to a physical page, and updates the page table entry to point to the new location of the data in memory. If memory is still contended, the page fault handler may have to swap another page in the process to make room for the page the application tried to access.

When an operating system practices this kind of swapping, finding the right pages to swap to secondary storage is a major problem [60]: Both pushing pages out to secondary storage and getting them back into RAM induces significant latency. Swapping frequently-accessed pages can therefore lead to significant overhead. When memory pressure occurs, the operating system must therefore find rarely-accessed pages to swap to preserve application performance. Current MMUs typically provide tools to aid the operating system in this task – the most common such tool is the reference bit described in Section 2.1.2. Based on these tools, various algorithms are available for selecting appropriate pages [60].

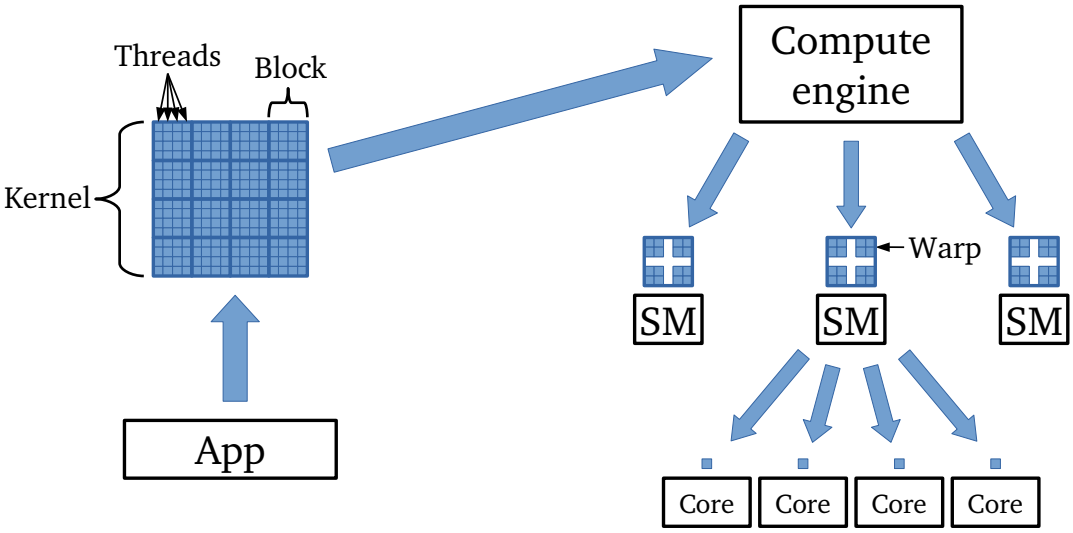


Figure 2.4: The compute model of contemporary GPUs. The application launches kernels consisting of a large number of threads grouped in thread blocks. The kernel is sent to the GPU's compute engine, which schedules each block to run on a streaming multiprocessor (SM) with free capacity. The SMs internally divide each block into warps, and then execute the threads in each warp on their compute cores in a single instruction, multiple thread (SIMT) fashion. Though not shown in the figure for simplicity, SMs may execute multiple blocks concurrently, and the cores interleave between threads of different warps with instruction granularity.

2.2 Graphics Processing Units

GPUs can be used for various purposes other than rendering graphics. In fact, current GPUs are small many-core computer systems capable of executing arbitrary code. Due to their massively-parallel nature, these GPUs can deliver tremendous levels of performance for applications that can be parallelized to a sufficient degree. As a result, the use of GPUs is growing ever more widespread, and GPUs have long found their way into the computers in present-day datacenters, even though these computers typically have no screens attached.

2.2.1 Compute Model

Today's GPUs function as asynchronous accelerators: Applications submit multi-threaded programs called **kernels** to the GPU and are then free to perform other work, while the GPU processes these kernels autonomously. For each kernel, the application can optionally ask to be notified once the kernel has finished execution.

Data transfers between CPU and GPU over the PCIe bus are typically expensive – sometimes even more expensive than the GPU computation itself. For best performance, GPU applications should therefore perform as much work as possible on the GPU between data transfers [47, Section 5.3.1]. As a consequence, applications using the GPU often execute in cycles: The application first copies data to the GPU, then executes one or more GPU kernels, and finally copies the result of the GPU computation back into system RAM.

GPU kernels consist of many **threads** which process the kernel's input in parallel. These threads are light-weight compared to CPU threads, and each GPU thread typically performs much less work than a typical CPU thread. For a typical matrix multiplication, for example, each GPU thread computes just one element of the result matrix. Since GPUs often have hundreds of compute cores, this fine-grained parallelization is necessary to fully utilize the GPU's resources.

Figure 2.4 illustrates the compute model used by current GPUs. Application developers must group the threads of each kernel into **thread blocks**. When the kernel is started, the GPU's compute engine then assigns each of these blocks to one of the GPU's **streaming multiprocessors** (SM). Each thread block is executed by exactly one SM – which allows threads in a block to share the SM's resources – but one SM may execute threads from multiple blocks concurrently. The latest generation of Nvidia GPUs features 80 SMs composed of 64 individual cores each [50].

Internally, the SMs subdivide each block into **warps**. A warp typically consists of 32 threads, but smaller warps may exist if the number of threads in a block is not an exact multiple of the warp size. Threads within a warp use a **single instruction, multiple thread (SIMT)** execution model [43]: In each cycle, all threads in the warp execute the same instruction, though possibly with different parameters. To load data from memory, for example, all threads in the warp must simultaneously execute a load instruction, but each thread may load a word from a different address. Different warps can execute independently on the same SM, and SMs with a sufficient number of cores can execute multiple warps in parallel. An SM-internal **warp scheduler** multiplexes the SM's cores among all active warps with instruction granularity. In principle, the SMs can thus execute an instruction from a different warp in each cycle, which allows the warp scheduler to hide memory latency by switching to a different warp after issuing a load instruction. The scheduler also offers a low-overhead barrier synchronization primitive to allow for coordination between warps.

Using a SIMT model greatly simplifies the SMs' design – for example, threads within a warp can share the same instruction decoder. However, this model can also lead to severe performance degradation if the control flow diverges within a warp: If multiple threads in the same warp execute different sides of a branch, the SM must execute both sides of the branch sequentially, discarding the result of

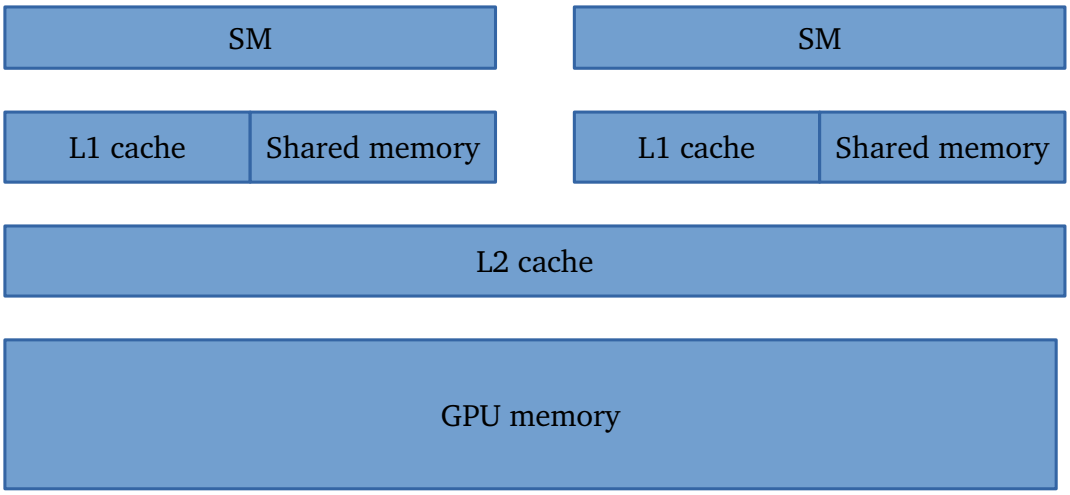


Figure 2.5: The cache hierarchy found in current GPUs. Each streaming multiprocessor (SM) includes its own L1 cache and shared memory, both of which are backed by the same physical memory. The L1 cache is transparent to the application, whereas the shared memory is a scratchpad managed explicitly by the threads running on the SM. In addition, the GPU includes an L2 cache which is shared between all SMs.

one side in each thread. To achieve good performance, developers must therefore carefully structure the code of their GPU applications to fit the GPU's compute model.

Each SM typically includes a large register file, which can be several kilobytes in size. Local variables of GPU threads are thus stored mostly in registers, but can be spilled to a stack in GPU memory if there is insufficient space in the register file. In addition, each SM includes an L1 cache and several kilobytes of **shared memory**. This shared memory serves as a scratchpad shared between all threads in a block which is accessible at the same speed as the L1 cache[1]. Finally, the GPU includes an L2 cache which is shared between all SMs. The entire cache hierarchy is depicted in Figure 2.5.

Nvidia recommends that data local to a thread block should be placed in shared memory whenever possible [47, Section 3.2.3]. Thread blocks therefore often repeat the application's GPU execution cycle: GPU threads copy data from GPU memory into shared memory, perform as much computation as possible on that data, and copy the result of the computation back into GPU memory.

[1] In fact, L1 cache and shared memory are backed by the same physical memory, which is partitioned between the two by the GPU's firmware. The tradeoff can be set at runtime using the CUDA runtime API function `cudaDeviceSetCacheConfig`.

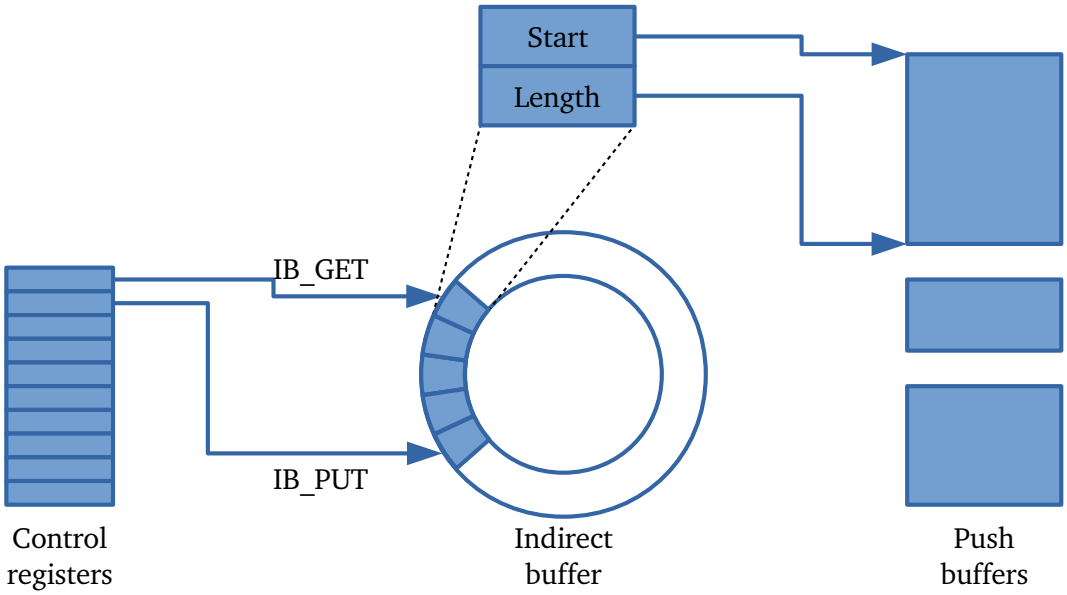


Figure 2.6: A GPU command submission channel. After writing a block of commands into the push buffer, the application places a descriptor item containing the starting address and length of the block in the indirect buffer. Then, the application updates the control register IB_PUT to inform the GPU that new commands have been submitted. IB_GET is updated by the GPU after fetching a block of commands.

2.2.2 Command Submission

Current GPUs communicate with the applications using them through in-memory data structures which Nvidia calls **command submission channels**. Modern Nvidia GPUs support multiple such channels, while some Intel and AMD GPUs are limited to one. Command submission channels are used to submit a stream of high-level commands – like "launch kernel" or "start DMA transfer" – to the GPU for execution. Note that these channels do not contain the actual code of the GPU kernels. Instead, kernel launch commands specify the address of a kernel's code in GPU memory.

The basic structure of a command submission channel is depicted in Figure 2.6. Each channel consists of a ring buffer – dubbed the **indirect buffer (IB)** – and a set of device registers [14]. Inside that register set, two particular registers – called IB_GET and IB_PUT – specify which parts of the indirect buffer currently contain valid entries. Specifically, IB_GET points to the head of the queue – i.e., the end of the queue the GPU *gets* entries from – while IB_PUT points to the tail – i.e., the end where new entries are *put*.

The entries of the indirect buffer are themselves pointers to **push buffers (PB)**, which in turn contain the actual commands. Each IB entry contains the starting

address and length of a single PB, while a PB contains a block of commands that logically belong together. A kernel launch, which actually requires a sequence of multiple commands, is thus represented by a single IB item. Although PBs can be stored at arbitrary locations in GPU memory, GPU drivers typically implement the PBs as another ring buffer to facilitate memory management.

The command submission channels found in Nvidia's GPUs can be mapped directly into the CPU address space of an application. Since these GPUs support multiple channels, each application can be given a dedicated channel. The application can then submit commands to the GPU by writing these commands into the push buffer, appending an item referencing these commands to the end of the indirect buffer, and then setting IB_PUT to point to the newly added IB item. IB_PUT, being a device register, doubles as a doorbell, informing the GPU that a new set of commands has been submitted. The main advantage of mapping channels directly into the application's address space is that applications need not enter the kernel to submit commands to the GPU. This approach therefore reduces the overhead of command submission considerably. At the time of this writing, Nvidia's binary GPU driver and the open-source PathScale Nvidia Graphics Driver (pscnv) [53] implement command submission this way, whereas the Nouveau driver [65] still requires applications to call into the kernel to submit commands.

2.2.3 Command Processing

The GPU includes several **engines** which process the commands submitted by the applications [14]. Three of these engines are relevant in the context of this thesis: **PFIFO** which implements the GPU's internal scheduling and context switching, **PCOPY** which handles asynchronous DMA, and **PGRAPH** which can execute arbitrary code and thus handles all CUDA kernels. Besides these three engines, the GPU also includes a number of engines for specialized tasks, such as video encoding and decoding. Since these engines are not relevant in the context of this thesis, we omit detailed descriptions here for brevity. In general, applications can choose which engine should execute a given command – however, not all commands can be executed on each engine, and submitting a command to an incompatible engine results in an error.

The PFIFO engine implements the GPU's internal scheduler and is thus the first to process every new command. In essence, PFIFO executes a loop consisting of four steps: i) fetch a single entry from one of the GPU's indirect buffers, ii) read the corresponding commands in the push buffer, iii) forward these commands to one of the other engines for processing, and iv) switch to the next indirect buffer. Since a set of commands referenced by a single IB entry typically represents a high-level command from the application – e.g., a GPU kernel launch – PFIFO thus implements simple round-robin scheduling of GPU kernels. The main advantage of this scheme is that the scheduling latency is completely hidden: Fetching

commands and switching to the next channel takes place while the previous kernel is still executing in a different engine.

PFIFO typically forwards CUDA kernels to PGRAPH for processing. PGRAPH is a general-purpose engine capable of executing arbitrary code on the GPU's SMs, and thus the only engine capable of executing CUDA kernels. However, PGRAPH is also capable of performing other tasks, such as DMA. This feature is often used when synchronous DMA is desired: Since each engine can only execute one command at a time, executing a DMA operation on PGRAPH ensures that the next kernel does not start before the DMA operation completes. For asynchronous DMA, a dedicated engine named (PCOPY) is available: Since different engines can execute commands in parallel, PCOPY allows the GPU to perform a DMA operation while a GPU kernel is running.

Since the GPU's engines operate asynchronously to the CPU, GPU commands are typically processed without any interaction with the application that submitted the commands. In some cases, however, the application may need to know whether one of its GPU kernels has finished execution. Polling IB_GET is insufficient in that case since PFIFO advances IB_GET immediately after fetching the last command in a block, and thus before this commands has been processed by one of the other engines. Instead, the GPU offers a special **fence command** which the application can submit to one of its command submission channels. This fence command takes a memory address and a value as parameters, prompting the engine executing the command – which can be any engine – to write the value to the memory address. Since each engine processes only one command at a time and commands from the same command submission channel are processed in order, execution of a fence command guarantees that all preceding commands from the same channel and targeting the same engine have finished execution. The application can thus wait for a command to complete by writing a fence command to one of its channels and subsequently polling the memory location passed to the command. In case polling is undesirable, the application can optionally pass a third parameter to the fence command, prompting the GPU to raise an interrupt upon executing the command. The GPU driver typically offers a corresponding system call which blocks until this interrupt is received. In case multiple fence commands are outstanding, the application can then read the memory addresses passed to all of its outstanding fence commands to determine which one was executed.

2.2.4 Virtual Memory

Modern GPUs support virtual memory similar to that found on the CPU. Each application using the GPU is confined to a dedicated virtual address space on the GPU. This confinement is implemented by attaching the application's command submission channels to the application's address space at creation time. GPU kernels launched via a given channel can then only access memory from the

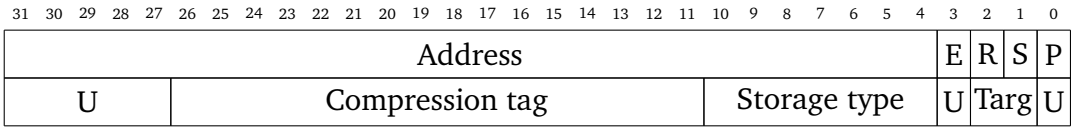


Figure 2.7: Page table entry of an Nvidia Fermi GPU

address space attached to that channel. The GPU's virtual address spaces are essential for allowing applications to submit GPU kernels without operating system intervention: Once address spaces are set up, there is no need for the operating system to examine each command submitted to the GPU since the GPU can autonomously ensure that applications cannot access each other's memory.

As on the CPU, the GPU's address spaces are composed of pages. Current Nvidia GPUs support two distinct page sizes: **Small pages** of 4 KiB and **large pages** of 128 KiB [14]. Whenever a GPU kernel accesses a virtual memory address, a dedicated MMU translates that virtual address to a physical address using a page table set up by the GPU driver. The process of translating virtual to physical addresses is similar to that described in Section 2.1.2.

The page table of a recent Nvidia GPU consists of two levels. The top level consists of a page directory which is 64 KiB in size, stored in 16 contiguous pages in physical GPU memory. Each entry in this page directory is 64 bits long and holds pointers to two distinct second-level page tables: A **small page table** which contains translations for small pages, and a **large page table** containing translations for large pages. Each page directory entry contains pointers to both of these page tables, allowing the driver to mix small and large pages within the memory region covered by a single page directory entry. The driver can also mark either page table as not present if only one page size is required within a page directory entry's memory region, or both if a region is not in use at all. A given virtual address is considered valid if either second-level page table is present and contains a valid mapping for that address; if both page tables contain a valid mapping, the large page table takes precedence.

Each of the GPU's second-level page tables covers 128 MiB of virtual address space, holding a physical address for each virtual page within that region. Since the small and large page tables use different page sizes, the page tables themselves are different in size to cover the same region of memory: Each large page table is 8 KiB long, while each small page table occupies 256 KiB. Like the page directory, both types of page table are stored in contiguous physical pages.

Figure 2.7 shows the entry format used by an Nvidia Fermi GPU for both the large and the small page table. The most important field in each page table entry is the address field, which holds the upper 28 bits of a physical address; for the large page table, the lower 5 bits of this field are always zero. During address translation, the MMU replaces the upper bits of a translated virtual address with the contents of this address field while leaving the lower 12 bits unchanged to

obtain the corresponding physical address. Note that both virtual and physical address space of current GPUs are only 40 bits wide.

The field labeled *Targ* is particularly relevant in the context of this thesis. This field controls which memory – GPU memory or system RAM – the physical address in the address field is located in. Specifically, three values are possible for this field:

- **VRAM** indicates that the physical address is located in the GPU's internal memory.
- **SYSRAM_NO_SNOOP** indicates that the physical address is located in system RAM. Whenever an address corresponding to this page table entry is accessed, the GPU performs a DMA transaction using the translated physical address as a PCIe bus address. This bus address may be further translated by the host system's IOMMU.
- **SYSRAM** behaves like SYSRAM_NO_SNOOP, but enables cache coherence between CPU and GPU. When the GPU issues a write operation over the PCIe bus, all entries in the CPU's cache which correspond to the operation's target address are invalidated. This mode is useful if the same physical memory is mapped into both a CPU and a GPU address space, which is sometimes used to exchange data between CPU and GPU.

It is important to note that code executing on the GPU is oblivious to the type of physical memory backing its virtual address space. If the driver maps system RAM in place of GPU memory into a GPU address space, accessing this system RAM is thus completely transparent to the application.

Besides the two fields described above, various other fields exist in each page table entry:

- **Present (P)** indicates whether this page table entry is currently valid. If this bit is zero, the MMU ignores the remainder of the entry.
- **Supervisor (S)** indicates whether regular GPU operations from applications can access the memory referenced by this page table entry. If set to 1, only code executing in a special supervisor mode which is only available to the GPU driver can access this page.
- **Read-only (R)** determines whether the page can be written to. A value of one prevents the page from being written.
- **Encrypted (E)** determines whether the contents of this page are transparently encrypted by the GPU. This encryption is useful if GPU data is stored in system RAM but should not be accessed by the CPU.
- **Storage Type** allows the driver to choose between linear addressing and various tiling modes used primarily in graphics contexts.
- **Compression Tag** is used in conjunction with texture compression.

In addition, Figure 2.7 includes some fields labeled “U”. While these fields are in use on current GPUs, their exact meaning is currently not publically known.

While GPUs do support several memory-related features not commonly found in CPUs – such as transparent encryption or texture compression – they also lack a number of features that are commonly taken for granted on the CPU. For example, the GPU’s page tables do not contain the reference and dirty bits found in CPU page tables. Neither the GPU nor the GPU’s driver can thus determine which of the GPU’s memory pages are accessed frequently. In addition, most GPUs currently in use do not support transparent page faults as found on the CPU. Instead, these GPUs treat an access to a page for which no page table entry exists as a fatal error, forcefully aborting the GPU kernel the access originated from and raising an interrupt to the GPU driver. Although the driver could update the GPU’s page table in response to that interrupt, current GPUs do not support restarting a kernel where it was interrupted – typically, it is not even possible to restart a kernel from the beginning since that kernel may have changed the contents of the GPU’s memory before it was aborted, making the result of a repeated execution unpredictable. Due to these restrictions, the most common techniques for memory management – most notably demand paging and swapping of data to secondary storage – cannot be used on current GPUs.

More recently, however, these restrictions of GPU hardware have been diminishing: The Pascal generation of Nvidia GPUs includes limited page fault support [49]. Upon encountering a page fault, these GPUs are able to stop the faulting GPU kernel and raise an interrupt. The fault is then handled in the GPU driver, typically by adding a mapping to the faulting application’s GPU page tables and signalling the GPU to resume the faulting kernel’s execution. However, details about this process are not publicly known since no documentation on the hardware of current GPUs is available. Therefore, page fault support is currently only available in Nvidia’s proprietary driver, but not in any of the open-source GPU drivers currently available.

2.2.5 Performance Monitoring Counters

Programming on current GPUs often requires extensive tuning to achieve maximum performance. To assist these tuning efforts, current Nvidia GPUs feature a set of performance monitoring counters. Although these counters can monitor a variety of events from all parts of the GPU, they also come with one significant limitation: Due to the hardware structure of the performance monitoring counters, it is often impossible to count multiple related events simultaneously – for example, read and write accesses to memory cannot be counted at the same time. Nvidia’s own profiling tools work around this problem by executing each GPU kernel launched by the profiled application multiple times on the same input, counting one event in each repetition.

On Nvidia GPUs, the performance monitoring counters are grouped together in a dedicated engine named **PCOUNTER** [14]. Handling the performance monitoring counters is the only task of PCOUNTER; specifically, PCOUNTER never executes any code.

PCOUNTER is internally divided into **domain sets**. Most sets are connected to either a partition of the GPU's SMs or one of the GPU's memory controllers, the only exception being one dedicated set handling events related to neither computation nor memory. As a consequence, each set sees events from only a portion of the GPU – in particular, memory accesses are spread across multiple sets since the GPU interleaves these accesses across all memory controllers for increased performance. To capture all events of a certain type for the entire GPU, it is therefore necessary to use counters from all sets, summing up the counters' results after the profiled GPU kernel finishes execution.

Each domain set is further divided into eight **domains**. Each of these domains is connected to a set of 256 **signal lines**. Each signal line is associated with a specific event – such as the completion of an instruction – or state – such as whether or not a certain engine is currently busy or idle. To support more than 256 distinct signals, most signal lines are connected to only one domain per set. In addition to the signal lines, there are some lines connecting the domains, allowing an output of one domain to serve as input for another.

Each domain contains four **counters** operating on a shared clock. On each clock tick, each of these counters can sample up to four of the domain's signal lines. The counter combines these four lines using a freely-configurable **logic function**, increasing its count in each clock tick if this logic function yields true. Combining multiple signal lines in this way is often necessary since usable events are typically composed of more than one signal.

Even though each domain includes multiple counters, it is often impossible to count multiple related events concurrently since the domains feature different **operating modes**, some of which require multiple counters to work in tandem. In the simplest operating mode, each of the four counters of a domain independently counts events from its configured signal lines. In another mode, it is possible to start the actual counting only after a certain number of events of a certain type have occurred. To that end, one of the four counters counts down from a configured value, enabling another counter once the value of the first counter reaches zero.

Some events – such as memory accesses, which are always performed by all threads in a warp – can occur multiple times per clock tick. For such events, the GPU's engines have the ability to submit an integer value for each clock tick using multiple signal lines, with each line representing one bit of the total number. Conversely, the performance monitoring counters have an operating mode which sums up these integers: Instead of using the logic function, the counter interprets

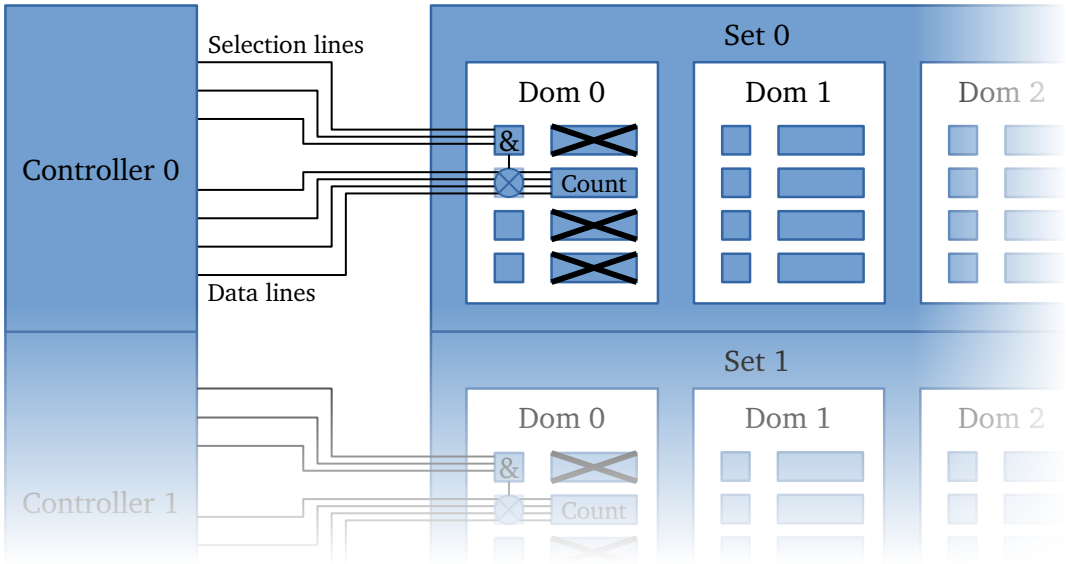


Figure 2.8: Counting memory accesses using the GPU's performance monitoring counters. Each of the GPU's memory controllers periodically transmits a four-bit number for each type of access using four signal lines (called data lines in the figure). The type of access is indicated by three more signal lines, called selection lines in the figure. PCOUNTER uses two counters from the first domain in each set to interpret these signals: The logic function of the first counter is connected to the selection lines, while the second counter sums up the values carried by the data lines whenever the first counter's logic function yields true. Note that only the first domain in each set is connected to the lines required for counting memory accesses, and that only the first two counters in that domain can be used with this type of signal.

its four input signal lines as a four-bit integer and adds the integer's value to its count in each clock tick. On the downside, however, these modes can use only one counter of each domain. Each domain can thus account for only one such event at the same time, which can occur up to 15 times per clock tick.

To reduce the number of signal lines required for such multi-line events, the GPU multiplexes some four-bit signals among multiple events as shown in Figure 2.8. For these events – which include memory accesses – the four signal lines carry a count for a different event in each clock tick, which yet other signal lines indicating the meaning of the current value. Another special operating mode exists to interpret this type of signal. In this mode, two of the four counters in a domain work in tandem while the remaining two counters remain unused: The first counter is used to configure which events should be counted, while the second one sums up the values for these events. To count read accesses to system

RAM, for example, the first counter is configured to combine three signal lines – one corresponding to memory accesses in general, one denoting a read access, and one indicating a PCIe bus transaction – using a simple, three-way logical and function. The second counter then adds a four-bit value to its internal count whenever the first counter's logic function yields true.

Although this scheme is quite complex, it also offers great flexibility. If, for example, the sum of both read and write accesses to system RAM should be counted, the only change required to the scheme described above would be to omit the read access line from the first counter. Although the domain still sees read and write accesses as distinct events in that case, the first counter's logic function would yield true for both events, causing the second counter to sum up both counts as desired. On the downside, however, all complex operating modes can only count one type of event at a time even if some of the domains' counters remain unused. As a consequence, separate counts for read and write accesses cannot be obtained in parallel since both are only countable by one domain of each set.

To enable convenient access to the GPU's performance monitoring counters, Nvidia provides an API named the **CUDA Profiling Tools Interface (CUPTI)** [48]. CUPTI provides a library interface to monitor both the GPU's performance monitoring counters and the CUDA operations (e.g., kernel launches or memory copy operations) issued by applications. CUPTI is intended as a tool to build more sophisticated profiling tools for GPUs – in fact, Nvidia's own profiling tools for GPUs are built on top of CUPTI. However, it is also possible for an application to profile itself by using CUPTI functionality. Most of our own knowledge about GPU performance monitoring counters was obtained by tracing GPU register accesses of such CUPTI-enabled applications.

2.3 Related Work

In this section, we review past research efforts relevant to this thesis. We discuss techniques for GPU virtualization – which must multiplex memory between multiple VMs – in Section 2.3.1, followed by a discussion of more general techniques for memory management on current GPUs in Section 2.3.2.

2.3.1 GPU Virtualization

Previous work on GPU virtualization largely falls into one of four categories [11]: Fixed passthrough, API remoting, device emulation, or mediated passthrough. In this section, we first describe these categories in general, before discussing specific research projects in more detail.

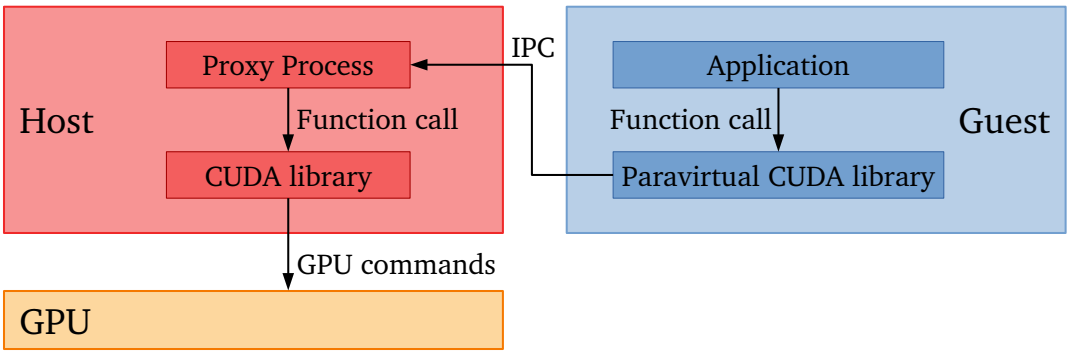


Figure 2.9: Basic principle of API remoting. The guest application calls the function of a paravirtual CUDA library. This library forwards the calls to a proxy process in the host, which executes the call on the application's behalf.

Fixed Passthrough

Fixed passthrough is a simple yet efficient method of virtualizing PCI and PCIe devices: The hypervisor maps the device's memory-mapped I/O registers into the guest-physical address space of the VM. The VM can then access these registers directly, without further intervention from the hypervisor. Fixed passthrough therefore allows the VM to access the device at native speed. On the downside, however, this approach requires that the device is given to a single virtual machine exclusively since fixed passthrough offers no way of coordinating multiple VMs accessing the same device.

To mitigate this issue, **single-root I/O virtualization (SR-IOV)** [10] was introduced. Devices supporting SR-IOV can be split into multiple virtual devices called **virtual functions**. To applications, these virtual functions appear identical to the original device, and each virtual function can be safely passed through to a different VM. Support for SR-IOV is found in many server-grade network adapters, but we are not aware of any GPUs supporting this feature.

API Remoting

The basic principle of API remoting is shown in Figure 2.9. To implement API remoting, applications running inside a virtual machine are linked with a paravirtual GPU runtime library. This library implements the same interface as the real runtime library in the host – e.g., CUDA, OpenCL or OpenGL – and is thus indistinguishable from the real runtime library to the application. Instead of implementing the functionality of the original runtime library, however, the paravirtual library forwards all calls to a proxy process running in the host system, typically using some kind of IPC or RPC mechanism. Each application is associated with a separate proxy process which encapsulates all GPU state associated

with the application; isolation between applications is then implemented by the operating system and GPU driver in the host. The proxy process executes the forwarded calls on the application's behalf by calling into the real runtime library, and sends the result of the call back to the paravirtual library, which returns it to the application.

The main advantage of API remoting is its ease of implementation: Since the proxy process simply calls into the real runtime library, no special knowledge about the inner workings of the GPU is required. However, this approach is also highly inflexible: Each implementation is tied closely to the runtime library in use. Consequently, an entirely new implementation must be created to support a new runtime library. In addition, API remoting requires a large amount of communication between the application and its proxy process. Since this communication crosses VM boundaries, API remoting tends to induce high overhead in applications.

Device Emulation

Device emulation can be seen as the opposite of API remoting: Instead of putting a virtualization component into the guest, device emulation attempts to present the guest with an entire emulated GPU, which is identical to the host's physical GPU. The hypervisor implements one such virtual GPU for each guest VM, maintaining separate state for each virtual GPU to keep VMs isolated from each other. Implementing this scheme is highly complex: The hypervisor must emulate all GPU control structures – e.g., command submission channels – as well as all GPU device registers, which can number in the thousands for a recent GPU. Emulation is typically implemented by trapping all accesses from the guest's device driver to the emulated structure, which tends to induce high application overhead since even simple operations often require a large number of device register accesses on current GPUs. If implemented properly, however, the virtual GPU is indistinguishable from a physical one – the guest can even use the same GPU driver as the host – and no assumptions about the guest's use of the GPU are necessary. In contrast to API remoting, device emulation is therefore not limited to a specific type of guest application. Instead, all applications – and even guest operating systems – are supported without requiring special support in either the guest or the hypervisor.

Mediated Passthrough

Mediated passthrough is a hybrid between API remoting and device emulation. The basic principle is shown in Figure 2.10: Only operations related to resource allocation are intercepted and forwarded to the hypervisor, while all other operations are sent to the GPU directly. In contrast to API remoting, mediated

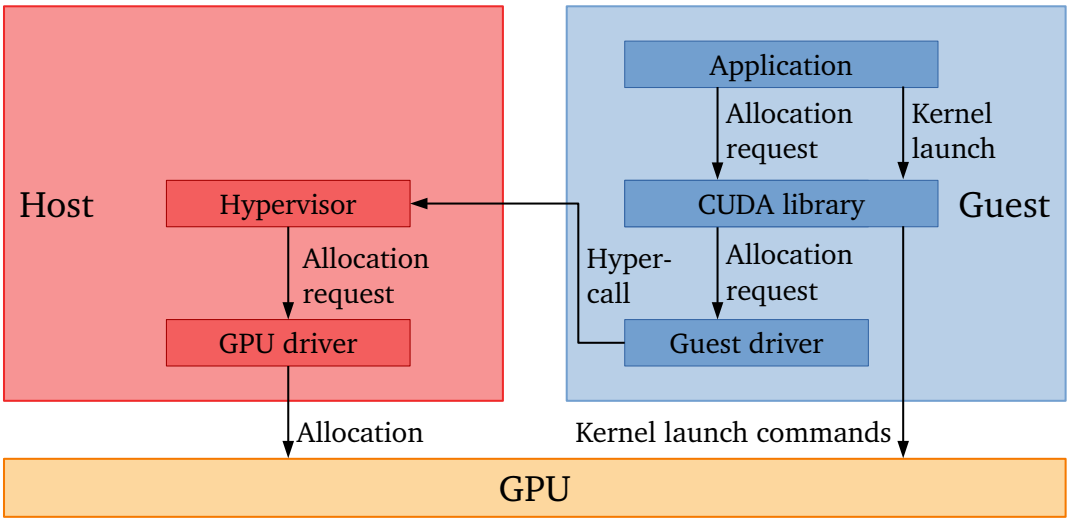


Figure 2.10: Basic principle of mediated passthrough. Resource allocation requests are forwarded to the hypervisor, while other GPU commands, such as kernel launches, are sent directly to the GPU.

passthrough operates on the driver level: A modified device driver intercepts those GPU operations that must be handled by the hypervisor. The hypervisor then forwards these operations to the actual GPU driver if it deems the operation safe to execute. Intercepting resource allocation requests allows the hypervisor to enforce isolation between VMs: Since all memory is allocated through the hypervisor, that hypervisor can easily ensure that VMs cannot access each other's memory by configuring all GPU address spaces accordingly. Once address spaces are set up, however, it is not necessary to intercept other commands, such as kernel launches, operating on those address spaces. As a result, the hypervisor can grant each guest application direct access to a command submission channel, which allows the application to submit kernels to the GPU without any overhead, while the GPU autonomously enforces address space boundaries.

The main advantage of mediated passthrough over API remoting is its increased flexibility: Since the guest driver implements the same interface as the GPU's original driver, the guest is not limited to a specific type of application; instead, any GPU runtime supporting the driver's interface can be used without modification. At the same time, mediated passthrough is much less complex to implement than device emulation: Since the original GPU driver in the host performs most of the actual work, much less knowledge about the inner workings of the GPU is required, though the resulting implementation is typically still closely tied to a specific GPU vendor. Most importantly, however, mediated passthrough typically causes lower overhead than both API remoting and device emulation since most commands are sent to the GPU directly.

Specific Projects

vCUDA [56] enables the use of CUDA in virtual machines. Being a prime example of a project using API remoting, vCUDA intercepts CUDA calls in a modified CUDA library and forwards them to a proxy process in the host system. To minimize the overhead of call forwarding, vCUDA builds upon VMRPC [25], a highly efficient, shared memory-based data transfer mechanism for virtual machines developed by the same authors. Using this approach, vCUDA supports sharing of a single GPU between multiple VMs as well as VM migration, while inducing an overhead of 11 % on average. However, even though vCUDA is able to safely share a GPU between VMs, it does not guarantee any fairness. In particular, vCUDA allocates GPU memory in a first-come-first-served fashion, though the authors briefly discuss the possibility of statically partitioning GPU resources, which could be implemented by modifying the result of certain CUDA calls to make applications believe that there are fewer resources – e.g., less RAM – than are physically available.

GViM [21] is a predecessor to vCUDA. Like its predecessor, GViM employs API remoting, but uses a guest kernel driver in addition to a fake CUDA library. Using the driver's extended privileges GViM establishes an efficient, shared memory-based communication channel between guest and host to enable fast data transfer between the application and the GPU. In contrast to vCUDA, GViM also addresses fairness between multiple VMs, implementing a scheduler for CUDA kernels which can guarantee a certain amount of GPU computation time to each application. However, GViM limits its fairness considerations to computation time, but does not address memory.

API remoting is not limited to the CUDA API: **VMGL** [41] uses the same technique to allow VMs to render 3D graphics using OpenGL. VMGL uses a custom network-based transport, which the authors call WireGL, over a loopback interface to forward GL calls to a proxy process in the host. In addition, the authors make modifications to the guest's X server to allow the guest to display self-rendered 2D graphics as well as 3D graphics rendered by the proxy process on the same screen. As a result, VMGL enables guest access the full OpenGL API as well as VM suspend and resume at about 14 % of overhead. VMGL allows multiple VMs to share the same GPU, but does not attempt to guarantee fairness between VMs. Specifically, VMGL allocates memory in a first-come-first-served fashion.

rCUDA [13] extends the idea of API remoting to allow applications to use GPUs located in a remote machine. To that end, communication between the fake CUDA library and the proxy process takes place over a network instead of an IPC mechanism within a single machine. This approach allows datacenter applications which do not saturate a GPU by themselves to share that GPU with other applications, which reduces the number of GPUs as well as the amount of energy required. RCUDA supports the entire CUDA API with the exception of

zero-copy data transfer, which is typically implemented by mapping the same memory into both a CPU and a GPU address space and is thus fundamentally incompatible with using a remote GPU over a network. Since the network is much slower than the PCIe interconnect typically used for GPUs, rCUDA induces a high application overhead of 71 % on average. In addition, rCUDA requires changes to the application's source code to avoid the use of undocumented functionality – however, this appears to be an implementation artifact since vCUDA has no such limitation. Like the projects described above, rCUDA does not specifically deal with GPU memory.

VOCL [64] implements a similar functionality as rCUDA, but based on OpenCL instead of CUDA. VOCL uses MPI instead of sockets for data transfer between machines and can thus leverage high performance interconnects efficiently and transparently. Compared to rCUDA, VOCL introduces four optimizations to reduce the application overhead. First, rCUDA does not properly handle local GPUs, but instead assumes all GPUs to be remote. In contrast, VOCL simply calls the real OpenCL library if a call made by the application targets a local GPU, bringing the overhead of using a local GPU down to virtually zero. Second, in case of a remote GPU, VOCL locally caches kernel arguments until the kernel is launched, and sends these arguments to the remote machine as part of the actual kernel launch operation. This optimization significantly reduces the number of round-trips required to launch a GPU kernel. Third, VOCL pipelines data transfer over the network with data transfer between CPU and GPU on the remote machine, which increases the transfer bandwidth to and from the remote GPU. Fourth, VOCL forces all errors of asynchronous calls to be returned asynchronously, which eliminates all waiting in those calls. Using these optimizations, VOCL reduces the overhead for compute-intensive applications to below 5 %. For applications transferring large amounts of data, however, the network still poses a bottleneck. As a result, these applications experience overheads of up to 150 %. Memory management is again outside the scope of the project's scope: VOCL allocates memory in a first-come-first-served fashion.

Becchi et al. [6] developed another virtualization solution for GPUs based on API remoting. Their work divides each GPU into multiple virtual GPUs (vGPUs), each of which contains the entire GPU execution state of one application. An application is mapped to a vGPU when its first kernel is launched, while vGPUs are mapped to a physical GPU dynamically on demand, including migration of vGPUs to a different physical GPU at runtime. vGPUs may also be mapped to a physical GPU located in a remote machine. To ensure fairness between vGPUs, the system includes a scheduler for GPU kernels: Kernels submitted for execution are queued in software, with the scheduler selecting the next kernel to execute whenever the previous kernel finishes execution. Becchi et al. also enable oversubscription of GPU memory: Their system initially allocates all memory in a staging area in system RAM and returns an internal handle instead of a pointer. When the

application subsequently launches a kernel, it passes the internal handles to the kernel as parameters, which allows the runtime to transfer exactly the data needed by the kernel to the GPU. If there is insufficient GPU memory available, unneeded data from the same address space may be evicted. Evicting data from other address spaces is possible as well, but only in a cooperative fashion: If no data from the same address space can be evicted, other applications are asked to voluntarily swap some of their data; however, these applications are free to decline the request. If all applications refuse to swap, the runtime can migrate the application's vGPU to a different physical GPU as a last resort. While this approach allows a GPU to run applications exceeding the GPU's physical memory capacity, Becchi et al. do not address fairness: Uncooperative applications can hold on to large amounts of memory, leaving other applications no choice but to migrate to a different GPU. In addition, their solution depends on software scheduling of GPU kernels since the runtime must check each kernel's parameters whether copying is needed prior to kernel launch. However, such software scheduling disables the GPU's internal, highly efficient scheduling and context switching and thus causes application overhead, which the authors measured at about 10 %.

The authors of **GPUvm** [58] opted for device emulation instead of API remoting. GPUvm provides a fully functional virtual GPU to each VM, allowing the VM to use the same GPU driver as the host without modification. GPUvm forwards trapped register accesses from the guest device driver to a *GPU access aggregator* in the host. This access aggregator serves a similar purpose as the proxy process in API remoting, but calls into the host's GPU driver directly instead of relying on a high-level interface such as CUDA. Due to the frequent trapping of device accesses, GPUvm's initial prototype caused an overhead of up to 140x, which prompted the authors to introduce a degree of paravirtualization, bringing the overhead down to less than 3x. Besides providing a fully functional virtual GPU, the authors also address the topic of fairness between VMs: GPUvm includes a scheduler for GPU kernels which distributes GPU computation time fairly among VMs. In addition, GPUvm partitions the available GPU memory between VMs to guarantee a fair share of that memory to each VM. The authors also discuss the possibility of allocating memory dynamically; in any case, however, GPUvm's memory allocation is limited to the amount of available GPU memory, while oversubscription of that memory is beyond the scope of the project.

LoGV [20] was any early work employing mediated passthrough to virtualize a GPU. In contrast to GPUvm, LoGV attempts to grant the guest VM direct access to GPU resources whenever it is safe to do so – most importantly LoGV managed to grant the guest VMs direct access to the GPU's command submission channels. LoGV provides a para-virtual device driver for the guest to trap resource allocation requests, but does not require modifications to the guest's user space software stack. LoGV enables safe sharing of a GPU between multiple guests as well as VM migration at an overhead of less than 3 %. However, LoGV focuses on safety

and performance rather than fairness. As a consequence, LoGV does not include any scheduling of GPU kernels, and allocates memory in a first-come-first-served fashion.

gVirt [61] is unique in that it uses mediated passthrough without relying on a para-virtual guest driver. GVirt targets Intel GPUs, which differ from dedicated GPUs in two main ways: First, these GPUs do not feature dedicated GPU memory, but use system RAM instead. Second, these GPUs only feature a single command submission channel, which makes passing this channel through to a guest application impossible. Nonetheless, gVirt presents the VM with a fully functional virtual GPU, to an extent such that the guest OS can use the same GPU driver as the host – though the authors did make minor modifications to that driver to reduce the virtualization overhead to less than 10 % for most workloads. GVirt multiplexes the GPU's sole command submission channel by alternating this channel between multiple VMs in a round-robin fashion, granting each VM access for a short timeframe before moving on to the next VM. This approach also achieves fairness between VMs with respect to GPU computation time. In addition, GVirt is able to multiplex the GPU's virtual memory between VMs. Intel GPUs use two distinct virtual address spaces: *Local GPU memory*, which is accessible only to the GPU, and *global GPU memory*, which is accessible to both CPU and GPU. Both of these address spaces are backed by system RAM and normally shared between all applications. To multiplex the local GPU memory, gVirt switches the GPU page tables along with the command submission channel, essentially granting each VM a dedicated address space. This constitutes a form of oversubscription since the total amount of local graphics memory available to all VMs is larger than the amount of memory normally supported by the GPU; however, the GPU is still limited to the amount of physical memory available, which makes this scheme unfit for GPUs including dedicated GPU memory. For the global GPU memory, gVirt does not use this type of switching to avoid consistency problems between CPU and GPU. Instead, gVirt employs a form of partitioning using a novel technique called *address space balooning*: When a region of global GPU memory is allocated by a VM, the hypervisor marks all guest-physical pages in this region as unavailable in all other VMs. The current implementation of gVirt does not support oversubscribing the global GPU memory; however, this limitation was later removed by gScale [66], which we describe below.

2.3.2 GPU Resource Management

Gdev [39] attempts to turn GPUs into first-class operating system resources similar to CPUs. Besides making GPU acceleration available to the operating system itself, Gdev introduced GPU scheduling at the operating system level: Gdev includes an admission control mechanism as well as a scheduler which can ensure a fair distribution of GPU time between applications after admission. Gdev's scheduler

requires applications to submit new GPU kernels into a software queue instead of the GPU's command submission channels. Whenever a kernel finishes execution, the scheduler then selects one of the submitted kernels and inserts the necessary commands for launching that kernel into a command submission channel. At the same time, Gdev configures the GPU to raise an interrupt when the submitted kernel finishes execution. When this interrupt arrives, Gdev's scheduler is activated again to select the next kernel to run. While this scheme achieves a fair distribution of computation time, it also disables the GPUs internal scheduling and context switching: From the GPU's point of view, only one kernel is queued at any given time, and whenever a GPU kernel finishes execution, the GPU must wait for the scheduler before launching the next kernel. Gdev's scheduling thus induces considerable overhead in some applications.

In addition to scheduling, Gdev makes several changes to the GPU's memory management. For example, Gdev enables GPU applications to allocate shared memory segments on the GPU. Normally, these segments must be explicitly allocated as shared by all applications wishing to share the same buffer. However, Gdev uses the same mechanism in conjunction with the scheduler to enable transparent oversubscription of GPU memory: When an application fails to allocate a new GPU buffer due to insufficient GPU memory, Gdev searches the address spaces of other applications for buffers of equal or larger size than the requested allocation. If a suitable buffer is found, Gdev transparently shares that buffer with the allocating application. Subsequently, Gdev swaps the contents of the shared buffer on kernel launches to give each of the two applications the illusion that it owns the buffer exclusively: Whenever one of the two applications launches a GPU kernel, Gdev copies that application's data to the shared buffer, evicting the data of the other application to system RAM in the process.

While this scheme allows applications to allocate more GPU memory than is physically available, it also comes with a number of disadvantages. First, Gdev's swapping fundamentally depends on software scheduling of GPU kernels: Once a buffer is shared between two applications, Gdev must be able to guarantee that only one of these applications can run a GPU kernel at any given time, which is impossible without scheduling kernels in software. Second, since Gdev does not know which shared buffers a GPU kernel actually needs, Gdev's swapping mechanism indiscriminately copies the contents of all shared buffers in the application's address space to the GPU prior to kernel launch, which may include data that the kernel does not touch. Third, Gdev requires all data in the application's address space to be in GPU memory before launching one of the application's kernels, which can be problematic in situations where very little GPU memory is available.

GDM [63] is an extension to Gdev which generalizes Gdev's approach to memory management. Instead of implicitly sharing buffers, GDM performs allocations in a *staging area* in the application's CPU address space. When an application

subsequently launches a GPU kernel, GDM moves all application data from the staging area to the GPU prior to kernel launch; if there is insufficient space left on the GPU, GDM evicts data from other applications to these applications' staging areas. Internally, GDM divides GPU buffers into blocks and performs all DMA operations on these blocks rather than entire buffers. As a result, GDM is able to evict only part of a buffer to system RAM if doing so frees up sufficient space on the GPU. In addition, GDM tracks which blocks have changed by computing an MD5 hash for each block and comparing this hash to the hash of the corresponding block in the staging area prior to eviction. Blocks found to be unchanged are not copied back into system RAM, but simply overwritten on the GPU. Finally, GDM is able to perform some copy operations asynchronously: The data of the next kernel can be copied to the GPU while the previous kernel is still running as long as doing so does not require data used by the running kernel to be evicted. Despite constituting a major advance compared to Gdev, however, GDM still suffers from essentially the same limitations as its predecessor: First, GDM still depends on software scheduling of GPU kernels since it must ensure that all the application's data is on the GPU prior to kernel launch, and that data needed by running kernels is not evicted from the GPU before the kernel finishes execution. Second, GDM does not allow GPU kernels to use system RAM directly, but requires the application's working set to be in GPU memory while the application's kernels are running, which may not be possible in situations where little GPU memory is available. Third, GDM typically does not have information about the exact data needed by each kernel, and may thus transfer unneeded data to the GPU on kernel launch. GDM attempts to alleviate the latter two disadvantages by introducing a new API which allows the application to specify explicitly which buffers are required by each of its GPU kernels. However, using this API requires modifications to the application, and does not solve the dependency on software scheduling.

TimeGraph [38] is a real-time scheduling system for GPUs mainly targeted at graphics applications, allowing these applications to maintain constant frame rates even under heavy load. To that end, TimeGraph includes a priority scheduler for GPU kernels as well as a reservation mechanism which can guarantee a fixed amount of GPU time to each application. Like Gdev, TimeGraph's scheduler maintains software queues for submitted GPU kernels, starting the highest priority kernel from its queues whenever a kernel finishes execution. In addition, TimeGraph includes a fast track mechanism – called the *high throughput policy* – which allows the application currently executing on the GPU to submit further kernels directly to the GPU as long as no higher-priority applications are waiting to use the GPU. In contrast to Gdev, TimeGraph focuses on scheduling only, and thus does not address memory. However, the authors explicitly measured TimeGraph's scheduling overhead for a single application instance, while related projects typically evaluated only the overall throughput of the GPU for multiple concurrent applications. Specifically, the authors measured decreases of 17–28 % in the

frame rate of several graphics applications if every GPU kernel was dispatched in software, and frame rate decreases of about 4 % using the high throughput policy – which would be incompatible with a GPU swapping mechanism such as Gdev's. These results indicate that software scheduling of GPU kernels is a problem at least for some applications.

PTask [54] defines a new API for GPU applications with the goal of minimizing the amount of data movement between CPU and GPU. An application using the PTask API is composed of multiple *parallel tasks* (PTasks) communicating through *pipes*. These pipes are similar in spirit to UNIX pipes: A PTask writes its output data into a pipe, and other PTasks then read this data from the pipe as input. Through the pipe abstraction, the PTask runtime gains information about which data is needed by which PTasks as well as the origin of that data. The runtime can thus schedule ready PTasks such that minimal data movement is needed, for example by scheduling a PTask on a GPU that already has most of the task's inputs in memory. Using the same information, the runtime could also oversubscribe GPU memory to some extent: Since the working set of each PTask is known, unneeded data can be identified and evicted from the GPU. Since the runtime is implemented partially in the kernel, this eviction would also work across address spaces. However, the authors explicitly do not target scenarios where the demand for GPU memory exceeds the GPU's capacity, but instead focus solely on minimizing data movement.

NEON [44] takes a different approach to GPU scheduling than its predecessors: Instead of dispatching every GPU kernel in software, NEON applies fair queuing to GPU kernels. Initially, NEON allows all applications to submit kernels directly to the GPU, but monitors each application's GPU usage. If an application is found to use more than its fair share of GPU time, that application's access to the GPU is temporarily suspended to allow other applications to catch up. Using this approach, NEON is able to maintain fairness between applications with respect to GPU computation time. At the same time, NEON keeps the scheduling overhead below 5 % since it only interferes with the GPU's internal scheduling and context switching when necessary to correct unfairness. NEON's approach is incompatible with a GPU swapping mechanism such as Gdev's: If GPU kernels are sent to the GPU directly, it is impossible to move a kernel's working set to the GPU prior to launch. However, NEON's comparatively low scheduling overhead indicates that decoupling swapping from software scheduling of GPU kernels is worthwhile.

GPU Maestro [52] aims to reduce interference between multiple GPU applications. The authors investigate the behavior of GPU applications sharing the same GPU resources – e.g., the same SMs – concluding that some applications suffer from performance degradation when sharing resources while others do not. The authors propose to partition the GPU's internal resources to mitigate this problem. As a first step, GPU Maestro probes all running applications for mutual interference: For each pair of applications running on the GPU, GPU Maestro temporarily places

a thread group of each application on the same SM. Subsequently, GPU Maestro uses the GPU's performance monitoring counters to detect any performance degradation in these thread groups compared to the group running alone on an SM. After probing all running applications in this manner, GPU Maestro assigns interfering applications to separate SMs, while allowing non-interfering applications to share the same SMs to increase utilization. Using this approach, GPU Maestro is able to significantly increase the GPU's overall throughput as well as to reduce the turnaround times of individual GPU kernels. GPU Maestro considers the SM's compute capacity as well as the size of the SMs' register file and shared memory in its placement decisions, but does not address GPU memory.

Agarwal et al. [4] investigate strategies for page placement in heterogeneous CPU/GPU systems. Somewhat counterintuitively, the authors discover that placing some GPU data in system RAM can actually result in an application speedup: Since current GPUs can access GPU memory and system RAM in parallel, distributing data over both memories increases the available memory bandwidth, resulting in a speedup if memory bandwidth is a limiting factor for the application. In addition, the authors develop a hinting scheme for applications with a working set larger than the available GPU memory. These hints allow the application to specify which data is critical to the application's performance and should therefore be placed in GPU memory. While this scheme is similar in spirit to our own, the authors focus on HPC systems where only one application with a large working set runs at any given time, but do not consider GPUs shared between multiple applications. As a consequence, the authors allocate data directly in GPU memory or system RAM according to the application's hints, but do not support moving data between the two memories at runtime in response to other applications starting or exiting. In addition, the authors consider application buffers as indivisible: Buffers are allocated entirely in either GPU memory or system RAM – it is not possible to allocate only part of a buffer in GPU memory even if space is available. Finally, the authors target a hypothetical platform in which CPU and GPU have cache-coherent access to each other's memories using an interconnect much faster than contemporary PCIe. This target platform is similar in spirit to APUs – which consist of a CPU and GPU on a single chip sharing the same memory bus, but typically do not include dedicated GPU memory – such a platform does not exist at the time of this writing.

To generate their application hints, Agarwal et al. also develop a compiler-based profiling mechanism for GPU memory accesses. In this approach, a modified compiler inserts additional instructions next to each instruction accessing memory. These additional instructions are then used to count the number of accesses to each virtual memory page at runtime. While this approach is effective in generating access profiles, it suffers from two main drawbacks: First, the profiling mechanism will only gather data for those parts of the application that have been compiled with the authors' modified compiler. As a consequence, not only the application

itself, but also all shared libraries used by the application must be compiled with the authors' compiler, which can pose a problem if the source code of some of these libraries is not available. Second, the authors' approach is inherently limited to code written in a language compatible with the authors' modified compiler – in this case, CUDA code. Nonetheless, the authors' profiling shows that most applications display a high degree of uniformity in their memory accesses within application buffers, while the number of accesses can vary greatly between buffers – a result which agrees well with our own profiling.

Region-Based Software Virtual Memory (RSVM) [34] uses a cooperative approach to GPU memory management. RSVM is a user space library which aims to make data transfer between CPU and GPU transparent to the application by copying data between CPU and GPU on demand. RSVM uses application-defined memory regions as its unit of memory management. By default, each application buffer consists of a single region, but application developers can also divide buffers into multiple regions during allocation. To detect accesses to memory regions, RSVM requires both CPU and GPU code to issue an explicit *map* call for each region before access. Conversely, the application is expected to *unmap* regions which are no longer in use. Each region can only be mapped at either the CPU or the GPU side at any given time; the contents of the mapped region are implicitly copied to the mapping side during the map operation. If a region is unmapped on the GPU side, that region stays in GPU memory if sufficient space is available; copying data between CPU and GPU is thus unnecessary if that region is not touched from the CPU side. If GPU memory is scarce, however, RSVM implements cooperative swapping of GPU data: If an application attempts to map a region to the GPU, but insufficient GPU memory is available, RSVM may evict unmapped regions from the application's address space to system RAM. Due to its implementation as a user space library, however, RSVM cannot swap regions from other application's address spaces and is thus unable to ensure fairness between applications. To select regions to swap, RSVM uses a not-frequently-used (NFU) scheme, treating regions as used whenever they are mapped. RSVM does not include the regions' sizes in its eviction decisions, which can lead to larger-than-necessary amounts of data being evicted since RSVM cannot partially evict regions. The main drawback of RSVM is that applications must be modified to use RSVM's API. This modification has been shown to induce overheads of about 20 % especially for compute-intensive applications even if no swapping is required. In addition, the required modifications can be complex for existing applications since for best performance, application buffers must be manually divided into regions, which requires in-depth application knowledge.

Gscale [66] is an extension of gVirt which enables oversubscription of the global GPU memory on Intel GPUs. Instead of relying on gVirt's address space balooning, gScale introduces a separate guest-physical address space for each application's global GPU memory. During GPU context switches, gScale switches the global

GPU memory page tables in the same way as gVirt does for the local GPU memory. For global GPU memory, however, switching page tables is complicated by the fact that this memory is accessible to both CPU and GPU: To present a unified view of the GPU's address space to both CPU and GPU, each access to global GPU memory is normally routed to the GPU over PCI express, translated using the GPU's page table, and then routed back to system RAM, again over PCI express. This scheme leads to problems in virtualized contexts where CPU and GPU are scheduled independently: If one CPU process is scheduled on the GPU and a second process simultaneously attempts to access global GPU memory, that second process would have its access translated using the page tables of the first process and thus gain access to the first process' memory. To overcome this problem, Gscale mirrors GPU page table entries to the CPU page table of each process, adding entries pointing directly to the system RAM backing the global GPU memory. Processes can thus access their own global GPU memory without relying on the GPU's page table, and thus independently of GPU scheduling. Using this scheme, Gscale is able to oversubscribe the global GPU memory as well, which increases the maximum number of concurrent VMs supported, while causing a small increase in overhead due to more complex page table handling. Like gVirt, however, Gscale's approach is not applicable to GPUs with dedicated memory.

Nvidia's CUDA has included a feature called **Unified memory** [24] since version 6.0. The goal of unified memory is to create a single address space used by both CPU and GPU such that both can use the same pointers and explicit copying of data between CPU and GPU is unnecessary. To that end, the CUDA runtime synchronizes buffer contents between CPU and GPU on demand: When a pointer to a buffer currently residing in system RAM is passed to a GPU kernel, the runtime unmaps the buffer referenced by the pointer from the application's CPU address space and copies the buffers contents to the GPU prior to kernel launch. When the same buffer is subsequently accessed from the CPU, that access raises a page fault, which prompts the runtime to copy the buffer's contents back into system RAM. Unified memory is thus able to oversubscribe GPU memory to some extent: Only the application's current working set – i.e., the buffers passed to the currently executing kernels – must be in GPU memory, while any remaining data may be evicted to system RAM. However, unified memory targets HPC environments – where there is typically only a single applications with a large working set executing on the GPU at any given time – while shared-GPU environments are outside the scope of the project. As a consequence, unified memory does not consider evicting of GPU data from other applications.

2.4 Summary

Even though current GPUs still function as accelerators and are thus subordinate to the CPU, these GPUs have adopted a number of features from the CPU world.

With respect to memory, for example, today's GPUs feature virtual address spaces and address translation hardware similar to those found in CPUs. The page tables defining the GPU's address spaces are managed by the GPU driver, allowing that driver to dynamically allocate memory to applications as needed. GPU address spaces can use both GPU memory and system RAM as a backing store; it is thus possible to make GPU applications use system RAM without the application's knowledge if the GPU's own memory proves insufficient. However, there is also a number of memory-related features that are common in the CPU world but missing from current GPUs. For example, current GPUs typically do not support page faults or preemption and do not include reference bits in their page tables. Traditional approaches to extending memory therefore do not apply to GPUs: The lack of page fault support precludes demand paging, the missing reference bits impede the use of traditional swapping policies, and the inability to preempt running kernels makes it difficult to rearrange memory allocations at runtime. Nonetheless, the feature set of current GPUs is sufficiently advanced to open up interesting possibilities with respect to GPU memory management.

Despite this wealth of possibility, however, the problem of oversubscribing GPU memory has not yet been solved. Most research projects that should address the problem – most notably those in the area of GPU virtualization – typically either allocate GPU memory in a first-come-first-served fashion, which results in poor fairness among applications, or statically partition that memory, resulting in poor utilization. Those research projects that employ more sophisticated techniques typically suffer from two drawbacks: First, these projects assume that all data a kernel might need must be physically on the GPU before that kernel can execute, leading to unnecessary copying of data between CPU and GPU. Second, existing solutions integrate memory management with software scheduling of GPU kernels to be able to intercede in the applications' execution on the GPU when memory allocations must be rearranged. Software scheduling of GPU kernels can, however, cause significant application overhead even if sufficient GPU memory is available. To date, we are not aware of a solution that achieves fairness, high memory utilization and low overhead at the same time.

Chapter 3

An Eviction Mechanism for GPUs

GPUswap is a novel mechanism to extend GPU memory with system RAM. When insufficient GPU memory is available, GPUswap evicts application data from the GPU to system RAM, but keeps the evicted data accessible to the GPU. As a result, applications can submit kernels directly to the GPU without requiring software scheduling, resulting in lower application overhead than with previous work. In addition, GPUswap performs evictions in response to memory allocation requests instead of kernel launches as in previous work, which reduces the amount of data transferred between CPU and GPU.

In this chapter, we present the design and implementation of our eviction mechanism. The corresponding eviction policy is discussed in Chapter 5.

3.1 Goals

Memory contention is always associated with overhead: Memory contention in a CPU-only system, for example, can lead to frequent swapping to secondary storage which may slow the system down to the point where it becomes unusable. We therefore consider memory contention to be an exceptional case rather than the norm in any system. Consequently, though GPUswap is meant to deal with memory contention, we chose to optimize for the common case that there is sufficient GPU memory available: An eviction mechanism should keep applications running if there is memory contention, but should not negatively affect applications if there is not. Specifically, we formulate the following goals for GPUswap:

Performance GPUswap should minimize application overhead. Specifically, since we expect memory contention to be the exception rather than the common case, GPUswap should cause no overhead at all if sufficient GPU memory is available. In the presence of memory contention, we consider overhead unavoidable. However, GPUswap should keep that overhead as small as possible.

Generality GPUswap should work out of the box with arbitrary applications. Previous work was often tied closely to a specific GPU API, such as CUDA, or even introduced new APIs. As a consequence, these designs limited the applications running on the GPU to those supporting the right APIs. In contrast, GPUswap should not make assumptions about the applications running on the GPU.

Transparency GPUswap should not require explicit cooperation from applications to operate. Not relying on cooperation has two distinct advantages: First, GPUswap does not require applications and their developers to explicitly manage resources, or even know about the amount of available resources or the current degree of sharing. Therefore, application development is simplified – a shared GPU can be used in the same way as a dedicated one – and existing applications continue to work without modification. Second, applications may not always cooperate, for example due to malfunctions or malicious intent. A transparent design allows GPUswap to enforce its decisions against the application's will if necessary.

3.2 Design

To overcome memory shortages, GPUswap uses system RAM to extend the GPU's memory: When GPU memory is running low, GPUswap evicts data from the GPU to system RAM. To keep the evicted data accessible to the GPU, GPUswap then maps the virtual address range corresponding to the evicted data to the data's new location in system RAM. This approach allows GPU applications to use more memory than is physically available on the GPU, while still allowing applications to submit kernels directly to the GPU since all application data is accessible at any time.

3.2.1 Overview

The basic architecture of GPUswap is shown in Figure 3.1. GPUswap is composed of three main components: An **eviction policy**, a **relocation mechanism** and an **accounting mechanism**. When an application allocates memory, GPUswap first examines the accounting data to determine whether there is sufficient GPU memory available to fulfill the request. If not, GPUswap invokes the eviction policy, passing the amount of memory that must be freed before the allocation request can be served. The policy's task is to select a set of application pages to be evicted to system RAM using data from the accounting mechanism. The set of selected pages is then passed to the relocation mechanism, which moves the contents of the selected pages to system RAM and updates the application's page tables. Once the relocation mechanism has finished its task, the request is forwarded to the original allocation mechanism in the GPU driver, which can now fulfill

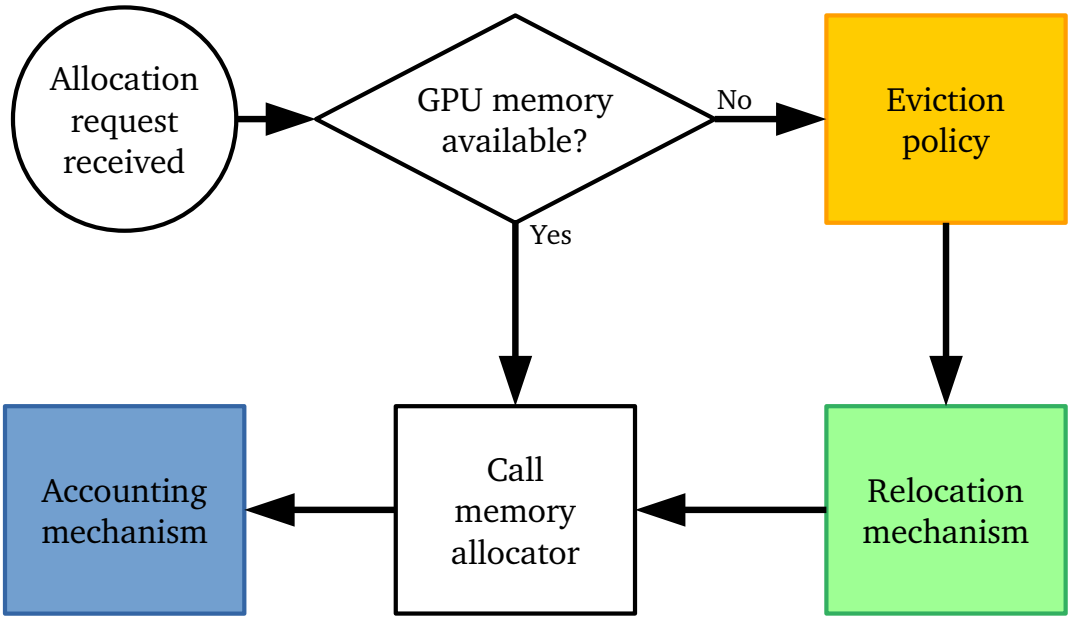


Figure 3.1: Flowchart of the basic operation of GPUswap. GPUswap's own mechanisms are shown in color. The memory allocator is part of the original GPU driver.

the application's allocation request. Finally, GPUswap's accounting mechanism keeps track of the available GPU memory and the amount of memory consumed by each application after the allocation as well as any necessary relocation have been performed. The accounting mechanism also divides the newly allocated GPU memory into fixed-size **chunks** and maintains a list of all allocated chunks. Both the policy and the relocation mechanism operate on these chunks rather than individual pages to reduce the processing overhead.

While memory allocation is a common case, GPUswap may be invoked by events other than memory allocation as well. When an application frees memory, for example, GPUswap performs the same process described above in reverse to maintain high memory utilization: The eviction policy selects a set of pages to return to GPU memory, which the relocation mechanism subsequently moves back to the GPU. It is also conceivable to invoke the eviction policy periodically, for example to reevaluate past decisions or to react to changes in application behavior.

GPUswap is designed to be part of the GPU's driver. Being part of the driver – and thus running in kernel mode – allows GPUswap to be fully transparent to applications: GPUswap can freely manipulate page tables and stop applications from running, both of which are necessary to implement many of the mechanisms used by GPUswap. In addition, the driver's API is used by all applications, regardless of the application type, which allows GPUswap to remain application-oblivious.

3.2.2 Memory Relocation

The relocation mechanism is the core of GPUswap. In the event of memory shortage, the mechanism receives a set of memory chunks selected by the eviction policy as input, and is then responsible for moving the contents of these chunks from the GPU to system RAM. In the process, the relocation mechanism must also make the chunks' new location accessible to the application by changing the appropriate entries in the GPU's page table. Note that the policy may also return chunks from a buffer that is currently being allocated. Since these chunks do not yet exist in GPU memory, our relocation mechanism allocates these chunks in system RAM directly without the need for a relocation operation.

To relocate memory from the GPU to system RAM, our relocation mechanism performs five steps:

1. Select an application with at least one selected chunk in its address space
2. Suspend all access to the GPU from that application
3. Move the data inside all selected chunks from the application's address space to system RAM
4. Update the application's page tables such that the virtual addresses of relocated chunks map to the data's new location in system RAM
5. Restore the application's GPU access

The relocation mechanism repeats these steps until all chunks returned by the policy have been relocated. After the relocation finishes, it is safe to call the GPU driver's original allocation mechanism since relocation frees up sufficient GPU memory to service the allocation request that triggered the relocation.

Chunks

Our relocation mechanism operates on chunks rather than pages or entire application buffers. Relocating entire buffers, which can easily be hundreds of megabytes in size, would likely lead to poor memory utilization: If a large buffer is moved to system RAM in response to a small allocation, most of the space freed by the large buffer's eviction would remain unused. In contrast, relocating memory with page granularity guarantees high utilization, but results in increased relocation overhead: Since the eviction policy is not required to select adjacent pages, each page must be relocated using an individual DMA operation, which requires time to set up. In addition, the policy itself may also require a large amount of processing time if a large number of pages must be selected for eviction in response to a large allocation request.

Operating on chunks provides a middle ground between these two extremes: Given that the chunk size is larger than the page size, but smaller than the typical application buffer, operating on chunks reduces the amount of processing and the number of DMA transactions required for relocations, while at the same time limiting the amount of unused GPU memory. Chunks thus introduce a tradeoff between utilization and overhead: Smaller chunks maximize memory utilization at the cost of increased relocation overhead, while larger chunks reduce the overhead at the cost of lower memory utilization. Since the optimal chunk size depends on a variety of factors, such as the bandwidth of the interconnect between CPU and GPU or the requirements of the applications, GPUswap does not set the chunk size to a fixed value, but instead allows the system administrator to configure a chunk size according to circumstances. We analyze the effects of GPUswap's chunk size on applications in Section 6.4.

GPUswap generally attempts to divide all application buffers into chunks of the configured size at allocation time. If, however, a buffer's size is not an exact multiple of the chunk size, the last chunk in the buffer cannot be filled to capacity. In that case, GPUswap allocates a chunk smaller than the configured size – which we call a **remainder chunk** – at the end of the buffer to avoid internal fragmentation.

Suspending Applications

While the relocation mechanism is moving data from the GPU to system RAM, applications cannot be permitted to make changes to this data. If an application were to change data that has already been copied to system RAM, that change would be lost when the application's mapping is updated to point to the data's new location. On the CPU, this problem is typically solved by mapping the data read-only during migration and using page faults to detect write attempts [9]. However, this method does not apply to current GPUs since these GPUs treat page faults as fatal errors. Therefore, the relocation mechanism must ensure that no GPU kernels can run in the application's address space while data is being relocated. To that end, the relocation mechanism suspends GPU access altogether for each application while chunks from the application's address space are being relocated.

Suspending GPU access for an entire application inevitably causes a delay in the application's execution. To minimize the effects of that delay, the relocation mechanism processes one application at a time: Since it is only necessary to suspend GPU access for the one application whose address space is being manipulated, all other applications can freely run GPU kernels. Although the application being processed still experiences a delay with this method, the GPU's utilization remains high since other applications are not affected.

Data Transfer

Once GPU access for an application has been suspended, the relocation mechanism can safely move data from the application's address space to system RAM. To that end, the relocation mechanism iterates over all selected chunks from the application's address space. For each chunk, the mechanism allocates a buffer of the same size in system RAM. Since the CPU never accesses the system RAM occupied by relocated chunks, the mechanism disables CPU cache coherence for this buffer by setting the *targ* field in the GPU's page table to *SYSRAM_NO_SNOOP*. Once the system RAM buffer has been allocated, the mechanism initiates a copy operation of the chunk's contents into that buffer. When the copy operation is complete, the mechanism updates the GPU's page tables to point to the newly allocated buffer in system RAM. The GPU memory occupied by the chunk can then safely be reused.

Chunks can be transferred between CPU and GPU in two ways: By issuing a DMA operation to the GPU, or by mapping the chunk's GPU memory into the GPU driver's address space and executing a copy loop on the CPU. For GPUswap, we chose to perform DMA transfers on the GPU for two reasons: First, at GPUswap's typical chunk size of several MiB, the GPU can be expected to deliver better overall performance than the CPU [16]. Second, we expect DMA transfers to cause less interference than copying on the CPU since the GPU is able to execute DMA transfers asynchronously: Submitting DMA operations to the GPU's PCOPY engine allows the GPU to execute these operations in parallel with both GPU kernels and any work applications may perform on the CPU. While GPUswap's transfers may still compete with asynchronous DMA transfers initiated by applications, we expect PCOPY to be the least busy of the GPU's engines since applications should generally minimize DMA to achieve best performance [47, Section 5.3.1].

Our relocation mechanism assumes that there is sufficient system RAM available to hold all relocated chunks. We consider this assumption reasonable since current server machines typically contain much more system RAM than GPU memory: Recent Intel CPUs support up to 3 TiB of RAM per socket [28] while even the most high-end GPUs currently available are limited to 32 GiB of GPU memory [51]. The relocation mechanism can, however, only use system RAM that is compatible with DMA from the GPU. Fortunately, current GPUs are not particularly demanding when it comes to DMA memory: Recent Nvidia GPUs can access the lower 1 TiB of physical address space and support scatter-gather-DMA – i.e., can perform DMA operations on memory that is not physically contiguous. Therefore, the only requirement for the memory used to relocate chunks is that this memory must be non-pageable: Since memory holding relocated chunks is mapped in a GPU address space, paging out that memory to disk would create inconsistency between CPU and GPU. Since regular kernel memory fulfills all requirements described above, the relocation mechanism can simply use the kernel's physical

page allocator to obtain appropriate memory. Note that the relocation mechanism is not constrained by the kernel's virtual address space: Since the CPU never accesses relocated chunks, no virtual-to-physical mappings are required on the CPU side.

The eviction policy is generally free to select chunks from the buffer that triggered the relocation – in particular, the policy has no other choice than to select chunks from that buffer if the buffer is larger than the total amount of GPU memory. Since chunks from a buffer that is currently being allocated do not yet contain any data on the GPU, our relocation mechanism does not need to perform a DMA transfer for these chunks. Instead, the mechanism modifies the original allocation request to allocate the selected chunks directly in system RAM.

3.2.3 Returning Data to the GPU

Using system RAM in place of GPU memory always induces application overhead since accessing system RAM over PCIe is necessarily slower than accessing native GPU memory. Therefore, GPUswap should ensure that data is kept in GPU memory whenever possible. As a consequence, GPUswap never evicts any data from the GPU while there is still GPU memory available. However, applications can also free memory after data has been evicted from the GPU. To maintain good utilization of GPU memory at all times, it is therefore necessary to actively return data to the GPU to re-use any memory that has been freed.

To enable returning of data to the GPU, our eviction policy must be able to select not only pages for eviction, but also pages that should be returned to the GPU. While this makes our policy responsible for two distinct operations, these two operations share similar goals and require the same information about which chunks are frequently accessed by the application. When unused memory is detected on the GPU, GPUswap first calls the eviction policy, specifying how much data should be returned to the GPU. In response, the policy returns a set of evicted chunks that should be relocated back to GPU memory. The policy may return fewer chunks than requested if more GPU memory has been freed than was previously evicted.

After the policy has selected a set of chunks, GPUswap moves the contents of these chunks back to GPU memory in a manner similar to eviction: For each application for which the policy has returned at least one chunk, GPUswap first suspends GPU access, and then schedules asynchronous DMA transfers for each of the application's selected chunks. Once these transfers are complete, GPUswap updates the application's page tables such that the virtual address ranges occupied by the selected chunks point to the new locations of these chunks in GPU memory. Finally, GPUswap restores GPU access for the application, and moves on to the next application with chunks selected for return to the GPU.

Although returning data to the GPU is important, it is generally not as urgent as evicting data: If data must be evicted from the GPU, there is always an allocation request outstanding which cannot be serviced before the eviction is complete. In contrast, there is no reason for a deallocation operation to wait for data to be brought in. Instead of returning data to the GPU immediately on deallocation, GPUswap therefore handles return operations asynchronously in a separate thread which is part of the GPU driver. This thread periodically checks if there is unused memory on the GPU, and returns data to the GPU if there is. While checking for unused memory periodically leaves some GPU memory unused for a short period of time, this approach has the advantage of batching return operations. Since returning data to the GPU requires suspending GPU access for the application owning the data, returning memory to the GPU immediately would result in applications' GPU access being suspended frequently if an application performs many deallocation operations in short succession. Since we frequently observed this behavior in the GPU applications we examined, we assume that the performance benefit of suspending GPU access less often outweighs the performance penalty from leaving GPU memory unused for a short period of time.

3.2.4 Memory Accounting

Our accounting mechanism handles two main tasks: Keeping track of the amount of allocated and free GPU memory, and dividing allocated buffers into chunks. The mechanism is invoked after each allocation or deallocation request and after our background thread has returned data to the GPU.

The accounting mechanism tracks allocated GPU memory both globally and per application. Tracking allocated GPU memory globally allows GPUswap to determine efficiently whether or not an eviction is required before an allocation request can be serviced. Tracking allocated GPU memory for each application can be useful for the eviction policy: A policy trying to maintain fairness could, for example, evict data from the application currently owning most GPU memory. Keeping track of an amount of allocated memory is as simple as incrementing or decrementing a counter after an allocation or deallocation; however, the accounting mechanism must ensure that updates to the stored amounts reflect the state after all relocations that may have taken place before the accounting mechanism was invoked. For example, a single allocation request can cause the amount of allocated GPU memory to change for multiple applications if chunks from multiple address spaces are evicted as a result.

In addition to keeping track of allocated memory, our accounting mechanism is also responsible for dividing allocated buffers into chunks. The accounting mechanism maintains two **chunk lists** for each application using the GPU: One contains data about chunks currently in GPU memory, while the other holds data about chunks in system RAM. The entries in the chunk lists only contain metadata

about chunks – such as the chunk's base address and size – but not the data stored in the chunk.

Whenever a new buffer is allocated in GPU memory, the mechanism divides this buffer into chunks and adds appropriate entries to the GPU chunk list of the allocating application. If an eviction was required before the allocation could be served, the accounting mechanism also moves the entries of all evicted chunks from the corresponding application's GPU chunk list to its system RAM chunk list. Conversely, after the background thread returns chunks to the GPU, the accounting mechanism moves the entries of all returned chunks back to the application's GPU chunk list. Finally, after each deallocation, the accounting mechanism removes the entries of all freed chunks from the appropriate lists.

3.3 Prototype Implementation

To demonstrate the viability of our approach, we created a prototype of GPUswap. In this section, we describe the implementation of that prototype. First, we describe how we integrated this prototype with the GPU driver in Section 3.3.1. Then, we present implementation details of various components of GPUswap: We describe the relocation mechanism in Section 3.3.2, our mechanism for suspending GPU access in Section 3.3.3, and the accounting mechanism in Section 3.3.4. Finally, we discuss the limitations of our prototype in Section 3.3.5.

3.3.1 Driver Integration

Our prototype is implemented as part of the GPU driver. Integrating GPUswap into the driver allows us to take shortcuts which both simplify the implementation and improve performance, such as directly accessing the driver's internal data structures or adding hooks to existing code to call into GPUswap in appropriate locations. Overall, however, GPUswap is mostly an add-on to the existing driver, requiring little change to existing code. In case the driver's source code is not available, GPUswap can also be implemented as a separate kernel module in a manner similar to NEON [44].

We implemented our prototype as part of the PathScale NVIDIA Graphics Driver (pscnv) [53]. Pscnv is a fork of the popular Nouveau driver [65] and, to our knowledge, the only open-source driver capable of mapping command submission channels into user space. We chose pscnv since we wanted to demonstrate that GPUswap works even when applications have direct access to the GPU's command submission channels. Unfortunately, pscnv has been unmaintained since 2012 and therefore restricts our prototype to Nvidia Fermi-generation GPUs and Linux version 3.5.

3.3.2 Memory Relocation

To minimize the number of times each application's GPU access is suspended, our relocation mechanism performs chunk relocations one address space at a time. To that end, the mechanism first sorts the chunks received from the eviction policy by the GPU address space the chunks belong to. Then, the mechanism iterates over all GPU address spaces for which at least one chunk has been selected. For each of these address spaces, the relocation mechanism performs the steps described in Section 3.2.2: First, the mechanism suspends access to all command submission channels attached to the address space. Then, the mechanism copies all selected chunks from the address space to system RAM, and updates the address space's page tables. Finally, it restores access to all suspended channels. Returning memory to the GPU is implemented in much the same way, the only difference being that data is being copied to from system RAM to GPU memory.

The design of the pscnv driver presented a challenge in implementing our relocation mechanism: Since the mechanism should move data between CPU and GPU using asynchronous DMA operations, it must be able to submit appropriate commands to the GPU. However, pscnv does not include any code to submit commands to the GPU from kernel space. Instead, pscnv expects applications to map their command submission channels to user space and subsequently submit all GPU commands from there. To allow our relocation mechanism to submit the necessary DMA commands, we therefore ported the corresponding code from Gdev's user space libraries [36] into pscnv. Since Gdev's libraries are designed to interact with the GPU's command submission channels directly, this port required only small modifications to the code.

Our prototype uses a dedicated command submission channel attached to a dedicated virtual address space to handle DMA operations. Our modified pscnv driver allocates this DMA channel and the corresponding address space when it is loaded into the kernel, but never maps this DMA channel into user space. To relocate a chunk from GPU memory to system RAM or vice versa, our prototype then performs the following steps:

1. Allocate space at the chunk's new location
2. Map both the chunk's old and new location into the DMA address space
3. Queue a DMA operation into the DMA channel
4. Wait for the DMA operation to complete
5. Remove both mappings from the DMA address space
6. Change the mapping in the application address space containing the relocated chunk to map to the chunk's new location
7. Free the memory at the old location of the chunk

In principle, relocation could be implemented without a dedicated DMA channel by using a command submission channel in the address space of the application owning the chunk. However, submitting commands to a channel owned by an application may inadvertently result in changes to the state of that channel which would be visible to the application. Furthermore, using an application channel would also require a temporary mapping of the chunk's new location in the application's address space, which would complicate memory management since we must ensure that this temporary mapping does not collide with existing mappings. Therefore, we chose to implement relocation using a separate channel and address space to avoid these issues.

For ease of implementation, our current prototype does not use fully asynchronous DMA operations. Instead, our prototype submits DMA operations for one chunk at a time, waiting for the operation's completion before submitting the next DMA operation. Although this scheme renders tracking of multiple in-flight DMA operations unnecessary, it potentially prolongs the time each channel remains suspended since the setup of the next DMA transfer does not overlap with the execution of the current one. However, we do not consider this limitation a major issue since for chunk sizes of a few MiB, the time spent on setting up DMA transfers is small compared to the transfers themselves as shown in Section 6.3.2. Our prototype does submit all DMA operations to the PCOPY engine, rendering them asynchronous from the GPU's point of view. Relocation operations thus execute in parallel with kernels running in other address spaces, which ensures that no compute performance is lost to relocations.

3.3.3 Suspending Applications

To maintain data consistency, our relocation mechanism must ensure that applications do not change the contents of chunks during relocation. On the CPU, such changes can easily be prevented by mapping all pages in a chunk read-only until the relocation is complete. However, this method does not apply to current GPUs since these GPUs do not support page faults in the same way as CPUs: Any page fault caused by writing to a chunk during relocation would be treated as a fatal error and result in the immediate termination of the faulting GPU kernel. As a consequence, the relocation mechanism must prevent applications from executing any kernels in their address spaces while a relocation is in progress.

GPUswap employs shadow channels similar to those implemented in LoGV [20] to suspend GPU access for applications. For each command submission channel owned by the application to be suspended, GPUswap creates an identical copy of the channel's control registers in system RAM. This shadow copy is then mapped into the application's CPU address space at the same address at which the real control registers were previously mapped. From the application's point of view, the shadow copy is not only identical to the real registers, but can also be

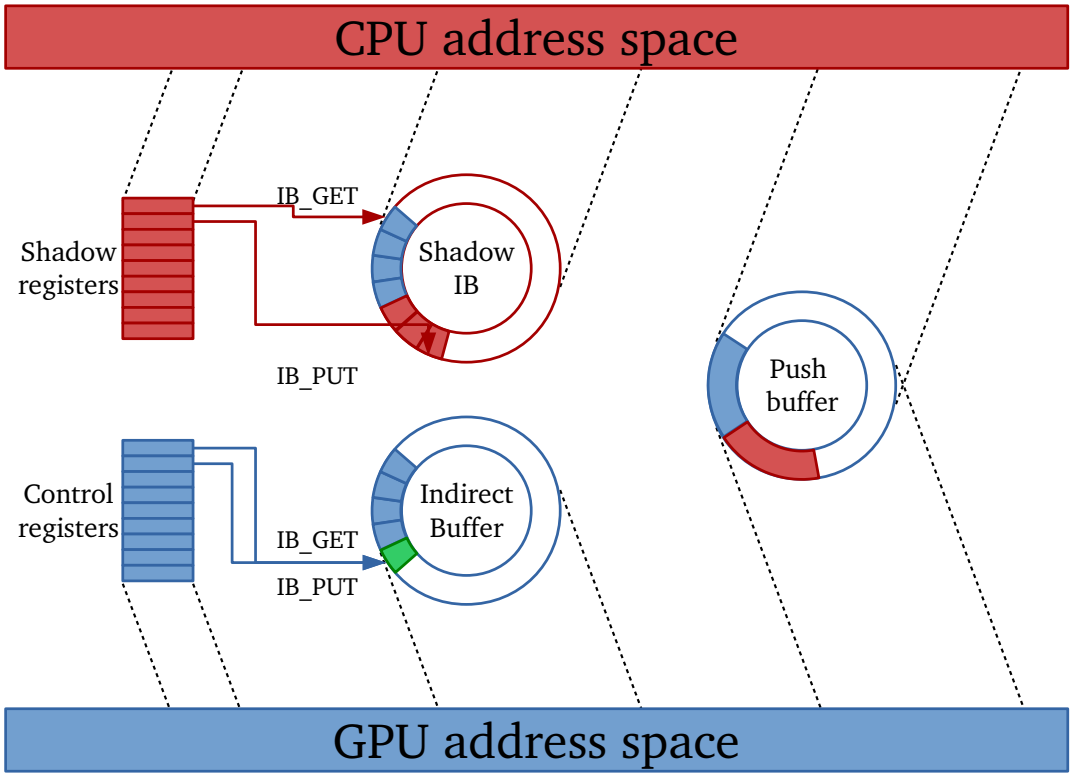


Figure 3.2: A suspended command submission channel. The application's push buffer (PB) is visible to both the application and the GPU, while the real indirect buffer (IB) and control registers are accessible only to the GPU, but have been replaced by shadow copies in the application's CPU address space. In this example, the application has submitted four GPU kernels which had not yet been executed at the time the application's GPU access was suspended. The real IB therefore contains four descriptor items pointing to the application's PB, and an additional descriptor item (shown in green) pointing to a PB in the driver's address space containing a fence command (not shown). All commands in the channel have been executed by the GPU. The shadow IB contains copies of the four descriptor items submitted prior to suspension, but the application is unaware that the corresponding commands in the PB have finished executing. In addition, the application has submitted three descriptors and a set of corresponding commands (shown in red) after its GPU access was suspended, which are not yet visible to the GPU.

freely modified. As a result, the application never experiences a page fault while submitting commands to the GPU – and is thus never prevented from executing code on the CPU – but the submitted commands do not execute on the GPU until the application's GPU access is restored. To restore the application's GPU access, GPUswap copies the contents of the shadow copy to the real control registers – which causes all commands that were submitted to the suspended channel to become visible to the GPU simultaneously – and restores the original mapping to the real registers.

Since command submission channels act as queues for commands awaiting execution, there may be commands already queued in the application's channels at the time the application's GPU access is suspended. Current GPUs offer no reliable way to un-queue commands after these commands have been written into a channel, but instead expect all commands in a channel to eventually execute to completion. Before our relocation mechanism can begin relocating chunks, the mechanism must therefore wait for any commands that were already queued in the application's channels at the time the application's GPU access was suspended to finish execution. To determine when it is safe to begin relocation, GPUswap submits a fence command to each suspended command submission channel. Whenever one of these fence commands is executed, the GPU raises and interrupt to the GPU driver. Since commands in a channel always execute in order, relocation can begin as soon as GPUswap's fence commands have been executed in all suspended channels. Note that GPUswap may receive additional interrupts while waiting for its fence commands to be executed since applications may use fence commands as well. On each interrupt, GPUswap must therefore check whether that interrupt was caused by one of its own fence commands.

While fence commands provide an efficient way to detect empty channels, submitting commands to suspended channels has the potential to violate our goal of operating transparently: Since the channel's indirect buffer (IB) is directly accessible to the application owning the channel, the fence command is visible to the application, and may even be overwritten if the application submits commands while its GPU access is suspended. To preserve transparency, GPUswap thus shadows the channels' IBs in the same way as the control registers to ensure that the fence command is never visible to the application. Shadowing the push buffer (PB) is unnecessary: GPUswap allocates a private PB for fence commands in kernel space and insert pointers to this private PB into the channel's IB. Figure 3.2 shows an example of a shadowed command submission channel after GPU access has been suspended for the application owning the channel.

Since submitting the fence command also changes the values of the channel's control registers, GPUswap must reset the control registers to a state that appears like the fence command was never submitted before restoring the application's GPU access. Since the GPU does not allow direct writes to IB_GET, GPUswap implements this reset by filling the channel's entire IB with pointers to a PB

containing a single NOP command and subsequently updating IB_PUT to point to the IB slot right before the fence command. As a result, IB_GET wraps around and eventually reaches the position expected by the application. Once IB_GET has reached its original position, GPUswap copies the contents of the IB's shadow copy to the real IB and restores the application's mapping to the real IB before restoring access to the control registers.

Since creating shadow copies of both the control registers and the IB is not atomic, it is possible for the application to submit commands while the shadow copy is being created. To ensure consistency, GPUswap unmaps both the control registers and the IB before creating the corresponding shadow copies. If the application subsequently writes to either the registers or the IB while the shadow copy's creation is still in progress, GPUswap stops the application's execution in the page fault handler until the process is complete. While this scheme may prevent the application from executing CPU code for short periods of time, this period is typically much shorter than the subsequent relocation.

3.3.4 Memory Accounting

Our accounting mechanism is responsible for tracking the amount of allocated memory and for dividing newly allocated application buffers into chunks. To track allocated memory, the accounting mechanism maintains a global variable for the total amount of allocated memory. Furthermore, GPUswap adds a field to pscnv's address space data structure which holds the total amount of memory allocated in that address space. The accounting mechanism increments the appropriate variables whenever an application allocates GPU memory, but ignores allocation requests if the application specifically requests system RAM.

Whenever an application allocates a buffer in GPU memory, our accounting mechanism divides the allocated buffer into chunks. To that end, GPUswap extends pscnv's internal buffer object (BO) data structure – which represents a single application buffer – by adding an array of chunk descriptors. Each of these descriptors holds metadata about a single chunk belonging to the buffer, such as whether or not the chunk is currently in system RAM, a pointer to the BO data structure and information about the physical pages backing the chunk.

Storing information about physical pages is not trivial since application buffers are typically not physically contiguous. If each chunk is larger than one page, the accounting mechanism must thus store a physical address for each page in the chunk. Pscnv's memory allocator normally stores this information in each BO as a linked list of physically contiguous segments. To leverage this data, we modified the memory allocator to instead attach a linked list of segments to each chunk descriptor as depicted in Figure 3.3.

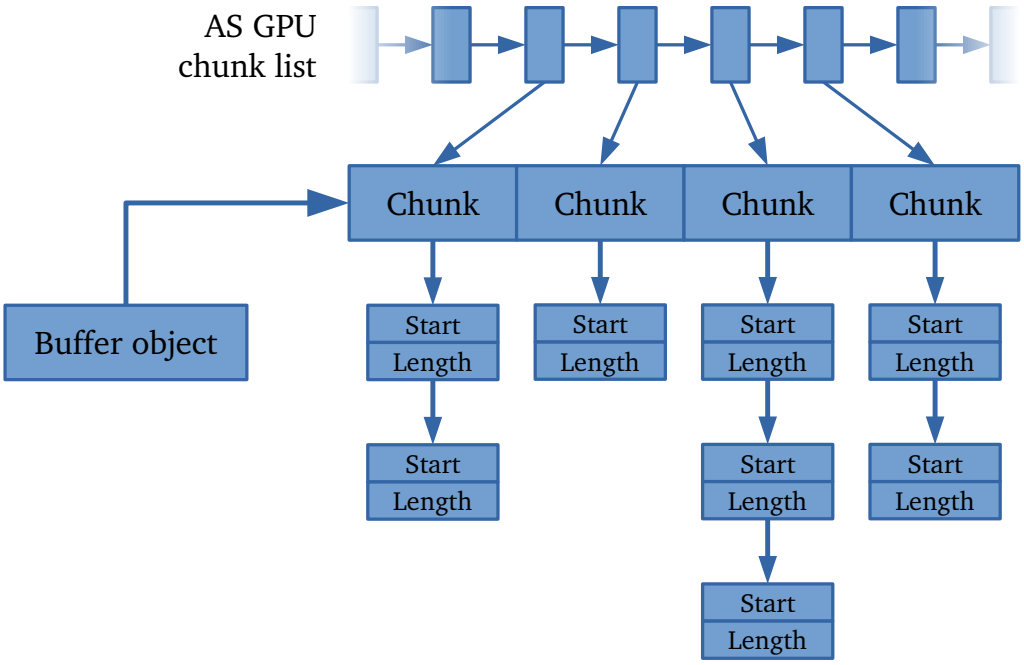


Figure 3.3: Storing chunk metadata in GPUswap. Each buffer object (BO) data structure contains an array of chunk descriptors. In addition, GPUswap maintains two linked lists for each GPU address space: One containing pointers to the descriptors of all chunks currently residing in GPU memory, and one containing pointers to all descriptors of evicted chunks (not shown). Each chunk descriptor contains a linked list of segment descriptors holding information about the physical memory backing the chunk. Each segment descriptor contains a starting address and a length, thus describing a physically-contiguous region of memory. Note that even though all chunks normally share the same size, the size of the physical segments is not fixed, leading to a variable number of segments per chunk.

In addition to the array of chunks in each buffer, our accounting mechanism maintains two linked lists for each address space: One, which is shown in Figure 3.3, stores pointers to the descriptors of all chunks allocated in GPU memory inside the address space, while the other holds pointers to the descriptors of all chunks in the address space that have been relocated to system RAM. Whenever a new buffer is allocated, the accounting mechanism appends pointers to all chunk descriptors in the corresponding BO's chunk array into the appropriate chunk lists[1] of the address space containing the buffer. Since GPUswap only stores pointers in the two chunk lists, it is not necessary to copy entire descriptor items between the two lists after chunks have been evicted from or returned to GPU memory. Furthermore, both lists are unsorted, which speeds up allocations by allowing the

[1] Note that the eviction policy may decide to allocate some chunks directly in system RAM.

accounting mechanism to simply append pointers to the newly allocated chunks to the list.

Our accounting mechanism currently ignores all buffers allocated by the GPU driver for its own use, such as command submission channels. These buffers are marked with a special flag in their BO structure, which makes them easy to detect. Although some of these buffers can in principle be allocated in system RAM, it is undocumented which data structures are required to be in GPU memory. In addition, these buffers are typically small compared to buffers containing application data. We therefore decided to err on the side of caution by keeping these data structures in GPU memory.

3.3.5 Limitations

Our current prototype relies heavily on the pscnv driver. As a consequence, since pscnv has been unmaintained for a long time, the prototype is limited to GPUs and a version of the Linux kernel that were state of the art when pscnv was developed. Specifically, pscnv – and hence our prototype – supports GPUs up to the Nvidia Fermi generation, and Linux up to version 3.5. However, these limitations are not conceptual ones: GPUswap can be integrated into any GPU driver as long as that driver's source code is available. Furthermore, GPUswap's design is not limited to Nvidia GPUs: In principle, GPUswap only requires the GPU to support mapping of system RAM into its address spaces, which is fulfilled by all recent Nvidia and AMD GPUs. Note that the ability to map command submission channels into user space is not a requirement for GPUswap. In fact, hiding command submission channels from applications would simplify our prototype by rendering the implementation of shadow channels unnecessary: If applications would have to call into the GPU driver to submit commands, GPUswap would know about each submitted command and could thus delay the execution of commands from suspended applications. Nevertheless, GPUswap should support applications accessing their command submission channels directly since calling into the driver on each command submission would cause non-trivial system call overhead. However, GPUswap's shadow channels can be applied to any GPU that supports mapping of command submission channels into user space.

Even though GPUswap is intended to be application-agnostic, our prototype implementation is limited to CUDA applications. The cause of this problem is the fact that the only user space runtime supporting pscnv is that of Gdev. Besides Gdev's own interface, this runtime supports the CUDA driver API, but not OpenCL or accelerated 3D rendering. However, this limitation is not a conceptual one either: Our prototype itself makes no assumptions about the nature of the applications using the GPU. Specifically, our implementation of shadow channels is oblivious to the commands submitted by the application, and our relocation

mechanism does not need to know about the contents of evicted chunks. In principle, GPUswap thus works with any GPU application.

The final major limitation of our prototype is that it does not yet properly handle buffers in GPU memory which are mapped into an application's CPU address space. This type of mapping is used for command submission channels – i.e., the IB and PB – but applications can explicitly set up such mappings as well if needed, for example to exchange data between CPU and GPU. Although GPUswap can evict chunks from these buffers just like any other chunk, our prototype does not currently update the CPU's page tables after eviction. Evicting a GPU buffer mapped on the CPU thus causes CPU and GPU to disagree about the physical location of the chunk after eviction. In our prototype, we did not implement support for such mappings since none of the applications we examined use this type of mapping for anything other than their command submission channels, which our prototype currently ignores anyway.

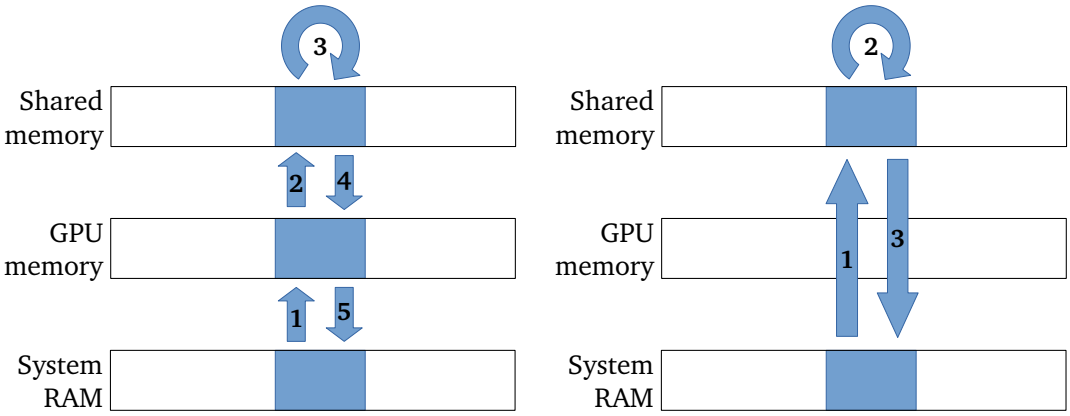
3.4 Discussion

In this Section, we revisit the design goals laid our in Section 3.1 and discuss how these goals are fulfilled by our design. In addition, we discuss the limitations of our design compared to previous work, as well as the impact of those limitations on GPU applications. In Section 3.4.1, we address the generality and transparency of GPUswap, which are closely related. Finally, we discuss the performance impact of GPUswap on applications in Section 3.4.2.

3.4.1 Generality and Transparency

GPUswap fulfills the generality requirement since our design makes no assumptions about the application's behavior: Our shadow channels are oblivious to the nature of the commands submitted into the applications' command submission channels, and our relocation mechanism only makes identical copies of relocated pages, but never interprets the contents of those pages. Therefore, even though our prototype is limited to CUDA due to pscnv, GPUswap is, in principle, compatible with any type of GPU application.

Overall, we consider our requirement of transparent operation to be fulfilled as well: Neither our shadow channels nor our relocation mechanism change the contents of the application's virtual address space from the application's point of view. As a result, the only aspect of GPUswap's operation that is visible to the application at all is its overhead: Applications could detect the presence of GPUswap by measuring the performance of their memory, which would show a noticable drop as soon as GPUswap starts evicting memory to system RAM.



(a) Data transfers with Gdev.
1. Gdev pulls buffer into GPU memory prior to kernel launch.
2. GPU kernel loads data from buffer into shared memory.
3. GPU kernel performs computation on data in shared memory.
4. GPU kernel writes results back to GPU memory.
5. Gdev evicts buffer when starting another application's kernel.

(b) Data transfers with GPUswap.
1. GPU kernel loads data into shared memory, bypassing GPU memory.
2. GPU kernel performs computation on data in shared memory.
3. GPU kernel writes results back to system RAM.

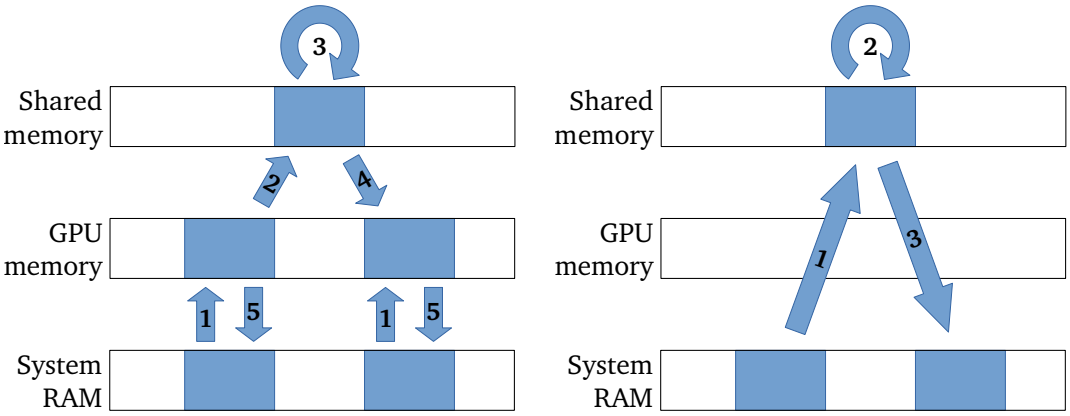
Figure 3.4: Data transfers performed during in-place computation with Gdev and GPUswap

However, GPUswap does not require applications to be modified or to otherwise cooperate in any way, and is capable of relocating memory even against the will of faulty or malicious applications.

Even though our design achieves both generality and transparency, it is important to note that the fulfillment of both goals also depends on the capabilities of the GPU in use: The memory of current GPUs supports several storage types, but it is undocumented whether all of these storage types are supported on pages backed by system RAM. While GPUswap is capable of evicting pages regardless of storage type, evicting data using a storage type that is only supported on GPU memory pages might thus lead to application errors.

3.4.2 Performance

The performance of applications running on a GPUswap-enabled driver depends heavily on whether or not there is sufficient GPU memory available. When sufficient GPU memory is available, we consider our goal fulfilled since, in contrast to previous work, GPUswap never performs any work when an application launches



(a) Data transfers with Gdev.
1. Gdev pulls both buffers into GPU memory prior to kernel launch.
2. GPU kernel loads data from input buffer into shared memory.
3. GPU kernel performs computation on data in shared memory.
4. GPU kernel writes results back to output buffer in GPU memory.
5. Gdev evicts both buffers when starting another application's kernel.

(b) Data transfers with GPUswap.
1. GPU kernel loads input data into shared memory, bypassing GPU memory.
2. GPU kernel performs computation on data in shared memory.
3. GPU kernel writes results back to output buffer in system RAM.

Figure 3.5: Data transfers performed with Gdev and GPUswap when using separate buffers for input and output

a GPU kernel. As long as sufficient GPU memory is available, GPUswap thus induces only negligible overhead on memory allocation requests due to memory accounting.

In the presence of memory contention, we expect GPUswap to induce considerable overhead. However, this problem is not singular to GPUswap: Since system RAM is slower than GPU memory, overhead is unavoidable whenever GPU data is evicted to system RAM, regardless of the method used for eviction.

In contrast to previous work, GPUswap only transfers evicted data over the PCIe bus if a GPU kernel actually accesses that data. On the one hand, this approach has the obvious advantage of avoiding unnecessary data transfers – data that is never accessed is never transferred over the PCIe bus. On the other hand, however, transferring data on access may lead to the same piece of data being transferred multiple times if that data is accessed more than once.

Despite the possibility of repeated transfers, we argue that GPUswap does not typically transfer larger amounts of data than previous work since in practise, this type of repeated transfer is rare. Most GPU applications rely heavily on the GPU's shared memory for performance. As a result, each GPU kernel typically accesses

data in GPU memory twice: Once to bring its input data into shared memory, and once more to write the result back into GPU memory after the kernel has finished its computation[1]. For applications performing in-place computation – i.e., writing the result of their computation into the same buffer that contained the input data – GPUswap thus performs the same number of PCIe transfers as Gdev, as depicted in Figure 3.4: With GPUswap, GPU threads load evicted data from system RAM directly into the GPU's shared memory, while Gdev transfers the evicted buffers to GPU memory prior to kernel launch, and likely evicts them again after the kernel finishes execution. As a result, GPUswap may actually be at an advantage since the application does not need additional accesses to GPU memory to bring evicted data into shared memory.

For applications using different buffers for input and output, GPUswap's advantage is even larger, as shown in Figure 3.5: If the application reads its input from one evicted buffer and writes the result into another, GPUswap transfers each buffer's contents over PCIe once. In comparison, Gdev requires twice as many PCIe transfers: Both buffers must be transferred to the GPU prior to kernel launch and may be evicted again after the kernel finishes execution.

Gdev is able to schedule a single, large DMA transfer for each buffer it must move between GPU memory and system RAM, while GPUswap forces the GPU to perform one PCIe bus transaction for each access from the GPU to evicted chunks. However, while one large transfer is superior to many small ones in terms of performance, Gdev's approach also requires the application to wait for all evicted buffers to be returned to the GPU before starting any GPU kernels. In contrast, GPUswap allows each GPU thread to resume computation as soon as the data the thread requires has been transferred to the GPU. As a result, many of the small transfers caused by GPUswap overlap with computation, which hides part of the overhead associated with large numbers of small transfers.

[1] It is possible for GPU threads from different warps to copy the same data into the shared memory of multiple SMs. However, the GPU's cache tends to coalesce such accesses into a single PCIe transfer: The first thread to access the data pulls that data into the GPU's L2 cache, which is shared by all SMs. Subsequent accesses from different threads then read the data from that cache without the need for a PCIe transfer.

Chapter 4

Profiling Memory Access Patterns of GPU Applications

The mechanism described in the previous chapter must be accompanied by an eviction policy to be effective. While a wide variety of algorithms for making eviction decisions exists in the CPU world [60], these algorithms typically rely on hardware features like reference bits or page faults to find frequently-accessed memory pages. Since these features are not available in current GPUs, profiling is currently the only viable way to gather information about the access frequency of GPU memory pages.

Memory profiling in previous work [4] relied on compiler modification, which induces two major limitations: First, profiling is restricted to the types of applications – e.g., CUDA or OpenCL – that are supported by the modified compiler. Second, only code which has been compiled using the modified compiler can be profiled, rendering memory accesses originating from shared libraries used by the application invisible to the profiler.

To overcome these issues, we developed a profiling mechanism for GPU applications based on the GPU's performance monitoring counters. Our profiling mechanism complements the existing work in two ways: First, we confirm the findings of previous work using a different method of measurement. Second, our method is able to account for some memory accesses which were invisible to previous work, leading to additional insights into application behavior.

4.1 Goals

Since the performance monitoring counters of current GPUs can only count events from one application at a time, our profiler is intended for off-line analysis of GPU applications. Profiling should provide detailed information about the

memory accesses of the analyzed application, without imposing restrictions on that application. Specifically, our profiler should fulfill the following goals:

Accuracy The data generated by our profiler is ultimately intended for use in eviction decisions. Since we can only evict data from the GPU with page granularity, the profiler should thus be able to count the application's memory accesses with page granularity as well. In contrast, a finer granularity is unlikely to yield additional insights. Furthermore, our profiler should be able to separate different types of access, such as read and write accesses, since different types of access to evicted data may impose different penalties on the application's performance.

Transparency Like GPUswap, our profiler should not require the profiled application to explicitly cooperate. Explicit cooperation would require the application to be modified, which is often not possible if the application's source code is not available. In addition, uncooperative applications could deceive the profiler about their memory accesses. As a consequence, we strive to keep the profiler's operation transparent to the application.

Generality Ideally, our profiler should support any type of GPU application out of the box. To use the performance monitoring counters of current GPUs, however, these counters must be explicitly enabled during GPU kernel launches by setting a flag in the command submission channel of the profiled application. As a consequence, a profiler using the performance monitoring counters must detect kernel launches of the profiled application and set the profiling flag itself if profiling is to be transparent. Detecting kernel launches in turn requires information about the application being profiled since different types of application may perform kernel launches differently. While a profiler using the performance monitoring counters thus cannot be fully application-agnostic, the design of our profiler should nonetheless remain as independent as possible from any specific type of application.

Note that we do not consider performance a primary goal of our profiler: The profiler is intended for off-line use and is thus unlikely to be used in a production environment. In addition, profiling is a relatively rare event: Given the profiling data is representative of the application's execution, data collected once can subsequently be reused many times. We therefore consider it generally acceptable if a profiler causes significant application overhead.

4.2 Design

Our profiling mechanism uses the GPU's performance monitoring counters to remain oblivious to the application being profiled: Peformance monitoring counters

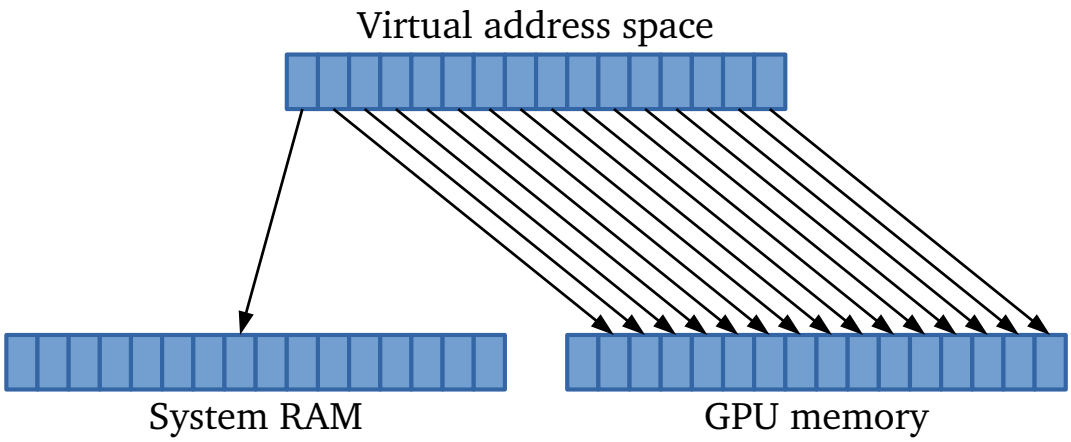


Figure 4.1: Separation of a page from the rest of its address space for profiling. In this example, the first page in the application's address space has been relocated to system RAM. Since no other pages from the address space are in system RAM, the total number of system RAM accesses equals the number of accesses to the relocated page. In the next iteration, the second page from the buffer will be in system RAM.

are not only compatible with any type of application, but also indiscriminately account for all memory accesses from the application, whether these accesses originate from the main application code or a library linked into the application. However, the GPU's performance monitoring counters were designed to monitor events for applications as a whole and can therefore not count accesses to individual memory pages. Our profiling mechanism must work around this limitation to obtain detailed information about the access patterns within an application's address space.

In this section, we present our design for a profiling mechanism which uses the GPU's performance monitoring counters to obtain access counts for individual pages in a transparent way. We give an overview of our design in Section 4.2.1, followed by detailed descriptions of the individual components of our profiler in the subsequent sections.

4.2.1 Overview

Out of the box, the GPU's performance monitoring counters can only count memory accesses for an entire GPU address space, but not for individual pages within that address space. To obtain an access count for an individual page, we must therefore separate that page from the rest of the address space in such a way that all events of a certain type are caused by accesses to that page. Our profiler achieves this separation by relocating individual pages to system RAM while ensuring that all other application data is in GPU memory, as depicted in Figure 4.1. Since only one

page from the application's address space is in system RAM at the same time, the total number of accesses to the relocated page equals the total number of accesses to system RAM from the application's address space, which is easily determined through the performance monitoring counters.

To obtain a complete access profile for the application's entire address space, our profiler uses this method on each page in the address space: Whenever the application being profiled launches a GPU kernel, the profiler repeats the kernel's execution once for each page in the application's address space, relocating a different page to system RAM in each repetition. After all pages in the application's address space have been processed in this way, this approach yields an accurate number of accesses for each page.

The main advantage of using the performance monitoring counters for profiling is that these counters require little knowledge about the application being profiled. Specifically, the counters do not require the application to be instrumented and are oblivious to shared libraries used by the application. In principle, our profiler is therefore not restricted to a specific type of application: While our profiler cannot remain fully application-agnostic since it must detect the application's kernel launches to enable the performance monitoring counters, the majority of the profiler's code can operate on any type of application. Specifically, the operation of the performance monitoring counters and the mechanism for relocating pages to system RAM are application-independent, while the code for detecting and repeating kernel launches – which is specific to the GPU runtime library in use – is comparatively small and easy to adapt to new types of applications.

Since each of the application's GPU kernels must be repeated once for each page in the application's address space, profiling an application using a large amount of memory can easily require millions of repetitions, taking a considerable amount of time. However, we do not consider profiling time to be a serious issue since we expect profiling to be a rare event: Given that profiling yields representative results, the application must only be profiled once, and the results can subsequently be reused many times. In addition, it is often possible to profile an application using a smaller input data set than will be used later in production [40], which speeds up the profiling process considerably. Finally, a coarser profiling granularity may be acceptable in some cases – GPUswap, for example, typically operates on chunks larger than one page and therefore does not benefit from a granularity smaller than its chunk size. In such cases, our profiler can easily trade accuracy for performance by relocating more than one page to system RAM in each repetition.

Compared to previous work [4], which relies on compiler modification, our approach has two main advantages: First, a profiling mechanism based on compiler modification is inherently limited to those applications that can be handled by the modified compiler – e.g., CUDA applications. In contrast, our profiling mechanism is capable of profiling arbitrary applications. Second, compiler-based profiling

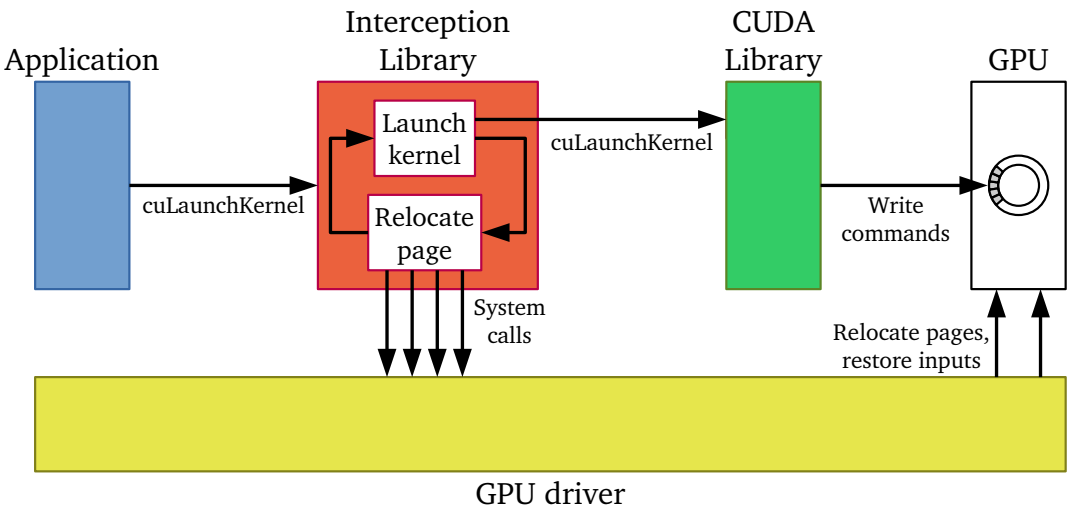


Figure 4.2: Repeating GPU kernel execution using an interception library. The profiler injects the interception library between the GPU runtime library – in this example, a CUDA library – and the application. The interception library then intercepts GPU kernel launches from the application and transparently repeats each kernel's execution once for each page in the application's address space. Between repetitions, the library issues the necessary system calls to restore the kernel's inputs, to read and reset the performance monitoring counters and to relocate the next page.

cannot profile memory accesses originating from libraries used by the profiled application unless these libraries were compiled by the modified compiler as well. In contrast, our work automatically covers memory accesses from all code running in the application's address space, including any shared libraries.

The main disadvantage of our approach is the time required for profiling: Our profiler must repeat the execution of each GPU kernel launched by the profiled application many times to collect a complete profile of the application's entire address space, which can take a considerable amount of time. While we cannot directly compare our approach to previous work since previous work did not report any numbers related to profiling time, we assume that previous work can collect access profiles much faster than our profiler since a compiler-based approach does not require repeating of GPU kernels.

4.2.2 Repeating Kernel Runs

To obtain a complete memory access profile for an entire address space, our profiler repeats the execution of each GPU kernel launched by the profiled application once for each page in the application's address space. To achieve this repetition,

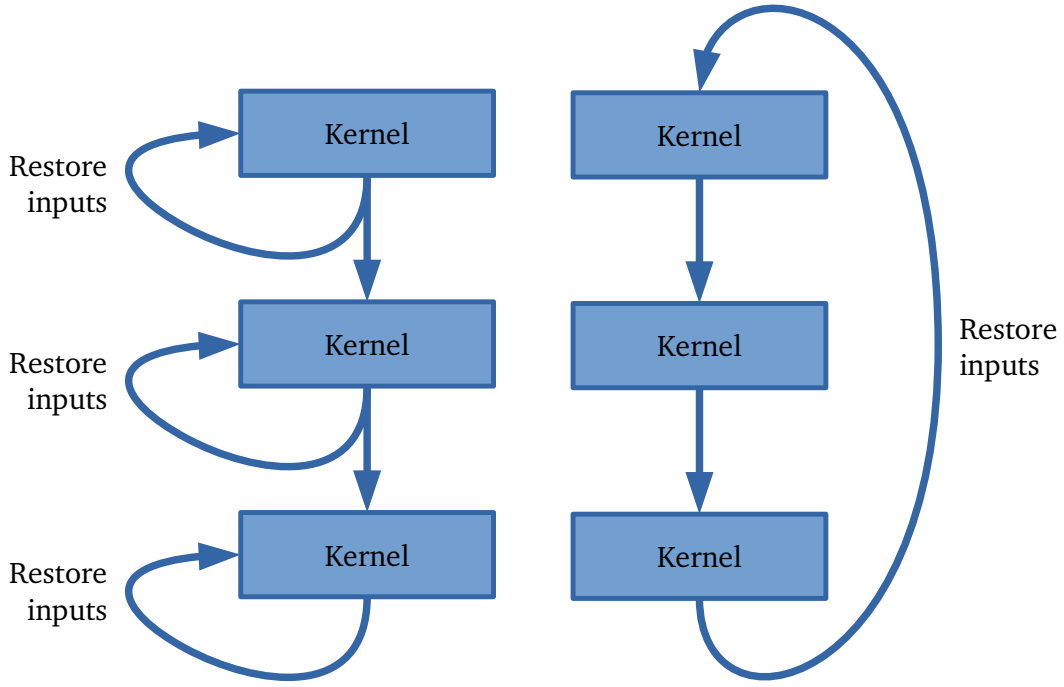
our profiler wraps each call to the GPU runtime library that launches kernels in a **kernel repetition loop**.

In our experiments, we frequently observed GPU kernels storing their result in the same buffer that previously held the input data. Without remedy, this behavior would cause each repetition to run on different input data, which would render the collected profiles meaningless. In addition to repeating kernel executions, our kernel repetition loop therefore restores each kernel's original input data between repetitions to ensure that the kernel's behavior is deterministic across iterations.

In total, our kernel repetition loop performs the following steps on each kernel launch:

1. Make a copy of all data accessible to the kernel and store that copy in system RAM to enable restoration of the original values between repetitions.
2. Relocate a page from the application's address space to system RAM as shown in Figure 4.1, mapping the relocated page to the same virtual address it originated from. Note that the page relocated in this step is different in each repetition.
3. Setup the performance monitoring counters to count all accesses to system RAM from the application.
4. Call into the original GPU runtime library to launch the application's GPU kernel.
5. Read the values of the performance monitoring counters and store them in a result file.
6. Return the page that was relocated in step 2 to GPU memory.
7. Check if all pages in the application's address space have been processed. If so, return to the application.
8. Otherwise, copy the application data saved in step 1 from system RAM back into GPU memory.
9. Return to step 2.

The kernel repetition loop can be implemented in two ways: Either by modifying the application's source code, or by injecting an **interception library** between the application and the GPU runtime library as depicted in Figure 4.2. The main advantage of using an interception library is that no access to the application's source code is required since the library's operation is completely transparent to the application. On the other hand, however, modifying the application has the advantage of allowing the kernel repetition loop to leverage application knowledge to reduce the amount of I/O required: By analyzing the application's source code, the developer can determine which buffers are modified during each



(a) Interception library (b) Integrated kernel repetition loop

Figure 4.3: Restoring kernel inputs during profiling for our interception library and with the kernel repetition loop integrated into the application. In this example, the application launches three kernels. Since the interception library does not know about the relationships between these kernels, it must profile each kernel individually. This implies that each kernel's input must be restored once for each page in the application's address space. In contrast, a kernel repetition loop integrated into the application can execute multiple kernels in a row before restoring the inputs of all of them, thus reducing the amount of DMA by a factor of three.

kernel's execution. With this information, the developer can then adapt the kernel repetition loop to only restore the contents of these modified buffers between iterations. In addition, if the application executes multiple kernels to compute a single result, the kernel repetition loop can execute the entire chain of kernels in each iteration as shown in Figure 4.3, only restoring the inputs of the first kernel after the last one has finished execution. Compared to an interception library, these measures can speed up the profiling process considerably.

Our approach builds on the assumption that each kernel's execution is sufficiently deterministic to perform the same memory accesses in each repetition. This assumption holds well since most GPU kernels compute labor-intensive mathematical functions, which, given the same input data, tend to perform the exact same computation every time. Note that our profiler only requires the number,

but not the exact timing of memory accesses to be deterministic: Since the GPU's performance monitoring counters cannot account for the exact time when each access took place, changes in the timing of memory accesses will not affect the resulting access profile.

Since our approach relies on injecting code into the application, our profiler is not fully application-agnostic since the injected code is always specific to the GPU runtime library used by the application. However, the amount of code actually injected is relatively small: Since each step of the kernel repetition loop is generic in nature – for example, relocating a page to system RAM works the same way for both CUDA and OpenGL-based applications – we can implement each step of the loop as a call into the GPU driver, which is then reusable for all types of applications. Beside the kernel repetition loop, our profiler generally requires no modifications to other parts of the application: Since the input data is not restored after the last repetition of each kernel, the application's remaining code is under the impression that the kernel executed as expected. As a consequence, only a small part of our profiler is actually application-dependent, and we expect this part to be easily adaptable to different applications or GPU runtimes.

4.2.3 Separating Pages

To count memory accesses for individual pages using the GPU's performance monitoring counters, our profiler separates these pages from the rest of the application's address space by relocating them to system RAM. To achieve this relocation, the profiler reuses the existing relocation mechanism of GPUswap, which we describe in Section 3.2.2.

Since the chunk size used by GPUswap is configurable, using GPUswap's relocation mechanism has the advantage of allowing the user to easily configure the profiling granularity. If fine-grained profiling is desired, the size of each chunk can easily be configured to one page. Conversely, a user wishing to speed up the profiling process can easily configure a larger chunk size, leading to fewer repetitions of each kernel. However, it is important to note that our relocation mechanism will never put multiple buffers in a single chunk: If the chunk size is larger than one of the application's buffers, the entire buffer will become a single remainder chunk. The coarsest granularity supported by our profiler is thus determined by the size of the application's buffers.

The only required modification to GPUswap's original relocation mechanism concerns the method of its invocation: GPUswap invokes its relocation mechanism transparently if memory contention is detected, whereas our profiler must be able to invoke the relocation mechanism explicitly from the kernel repetition loop inside the profiled application or the interception library. To allow the profiler to invoke page relocation explicitly, we modify GPUswap's original design by adding

a system call for explicit relocation. Using this system call, the profiler passes an index into the application's chunk list to GPUswap. GPUswap then relocates the chunk referenced by the corresponding chunk list entry to system RAM, while at the same time returning all previously relocated chunks to the GPU. The call then returns the virtual address of the relocated chunk to allow the profiler to map each chunk to an application data structure. If the application passes a number for which no entry exists in the application's chunk list, the call returns an error.

Our relocation system call identifies pages using chunk list indices instead of virtual addresses to simplify the kernel repetition loop inside the application: The loop can simply increment the chunk index in each iteration until the system call returns an error, at which point all pages in the application's address space have been processed. As a result, the kernel repetition loop does not require information about the application's address space layout: Unallocated regions in the application's address space are skipped automatically since the chunk list only contains entries for allocated pages, and no knowledge about the amount of allocated memory in the application's address space is required since the loop exits automatically once all pages have been processed. In addition, using chunk list indices causes our profiler to ignore all buffers which cannot be relocated to system RAM at runtime – such as those allocated by the GPU driver – since GPUswap never includes these buffers in its chunk list to begin with.

Since GPUswap's chunk list is unordered, our profiler does not guarantee that the pages in the profiled application's address space are processed in any particular order. However, since our relocation system call returns the virtual address of the relocated page, the profiler can match each page to its location in the application's address space regardless of ordering.

4.3 Prototype Implementation

To demonstrate the viability of our approach as well as to gain insight into the memory usage of GPU applications, we created a prototype implementation of our profiler. The largest part of that prototype is integrated into our pre-existing prototype of GPUswap. Specifically, our prototype reuses GPUswap's existing chunk relocation code, but augments that code with an API allowing the kernel repetition loop to save and restore kernel input data and to explicitly invoke chunk relocation.

In this section, we present details about the prototype implementation of our profiler. First, we explain how to guarantee deterministic kernel execution across repetitions in Section 4.3.1. In Section 4.3.2, we then discuss relocation of individual pages to system RAM. Finally, we present our handling of the GPU's performance monitoring counters in Section 4.3.3.

4.3.1 Repeating Kernel Runs

Our prototype uses an interception library to wrap each of the application's kernel launches in a kernel repetition loop. This library intercepts all calls to the CUDA API which either allocate memory or launch GPU kernels. By intercepting memory allocation calls, our library collects information about the location and size of all GPU memory buffers allocated by the application. When the application subsequently launches a GPU kernel, the library uses CUDA's existing API to save the contents of each of the application's buffers to system RAM, and subsequently restores each buffer's contents between iterations of the kernel repetition loop.

For comparison, we also created modified versions of our benchmark applications with the kernel repetition loop integrated directly into the application's source code. In these versions, we made heavy use of application knowledge to reduce the amount of I/O required: The kernel repetition loop integrated into these applications restores only the contents of modified buffers between iterations and executes as many kernels as possible before restoring any inputs.

4.3.2 Separating Pages

To separate pages from the rest of the application's address space, our profiler provides an API call for moving individual pages to system RAM. Our kernel repetition loop invokes this API call in each iteration to ensure that a different page is evicted in each repetition of the profiled GPU kernel. The call takes an index the profiled application's chunk list as an argument, and triggers GPUswap's pre-existing eviction mechanism on the chunk referenced by that index. At the same time, the call unconditionally returns all other chunks in system RAM to GPU memory.

For profiling to yield meaningful results, the contents of the application's chunk list must not change during profiling of a single kernel. As a consequence, applications cannot be allowed to allocate or free memory while our kernel repetition loop is running: Allocating or freeing memory would add or remove entries in the application's chunk list, changing the index of other chunks in the process. Since the kernel repetition loop simply increments a counter in each repetition, changing chunk indices while the loop is running could result in some chunks being missed or profiled twice under different indices. For single-threaded applications, our profiler mitigates this issue by wrapping the kernel repetition loop tightly around kernel launch operations: If the loop does not contain any operation other than calls into the kernel part of our profiler and the actual kernel launch, the application is guaranteed not to allocate or free any memory before the loop has finished running.

Multithreaded GPU applications present two additional challenges to our profiler. First, a multithreaded application could allocate or free memory from a different thread during profiling, which without our profiler would be safe to do as long as the application does not free any memory that is touched by the running kernel. Second, a multithreaded application may launch multiple kernels in parallel, causing our profiler to relocate two pages to system RAM at the same time with no way of telling which page accumulated how many accesses. To mitigate these issues, our profiler introduces a lock which must be held during the entire runtime of the kernel repetition loop as well as during any operation changing GPUswap's chunk list. In essence, this lock forces the profiled application to behave as if it was single threaded: Any thread attempting to launch a kernel or allocate or free memory while a kernel is being profiled is stopped until the running kernel repetition loop completes, which guarantees consistent profiling results.

4.3.3 The Performance Monitoring Counters

Our current prototype does not implement separate system calls to handle the GPU's performance monitoring counters. Instead, our interception library maps the counters' control registers into user space and subsequently accesses these registers directly: Before launching the profiled kernel, the loop iterates over all memory-related domain sets, configuring the first domain in each set to account for the desired type of event and resetting the domains' counter values to zero. Whenever a kernel finishes execution, the loop sums up the counter values of the first domain in each set and stores the final sum as the number of accesses made to the currently evicted page by the profiled GPU kernel.

While the code related to the performance monitoring counters accounts for most of the code in our interception library, this code is entirely generic: When integrating our kernel repetition loop directly into our benchmark applications, we were able to use the same code without modification. In addition, this code does not rely on CUDA and could thus be used by an interception library targeting a different API – such as OpenGL – without modification as well.

On current Nvidia GPUs, only the first domain in each memory-related domain set can count simple read and write operations. As a consequence, it is not possible to count memory read and write operations simultaneously since counting any type of memory access requires one of the counters' special operating modes, thus blocking the entire domain. To overcome this issue, Nvidia's profiling tools repeat each kernel multiple times, once for each desired event. Our kernel repetition loop replicates this behavior: The loop repeats each kernel twice for each page in the application's address space – once for read and once for write operations.

As a final complication, Nvidia GPUs require the profiled application itself to explicitly enable the performance monitoring counters by setting a flag in the

kernel launch commands written to the application's command submission channels. If that flag is missing for a given kernel, no events generated by the kernel's execution will be counted. Since performance monitoring counters can be a potential vector for side channel attacks [7], we assume this flag to be a security measure to prevent the activation of the performance monitoring counters against the application's will. Without breaking transparency, this security measure can be overcome in three ways: First, our profiler could use a modified version of the GPU's user-mode runtime library which sets the profiling flag unconditionally in all kernel launch commands. From the interception library's perspective, this approach effectively removes the counters' security altogether, which allows us to implement the interception library using only calls to GPUswap and the GPU's pre-existing user space runtime. Second, our interception library could submit all GPU kernels directly into the application's command submission channels, setting the profiling flag in the process. However, this approach would require the interception library to re-implement large parts of the GPU's user space runtime. Third, our profiler could virtualize the application's command submission channels in the GPU driver: The profiler could force the application to submit all commands into a shadow channel similar to those used by GPUswap, and subsequently set the profiling flag before copying the commands to the real channel. Since the latter two options are complex to implement, our current prototype uses a modified version of the GPU's runtime library for simplicity.

4.4 Profiling Results

Using the prototype of our profiler, we examined the memory access patterns of several GPU applications. In our profiling experiments, we pursued three main goals: i) To confirm the results of previous work [4], ii) to assess the time needed to collect a complete access profile using our profiler, and iii) to gain additional insight into the behavior of GPU applications which might be helpful in the development of an eviction policy for GPUswap.

4.4.1 Profiling Setup

In our profiling experiments, we used eight applications from the Rodinia benchmark suite [8]. We limit ourselves to those eight applications since these applications have been previously ported to the CUDA runtime API [37], which is the only API supported by GPUswap, and hence by our profiler as well. We used two versions of each application in our experiments: An unmodified version for use with our interception library, and a modified version with the kernel repetition loop required by or profiler added to the application's source code. As we expected, the modifications required for the latter version were relatively small: The kernel repetition loop amounts to about 25 lines of code.

Since one of our goals was to gain an understanding of the behavior of GPU applications, we determined an access count for each individual page in each application's address space, which is the finest granularity supported by our profiler. In addition, we configured the profiler to account for read and write accesses separately.

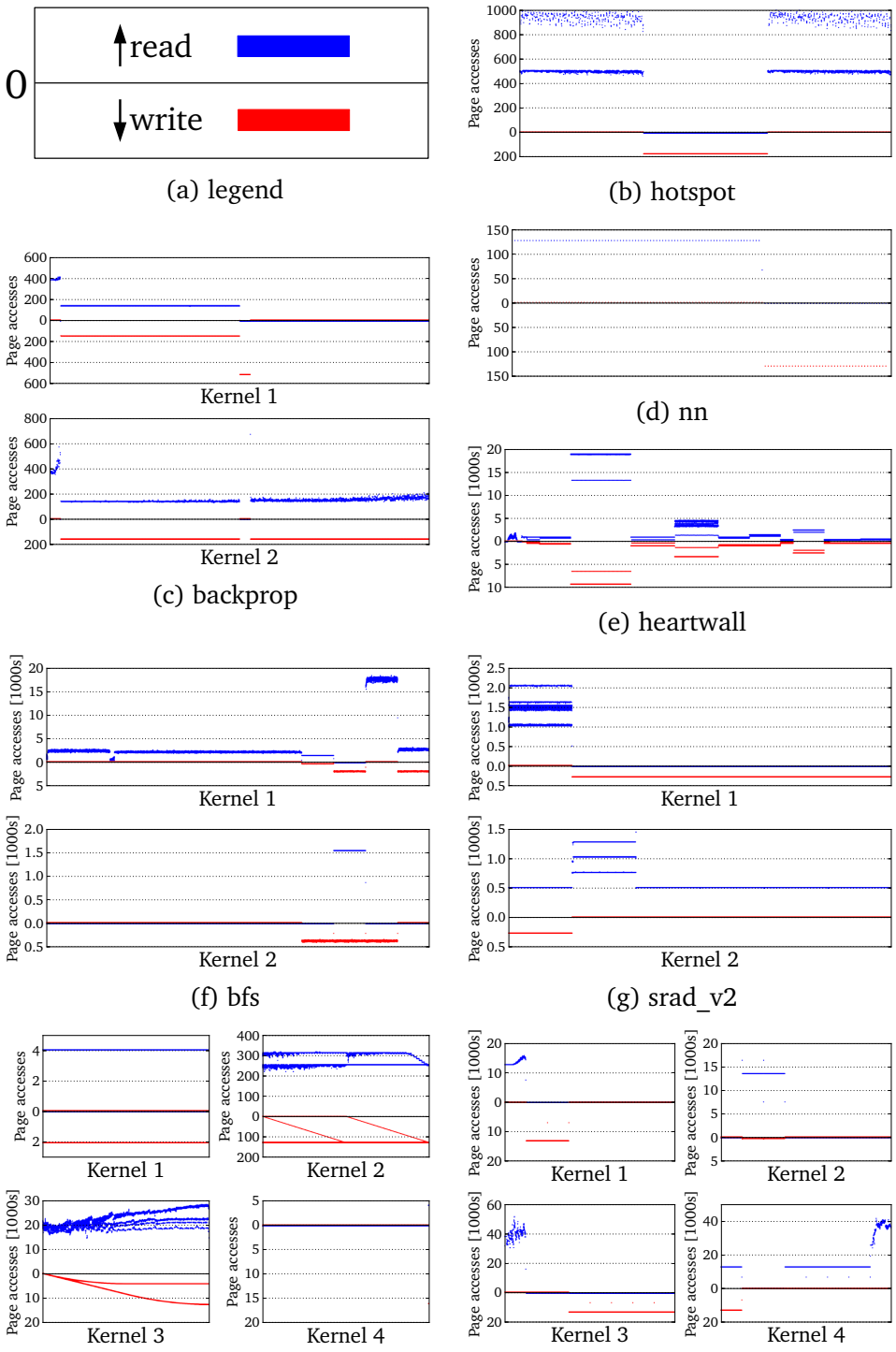
The test system used in our profiling experiments consists of a Core i7-4770 CPU clocked at 3.4 GHz and 16 GiB of system RAM. The GPU used in our experiments was a GeForce GTX 480. In all our experiments, both CPU and GPU were locked to the highest available clock frequency. Our test system ran Ubuntu 12.04 based on Linux version 3.5.7. We used pscnv as our GPU driver since that driver is a prerequisite for GPUswap. In user space, we used Gdev's CUDA runtime, modified to set the profiling flag on all kernel launches as discussed in Section 4.3.3.

4.4.2 Observations

The results of our profiling experiments are shown in Figure 4.4. The figure shows a separate access profile for each GPU kernel launched by each application. In all plots, the X axis represents the application's virtual address space, omitting unallocated regions for readability. Read and write accesses are shown on the Y axis, separated by a horizontal line at zero accesses. Read accesses are shown above that line and in blue, while write accesses are shown below the line and in red. Note that each page is represented by a single dot; any structures visible in the plots are composed of these dots without additional processing. We collected profiles for each application using our interception library and by adding the kernel repetition loop directly into the source code of our benchmark applications. For all applications, the results were identical for both methods; the profiles shown in Figure 4.4 are those collected from the modified applications.

Interestingly, the profiles of most applications contain clearly visible horizontal lines, indicating contiguous regions of virtual address space in which all pages share a similar number of accesses. An analysis of the applications' source code as well as our instrumentation of the applications' memory accesses show that each of these regions corresponds to a single application buffer.

For most of the applications, the difference in the number of accesses per page is large between these buffers compared to the difference between pages within the same buffer. However, there are also applications for which multiple buffers share the same number of accesses, thus appearing as a single horizontal line: For srad_v1 – shown in Figure 4.4(i) – for example, the top line in kernel 2, the bottom-right line in kernel 3 and the middle line of kernel 4 are all composed of multiple buffers. The boundaries of those buffers are indicated by the isolated dots between the solid horizontal line and the zero line: Since the size of each buffer is not an exact multiple of the GPU's page size, the last page in each buffer



(a) legend (b) hotspot (c) backprop (d) nn (e) heartwall (f) bfs (g) srad_v2 (h) lud (i) srad_v1

Figure 4.4: Profiling results for various applications from the Rodinia Benchmark Suite. The X axis in each plot represents the application's virtual address space with unallocated regions omitted. The Y axis shows the access count for each page. Read requests are shown in blue and above the zero line, while write requests are shown in red and below.

is not entirely filled with application data and is therefore accessed less often than the rest of the buffer. The only application for which no individual buffers are visible at all is lud – shown in Figure 4.4(h) – which only allocates a single buffer on the GPU.

While in general, the number of accesses per page appears to vary mostly between buffers, some applications do exhibit non-trivial differences in the number of accesses between different pages within the same buffer. For example, the trace of srad_v2, depicted in Figure 4.4(g), shows multiple parallel lines in the same region of the application's address space. This result indicates that the application buffer stored in that region is not read sequentially, but accessed in a more intricate pattern. Even for these applications, however, the difference in the number of memory accesses between buffers is typically larger than the difference within each buffer: For example, kernel 1 of srad_v2 accesses each page in the leftmost buffer more frequently than any page outside that buffer, even though the number of accesses per page is not uniform within the buffer.

Besides the buffers shown in Figure 4.4, our traces identified another large buffer in the address space of each application, which we omitted from the Figure for legibility since this buffer would have taken up most of the space in all plots. This buffer is allocated by the GPU's runtime library and serves as stack space for the application's GPU threads. While this buffer was present in the address space of all applications, the number of accesses to this buffer was generally low: Most pages in the buffer were never accessed at all, while those that were rarely reached 100 accesses. This result is consistent with our understanding of GPU hardware: The GPU typically attempts to keep each kernel's local variables in registers, only spilling variables to the stack if there is insufficient register space. Nonetheless, a stack must be present for each thread executing on the GPU: Since the GPU may spill data to the stack at any time, a missing stack could cause random GPU page faults, thus leading to the unexpected termination of GPU kernels. In addition the stack buffer, the GPU's runtime library allocates several other buffers, for example to store the code of the application's GPU kernels. These buffers are included in Figure 4.4, but are not clearly visible since each of these buffers typically consists of only a few pages. Note that all buffers allocated by the runtime library can be evicted to system RAM just like any application buffer.

To verify our results, we repeated our profiling experiments with different input data for some of the applications. The results of these experiments were generally consistent with those shown in Figure 4.4: While the absolute number of memory accesses changed with the input size, the relative number of accesses between different buffers did not. In particular, the difference in the number of accesses between buffers always tended to be larger than the difference between pages in the same buffer, and the same buffer always received the largest number of accesses, regardless of the input supplied.

Comparison to Related Work

Agarwal et al. [4] conducted an analysis similar to our own on several GPU applications. Four of the applications used in the authors' analysis – backprop, bfs, nn and srad_v1 – are present in our own analysis as well. For three of these applications, however, Agarwal et al. report only aggregate results in the form of a cumulative distribution function (CDF) of memory accesses over the application's address space: The authors sorted the pages in the application's address space by the number of accesses to each page, and then plotted a CDF of the number of memory accesses over the sorted pages. Applications with the same number of accesses to each page in their address space thus yield a linear CDF, while applications with large differences between pages show a CDF curved towards the upper-left corner of the plot.

For three of the applications analyzed by Agarwal et al., the authors' results agree well with our own: For backprop, Agarwal et al. report a CDF indicating that the application's address space is split into two parts, with one half of the address space receiving about twice as many memory accesses than the other. Consequently, our profiler identified two large buffers occupying almost the entire address space of the application, with one of these buffers receiving twice as many accesses as the other. For srad_v1, the CDF reported by the authors is almost linear, indicating that all pages in the application's address space exhibit a similar number of accesses. However, the CDF shows two inflection points, indicating that there are two buffers, each occupying a little over 10 % of the application's address space, which receive more memory addresses than the rest of the address space. Our own profiler identified eight equal-sized buffers for this application, as can be seen in Figure 4.4(i): The buffers on the far left and the far right of the figure receive a large and intermediate number of accesses per page, respectively, while the remaining buffers exhibit similar and low numbers of accesses per page. Since each of the application's buffers occupies 12.5 % of the application's address space, these results agree with the inflection points observed by Agarwal et al.

For bfs, Agarwal et al. present more detailed results: The authors identified two large buffers and several smaller ones, with the smaller ones receiving a larger number of accesses per page. The same is visible in our own analysis (Figure 4.4(f)): The two large buffers to the left of the figure receive relatively few accesses per page, while the smaller ones to the right are accessed more frequently. Furthermore, Agarwal et al. ordered the application's buffers by the number of accesses per page in each buffer. Consequently, sorting the application's buffers by the number of accesses per page reported by our own profiler yielded the same ordering. Finally, it is important to note that Agarwal et al. used a much smaller input than our own in their analysis: The authors' plot of bfs' address space only shows a little under 128 KiB of data, whereas in our own profiling, the application allocated more than 45 MiB of memory. Nonetheless, our analysis shows results

very similar to Agarwal's, which indicates that profiling of GPU applications can yield results that are representative for different inputs.

For nn, however, the results of Agarwal et al. differ from our own. The CDF reported by the authors indicates that nn's address space is split into two parts: The first part occupies one third of the application's address space while the second part occupies the rest. The same distribution is visible in our own results (Figure 4.4(d)): The application's address space consists of an input buffer which takes up two thirds of the address space and is only read, and an output buffer which occupies one third of the address space and is only written. According to the authors' analysis, however, the smaller of the two buffers receives more memory accesses per page than the larger one, resulting in the application's CDF being slightly curved. In contrast, our profiler reported the same number of accesses per page for both buffers, which corresponds to the CDF being a straight line.

To understand this result, we examined the source code of nn's sole GPU kernel. Each thread launched by this kernel reads two values from the application's input buffer and combines them into a single result, which is then written into the output buffer. All three of these memory accesses are performed directly on GPU memory – nn does not use the GPU's shared memory at all. Regardless of the input given to the application, the input buffer should thus receive twice as many memory accesses as the output buffer, which appears to contradict the results of Agarwal et al.

The results of our own profiler are nonetheless plausible since the GPU is often able to coalesce memory accesses. In microbenchmarks, we were able to identify two different types of coalescing: First, the GPU is able to coalesce multiple accesses from different threads in the same warp into a single memory access. Second, the GPU's cache tends to coalesce read accesses into a single memory access even if these accesses originate from different warps: If two values that are adjacent in memory are read in close succession – which is the case for nn – the first read will bring both values into the cache in a single memory access. The result of the second read is then served from the cache and therefore not counted as another memory access. However, the GPU's caches appear to be strictly write-through and therefore do not coalesce write accesses. Since Agarwal et al. used a simulated GPU with a modified memory subsystem in their experiments, their simulated GPU may have exhibited a different coalescing behavior than the physical one used in our experiments, which would explain the difference between their results and our own.

In addition to the four applications discussed above, Agarwal et al. examined several other applications, and frequently observed large differences in the number of accesses between application buffers. For some applications, however, they also found large differences within individual buffers, similar to those our profiler reported for lud. Overall, Agarwal et al. came to the same two conclusions that we

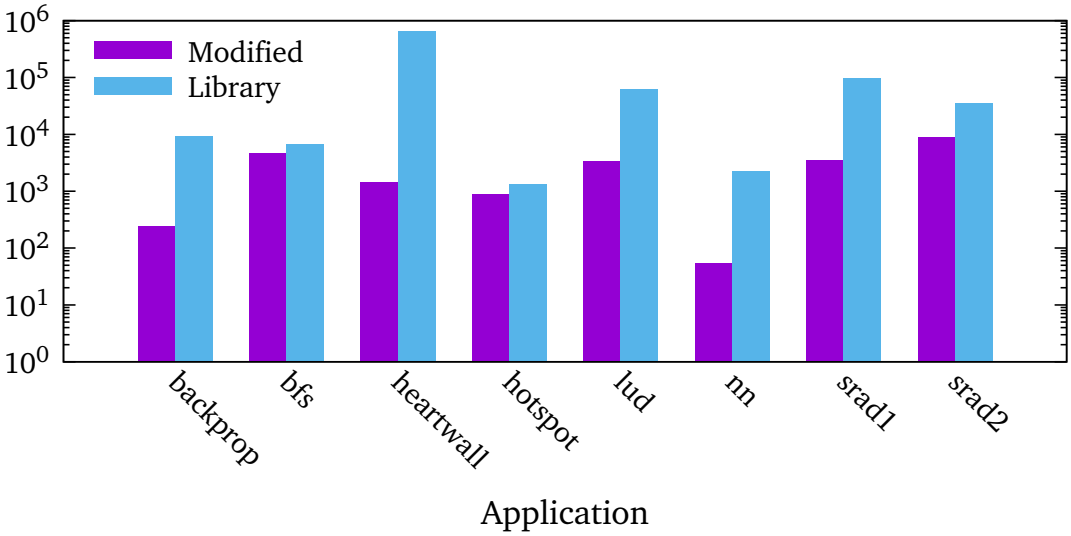
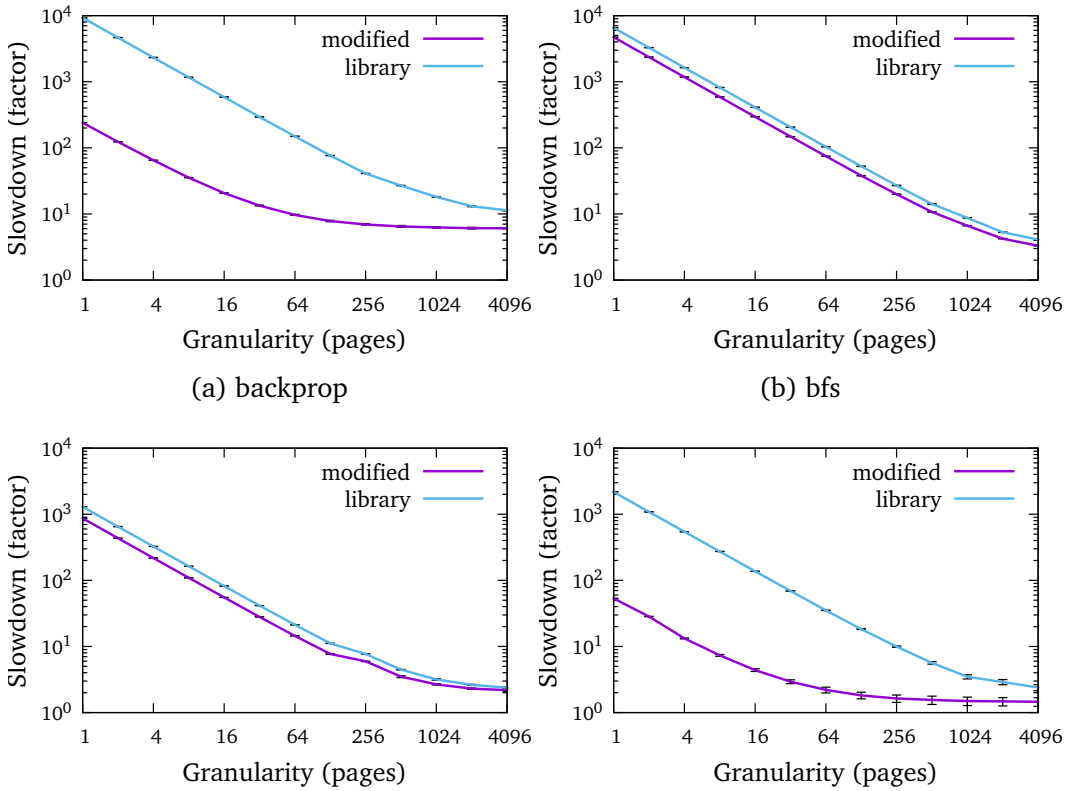


Figure 4.5: Slowdown due to profiling with page granularity for various applications. In this experiment, our profiler was configured to a granularity of one page, which is the finest possible granularity. "Modified" shows the slowdown for applications with the kernel repetition loop integrated directly into the application's source code, while "Library" indicates the slowdown when using our interception library on the unmodified application. We omitted error bars from this figure since the standard deviations of our results are negligibly small for all applications.

drew from our analysis: First, the number of memory accesses in the address space of GPU applications differs mostly between buffers. Second, some applications show a different number of memory accesses within the same buffer, which implies that a placement policy based on pages may yield a performance improvement over one based on buffers. Nonetheless, the authors' own policy, which is based on the results of their analysis, operates on buffers rather than individual pages.

4.4.3 Profiling Duration

The main drawback of our method of profiling is that our profiler must repeat the execution of each GPU kernel launched by the profiled application many times, thus causing a considerable slowdown in the application's execution. To quantify this slowdown, we measured the time needed to profile each of our benchmark applications both using our interception library and with the kernel repetition loop integrated directly into the application's source code. For comparison, we also measured the execution time of each application without any profiling, which allows us to calculate the slowdown induced by profiling.



(a) backprop (b) bfs

(c) hotspot (d) nn

Figure 4.6: Profiling time with different granularities for four of our benchmark applications. “Modified” shows the slowdown for applications with the kernel repetition loop integrated directly into the application’s source code, while “library” indicates the slowdown when using our interception library on the unmodified application.

Figure 4.5 shows the slowdown of each application for both methods of profiling with the profiling granularity set to one page, which is the finest granularity supported by our profiler. The results show that profiling indeed increases the applications’ execution times significantly: Our interception library induced slowdowns of more than 1000x in most applications, taking several hours to collect a complete access profile of the entire address space. In the worst case (heartwall), collecting a complete profile took about 4.5 days, which corresponds to a slowdown of more than 600000x.

Compared to our interception library, integrating the kernel repetition loop directly into the application decreased the slowdown induced by our profiler by more than an order of magnitude for most applications. Most notably, the slowdown of heartwall – which had been the worst case for our interception library – dropped from a factor of over 600000 to a factor of 1400. Consequently, integrating the

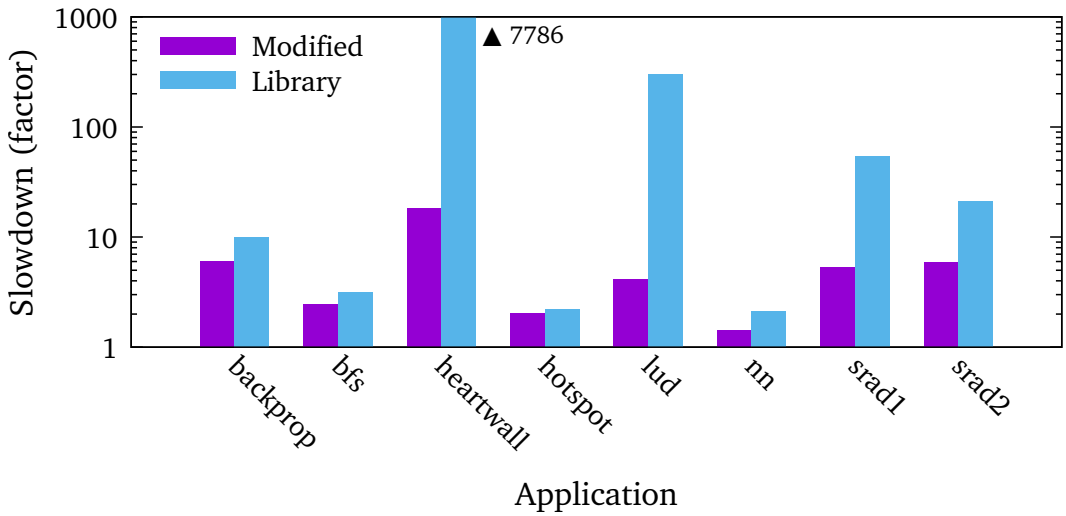


Figure 4.7: Slowdown due to profiling with buffer granularity for various applications. In this experiment, our profiler was configured to relocate an entire application buffer in each iteration of the kernel repetition loop. As in Figure 4.5, "Modified" shows the slowdown for applications with the kernel repetition loop integrated directly into the application's source code, while "Library" indicates the slowdown when using our interception library on the unmodified application.

kernel repetition loop into the applications brought the absolute time to collect a complete profile into an acceptable range as well: The slowest application was bfs with a profiling time of just under two hours. While the slowdown induced by our profiler is still severe even for our instrumented applications, the results thus clearly indicate that exploiting application knowledge can reduce the profiling time significantly.

Depending on the reason for profiling an application's memory accesses, a fine profiling granularity may not always be necessary. If a coarser profiling granularity is acceptable, our profiler allows the user to trade profiling granularity for faster profiling times. To quantify this tradeoff between granularity and profiling time, we measured the profiling slowdown for four of our benchmark applications with our profiler configured to different granularities. We chose to limit this experiment to the four fastest-running applications since the experiment would otherwise have taken a prohibitively long time to complete: We estimate that running this experiment on heartwall alone would have taken more than one month.

Figure 4.6 shows the profiling slowdown for the four fastest-running applications for various granularities. As can be seen, the profiling time initially decreases linearly with increasing profiling granularity. This result reflects our profiler's behavior: If the granularity is doubled, the profiler must repeat the application's

execution half as often, thus exactly cutting the profiling time in half. Towards the right of the figure, the profiling time approaches a constant value as the profiling granularity approaches the size of the application's buffers. Since our current prototype never puts multiple application buffers into a single chunk, a further increase in granularity then has no effect on the profiling time since the profiler is already relocating an entire buffer in each iteration.

Figure 4.7 shows the profiling slowdown when relocating an entire buffer in each iteration of the kernel repetition loop – which is the coarsest possible granularity supported by our current prototype – for all applications. At this granularity, the profiling time was reduced to an acceptable level for all applications: Even when using our interception library, our profiler collected a complete access profile in under one minute for most applications, while the slowest application (heartwall) took 1.25 hours to profile. Integrating the kernel repetition loop into the applications reduced the profiling time even further. However, since the profiling times using our interception library were generally acceptable, we consider the effort of instrumenting an application to be worthwhile only for extreme cases like heartwall at coarse granularity.

4.4.4 Implications on Policy

From our observations, we can draw several conclusions influencing the design of an eviction policy for GPUswap. The most important of these conclusions is that it may not always be necessary to identify the best individual page to evict: Since for the applications we examined, the number of accesses to a given page is often defined by the buffer that page belongs to, it may be sufficient to identify the right buffer to evict pages from. We assume this conclusion to hold for other GPU applications as well since these applications typically perform similar operations on large amounts of data in many threads. In addition, real-world applications tend to allocate more buffers for the policy to choose from than the relatively small applications in the Rodinia Suite – for example, we observed Google's project deepdream [19, 46], which builds on the caffe deep learning framework [35], to allocate 96.6 MiB of data in 746 buffers.[1] Therefore, we assume an eviction policy based on buffers rather than pages to have a sufficiently large number of choices to make useful eviction decisions.

It is, however, important to note that for many applications, some differences exist between pages within the same buffer. Even though these differences are typically small compared to the difference between buffers, they are often large enough to suggest that selecting the correct individual page can yield an additional benefit over selecting a random page from the right buffer. Unfortunately, detecting the

[1] Note that a larger number of buffers does prolong the profiling time at coarse granularity. However, we still expect profiling to be possible in reasonable time for such applications.

right pages is difficult on current GPUs due to the lack of reference bits in their page tables. Nonetheless, finding a way to identify frequently-accessed pages within each buffer – e.g., those forming the topmost line in the leftmost buffer of srad_v2 – could yield additional benefit for GPU memory management.

Finally, our observations show that an eviction policy cannot afford to generally ignore data structures allocated by the GPU runtime library in its decisions. For example, pages in the stack buffer are likely good candidates for eviction to system RAM: Since the GPU always attempts to keep local variables in registers, we expect the number of memory accesses to the stack buffer to be low for the vast majority of GPU applications. However, while the stack buffer cannot be ignored, there is also no guarantee that the stack buffer is a good candidate for eviction for all applications. Therefore, any eviction policy cannot unconditionally evict pages from the stack buffer, but must instead treat this buffer the same way as any other buffer in the application's address space.

Chapter 5

Potential Eviction Policies for GPUs

GPUswap requires an eviction policy to decide which data to evict in response to memory shortages. Creating such an eviction policy for current GPUs is not straightforward: To minimize the performance impact of its eviction decisions, any policy requires information about which data in GPU memory is frequently used. On current GPUs, this information is difficult to obtain since the hardware of these GPUs lacks appropriate hardware support, such as reference bits. The well-known techniques for solving this problem on the CPU therefore do not apply to GPUs. As a consequence, we must design a novel eviction policy that depends only on features available in current GPUs.

5.1 Goals

GPUswap is intended for use in a shared environment where multiple, mutually untrusted clients use the same GPU. Since these clients are typically unwilling to sacrifice the performance of their own applications for the benefit of others, our eviction policy should treat all clients equally. At the same time, the policy should minimize its own impact on the overall performance of the system. Specifically, we formulate the following goals for our policy:

Fairness In general, our policy should treat all running applications fairly in the sense that every application should receive an equal share of the GPU's resources. However, which resources should be considered by our policy is a matter of choice. In the context of our policy, we define fairness in terms of the amount of memory consumed by each application: Each application should receive an equal share of the GPU's memory. Conversely, applications that do not consume more than their fair share of memory to begin with should not suffer any overhead.

Performance While distributing resources fairly among applications is important, doing so may lead to a sub-optimal total overhead across the whole system. While our policy should always attempt to minimize the overhead caused by its eviction decisions, we consider fairness a more important goal than performance, especially since overhead is to be expected if data is evicted to system RAM.

Generality Our policy should not impose restrictions on the applications that can run on the GPU. Specifically, the policy should support running legacy applications which are not aware that eviction is taking place and for which no profiling data is available. While these applications will likely suffer a larger performance penalty than cooperating ones since the policy may lack necessary information without cooperation, we consider a higher penalty for non-cooperating applications acceptable as long as only the non-cooperating application itself is affected.

5.2 Policy Ideas

Whenever it is necessary to evict data from the GPU to system RAM, an eviction policy must make two decisions: First, the policy must select an application – which we call the **victim** – that must give up some of its GPU memory. Second, the policy must select one or more chunks of GPU memory from that application's address space which are subsequently moved to system RAM by the relocation mechanism. In this section, we examine several possible policies for both victim and chunk selection as well as their viability on current GPUs.

All policies described in this section have in common that, in addition to the memory already allocated in each application's address space, they must include the allocation request that triggered the eviction operation in their decisions. If the allocation request was ignored, the policy would be unable to enforce fairness in all situations: For example, any application could easily claim the entire GPU memory for itself simply by allocating a very large buffer. To mitigate this issue, GPUswap passes information about all chunks that will be created from the requested buffer to the eviction policy. If the policy selects chunks from the requested buffer for eviction, GPUswap subsequently allocates these chunks directly in system RAM.

5.2.1 Victim Selection

The first step in making an eviction decision is to find a victim application. This step is mostly responsible for ensuring fairness between applications: The victim always suffers a performance penalty when some of its data is relocated to system RAM since system RAM is much slower than GPU memory. Therefore, careful

selection of victims is required to ensure that the overhead of memory relocation is distributed fairly between applications. In comparison, the impact of the subsequent chunk selection step on fairness is limited since that step is restricted to the victim's address space and therefore does not influence other applications.

Performance-Based Selection

Different applications often show a different degree of sensitivity to memory bandwidth and latency: Applications for which memory is the main bottleneck may suffer large performance penalties even when small amounts of their data are evicted into slower memory, whereas applications limited by other factors may show near-optimal performance even if all of their data resides in the slower memory. To maximize fairness, our first potential policy – which we call the **performance-based policy** – thus focuses on the actual overhead induced by its eviction decisions: The policy's goal is to keep the slowdown experienced by each application proportional to the amount of memory consumed by that application. Since users typically care about the actual performance of their applications, we assume that this policy would be perceived as fair by most users. In addition, the policy would likely maximize the system's overall performance since applications insensitive to memory performance could give up large amounts of memory, thus allowing more sensitive applications to keep more of their data in GPU memory.

To ensure that each application's penalty is proportional to its memory consumption, the performance-based policy performs three steps to selects a victim: First, the policy measures each application's current slowdown compared to the application executing with all of its data in GPU memory. Second, once each application's slowdown is known, the policy calculates each application's relative performance penalty by dividing the application's measured slowdown by the amount of memory allocated in the application's address space. Finally, the policy selects the application with the smallest relative performance penalty as the victim.

One problem of this policy is that there is no correct behavior that reliably prevents an application's data from being evicted: Even if an application allocates only a small amount of memory, that application will initially have a relative performance penalty of zero since none of its data has been evicted. To make matters worse, even a single chunk represents a large portion of such an application's address space and is therefore likely to cause a large performance penalty if evicted. To mitigate this issue, the policy can ignore applications for which the total amount of allocated memory is below a certain threshold, which may depend on the number of running applications for increased fairness: For three applications running concurrently, for example, the policy could ignore any application which has allocated less than one third of the total amount of GPU memory available.

The main problem of the performance-based approach is that application performance – and hence the performance penalty of applications – is difficult to

measure without knowing the nature of the application: If the policy does not know what the application is doing, it cannot determine the amount of progress made by the application per unit of time. A possible solution to this problem may be to estimate the application's performance using the GPU's performance monitoring counters [27, 57]. To that end, the developer first profiles the application running in isolation to obtain readings of various performance monitoring counters as a baseline. The policy can then derive an estimate of the application's current performance by collecting readings of the performance monitoring counters at runtime and comparing these readings to the profiled values.

The main problem with this type of performance estimation is that the performance monitoring counters of current GPUs can only collect events for one application at a time. As a consequence, the policy would have to cycle through all applications after each eviction decision to obtain an updated value for each application's current slowdown. This cycling in turn leads to three problems: First, cycling through all applications can take a large amount of time if the number of applications using the GPU is large. During this time, the policy cannot make sensible eviction decisions since accurate information about the current slowdown is not yet available for all applications. Second, the policy would have to control which application is running at any given time to be able to attribute all measurements correctly. However, this type of control requires scheduling of GPU kernels in software, which would defeat the entire purpose of GPUswap. Third, cycling through all applications would yield only a sample of each application's performance during a short timeframe, which may not be representative if the application's execution consists of multiple phases. For these reasons, we consider a performance-based policy to be impractical for the time being. However, such a policy may well become practical on future generations of GPUs if the performance monitoring counters of these GPUs are able to monitor multiple applications concurrently.

Access-Based Selection

While it is difficult to measure an application's performance without knowing the nature of that application, it may be possible to estimate that performance based on other factors. Our second potential policy, which we call the **access-based policy**, takes this approach by assuming that the application's performance is proportional to the number of GPU memory accesses performed by the application. Consequently, this policy attempts to guarantee the same number of GPU memory accesses per unit of time to each application. To that end, the policy periodically measures the current number of GPU memory accesses per unit of time for each application. Measuring the number of memory accesses periodically is necessary since by the time a victim must be selected, the application that performed the memory allocation which triggered the policy will be blocked until the allocation request completes. When a victim must eventually be selected, the policy returns the application performing the largest number of accesses per time as the victim.

Compared to the performance-based policy, the access-based policy is much more easier to implement on current hardware: While performance estimation using performance monitoring counters requires complex models and off-line profiling of applications, the number of memory accesses per time is a clear metric which can be measured accurately using the GPU's performance monitoring counters. On the downside, however, this metric may not be an accurate estimate of the application's performance since not all applications are equally sensitive to memory performance: Some applications are able to overlap even high memory access latencies with other work and thus do not suffer noticeable slowdowns if memory performance decreases, while others must stall on memory accesses and thus experience severe overhead if memory performance degrades [26].

On current GPUs, our access-based policy suffers from the same problems as the performance-based policy: Since the performance monitoring counters of current GPUs can only monitor one application at a time, the policy would have to periodically cycle through all applications to ensure that a recent measurement is available for each one. This cycling can take a long time if the number of applications is large, requires software scheduling of GPU kernels and yields only a sample of the number of memory accesses per time for each application which may not be representative if the application exhibits phase behavior. Like the performance-based policy, we therefore consider the access-based policy to be impractical on current GPUs.

Memory-Based Selection

Since both the overhead caused by evicting data to system RAM and the number of memory accesses are difficult to distribute fairly on current GPUs, our third potential policy – called the **memory-based policy** – instead focuses on the amount of memory consumed by the applications. Specifically, the memory-based policy attempts to guarantee the same amount of memory to each application. Whenever a victim must be selected, the policy performs a single step: The application owning the largest amount of GPU memory is selected as the victim. Compared to the two policies described above, the memory-based policy is much simpler to implement: No profiling of, cooperation from or cycling through applications is required since the only input required by the policy – the amount of memory allocated by each application – is readily available for all applications through GPUswap's accounting mechanism.

Despite its simplicity, the memory-based policy has several desirable properties. First, it automatically converges towards a fair state: If the application owning most memory is always the one to give up memory, each application will own the same amount of memory eventually. In addition, the applications owning most GPU memory – which likely caused the memory shortage at hand – are automatically punished first. Finally, it is easy for applications to behave well

under this policy: Each application is guaranteed $1/n$ of the GPU's total memory for n running applications. If an application consumes less than this fair share of memory, that application will never see any of its data evicted. Conversely, if an application leaves part of its fair share unused, that unused part of the application's share will be equally divided among the other applications.

The main disadvantage of the memory-based policy is that application performance may not be determined by the amount of GPU memory available: On the CPU, it has been shown that some applications are highly sensitive to memory performance, while others are able to tolerate decreased memory performance without significant performance overhead [26]. If the same is true for GPU applications as well, some applications may experience larger overhead than others if the same amount of data is evicted to system RAM. However, the memory-based policy does not take varying sensitivity to memory performance into account at all, but instead treats all applications equally. Nonetheless, we assume that this policy is the only of the three policies described in this section that is practical to implement on current GPUs.

5.2.2 Chunk Selection

Once a victim has been selected, the eviction policy must select a chunk of memory from that victim's address space for eviction to system RAM. The policy's main goal in this step is to minimize the performance impact of its decision on the victim. Depending on the policy, however, the decision made in this step may also have an impact on fairness: With performance-based victim selection, for example, the amount of overhead induced by the decision made in this step may influence future victim selections.

Random Selection

The simplest chunk selection policy is to select a chunk from the victim's address space at random. The advantage of this **random selection policy** is that it can be applied to any application without prerequisites: Since random selection requires no input beside the application's chunk list, no information about or cooperation from the application are necessary. In addition, this policy does not depend on features that are not supported by current GPUs, such as reference bits. On the downside, however, this policy is likely to produce suboptimal results: The policy does not attempt to select chunks that have little impact on the victim's performance. As a result, one can expect not only high overhead in the victim application but also large variance in the overhead between different runs of the application, depending on which chunks are selected for eviction in each run. Nevertheless, random selection can serve as a fallback in case other policies cannot be applied, for example due to lack of profiling data or hardware support.

Sampling-Based Selection

While current GPUs lack reference bits, it may still be possible to collect information about frequently-accessed memory regions at runtime using the GPU's performance monitoring counters. Since these counters are limited to collecting information about one application at a time, the idea of our **sampling-based selection policy** is to use these counters to periodically collect samples of each application's memory accesses. To that end, the policy evicts one page from an application buffer to system RAM, and subsequently runs one of the application kernels to determine the number of accesses to that page. The policy repeats these steps for each buffer and each application. Assuming that all pages in a buffer share the same number of accesses, these measurements can then be used to derive a priority for each buffer at runtime.

The main advantage of sampling-based selection is that this type of policy does not require any prior knowledge about applications: All information required for the policy's operation can be collected at application runtime. On the downside, however, this type of policy has two main disadvantages. First, evicting pages to system RAM is likely to cause overhead in the application being sampled: Even if only one page is evicted to system RAM, GPU threads accessing that page could be slowed down considerably, possibly prompting other threads to wait since current GPUs only offer barrier synchronization. While we do not expect this overhead to be severe for most applications if only a single page is evicted at a time, the policy induces this overhead even if no memory pressure is present, thus violating the main goal of GPUswap described in Section 3.1.

An even more severe problem with this type of policy is that sampling always introduces inaccuracy. If the number of accesses per page is not uniform within a buffer, for example, the evicted page may not be a representative sample. While the accuracy of sampling can be increased by evicting more pages from the buffer – or even the entire buffer – evicting a larger number of pages also leads to a larger application overhead. In addition, a single GPU kernel is typically not representative of the application's execution. The policy would thus have to sample each buffer once for each kernel before meaningful eviction decisions can be made. We therefore expect sampling to take too long to be practical: Most applications we examined do not repeat each of their kernels once for each of their buffers, which implies that the policy would have no chance to collect the necessary information before the application finishes execution.

Reverse Swapping

In contrast to current GPUs, the IOMMUs of current CPUs often include support for both reference and dirty bits. Specifically, the IOMMU in recent Intel CPUs can set these bits in the application's page table during first-level translation [29],

while AMD's IOMMUs support these bits during both first- and second-level translation [3]. While this feature cannot provide information about pages in GPU memory, it could enable an eviction policy to detect frequently-accessed pages after these pages have been evicted, thus allowing the policy to detect and reverse bad decisions. Our **reverse swapping policy** leverages this feature as follows: When chunks must be selected for eviction, the policy returns a random set of chunks from the victim's address space. Then, the policy examines the reference bits in the IOMMU's page tables in regular intervals. In each interval, the policy records which of the evicted pages have their reference bits set, and subsequently resets the reference bits of all evicted pages. Finally, whenever one of the victim's GPU kernels finishes execution, the policy prompts GPUswap to move chunks containing frequently-accessed pages back to GPU memory, evicting an equal number of random chunks in the process. Eventually, if this process is repeated often enough, only rarely-accessed pages will remain in system RAM, while all frequently-accessed pages will reside in GPU memory.

Like the sampling-based selection policy, reverse swapping does not require prior knowledge about the applications using the GPU, and the applications need not cooperate in any way – in fact, it is not even necessary to gather any information at all about applications before the reverse swapping policy can make eviction decisions. In contrast to sampling-based selection, applications are therefore not delayed when their allocation requests trigger an eviction.

The main problem with this type of policy is that since the policy initially has no information about which chunks contain frequently-accessed pages, the policy will have to perform multiple iterations before all frequently-accessed pages reside in GPU memory. This matter is further complicated by the fact that different GPU kernels access different pages. Pages that are not accessed at all by one kernel may thus be evicted to system RAM even though the next kernel to run will access these pages frequently. While the policy can incorporate information about all GPU kernels seen so far in its decisions, we assume that the policy must observe each kernel multiple times before sufficient information is available to decide which pages should be in GPU memory. Therefore, we expect this type of policy to take a rather long time to converge, during which frequently-accessed pages in system RAM may cause significant application overhead. In addition, this type of policy must frequently move data between GPU memory and system RAM between iterations, which causes additional overhead.

Priority-Based Chunk Selection

Ideally, our eviction policy should select chunks which are rarely accessed by the application since these chunks likely have the smallest impact on the application's performance. While current GPUs do not allow the policy to collect information about the number of accesses to a given chunk due to lack of hardware support,

it is possible to obtain this information ahead of time through profiling, for example using a profiling mechanism like the one we described in Chapter 4. The information can then be used to assign a **priority** to each chunk. The application can subsequently pass these priorities to GPUswap as an additional parameter to its memory allocation requests. When a chunk must be selected for eviction, the **priority-based chunk selection policy** simply returns the chunk with the lowest priority from the application's address space. Similar techniques have been used in the past, for example in self-paging systems like Nemesis [23].

The main advantage of this type of policy is that it can capture the application's memory access patterns even within buffers. Therefore, we expect this policy to achieve much lower eviction overheads than random selection. In addition, application performance should also be more consistent across executions than with random selection: Since each chunk's priority is static, the same chunks are chosen for eviction each time the application is executed. However, there are also a number of downsides to this policy. First, the application must be profiled before this policy can operate, which can be a time-consuming process. Second, if the application is profiled using smaller input data than is later used in production, some chunks seen in production may not exist in the profiling data. For some applications with regular access patterns, it may be possible to interpolate the number of accesses to the missing chunks; however, such interpolation is always less accurate than direct profiling.

The most severe problem with this type of policy, however, is that the number of memory accesses per chunk may not be a good metric for the chunk's impact to the application's performance: On the CPU, it has been shown that applications differ in their sensitivity to memory latency [26]: Some applications have been shown to experience severe overhead if the latency of memory accesses increases, while others are apparently able to hide this latency and thus experience next to no overhead. Since GPU kernels are essentially small, independent programs processing a self-contained task, these kernels may exhibit similar effects: The same number of accesses to a chunk could cause a different amounts of overhead in different kernels. On the upside, the policy itself only operates on priorities, but does not care how these priorities are generated. Therefore, if another metric turns out to be better suited to assess the performance impact of evicting specific chunks, this metric can be used instead of the number of memory accesses without modifications to the policy.

Priority-Based Buffer Selection

A straightforward way to reduce the complexity of priority-based chunk selection is to assign priorities to buffers instead of chunks. The resulting **priority-based buffer selection policy** operates in a similar fashion to priority-based chunk selection: First, the application is profiled to determine the number of memory

accesses to each application buffer. Based on the results of the profiling, a priority is then assigned to each application buffer. The application then passes these priorities to GPUswap as part of its memory allocation requests. Finally, whenever a chunk must be selected for eviction, the policy searches the victim's address space for the buffer with the lowest priority and selects a random chunk from that buffer for eviction.

Although priority-based buffer selection is based on the same basic idea as priority-based chunk selection, it solves two of the main problems of the chunk-based policy: First, we expect a buffer-based policy to be less sensitive to the input data used in profiling than a chunk-based policy: Changes in the number of chunks between profiling and production can simply be ignored since we found the relative importance of buffers not to change with varying input data. Second, the profiling data necessary for the operation of a buffer-based policy can be obtained much faster since only the total number of memory accesses per buffer is of interest. If the profiling method described in Chapter 4 is used, it is therefore sufficient to repeat the application's execution once per buffer rather than once per chunk. The third problem, however, affects both policies alike: The number of memory accesses to a buffer may not be a good metric for the buffer's impact on the application's performance. As with priority-based chunk selection, however, priority-based buffer selection does not depend on a particular method for generating the priorities. The number of memory accesses can thus be replaced by a different metric without modifications to the policy.

An additional problem with priority-based buffer selection is that even though chunks in the same buffer often share a similar number of accesses, that may not be the case for all applications. In our own profiling experiments, we observed some applications with major differences in the number of accesses between pages in the same application buffer. Since these differences cannot be captured by a single priority for an entire buffer, we expect priority-based buffer selection to produce larger overhead than priority-based chunk selection for this type of application. However, our profiling experiments also showed that these applications appear to be the minority: For most applications we examined, pages in the same buffer share a similar number of accesses. We therefore expect priority-based buffer selection to yield a performance similar to that of priority-based chunk selection for most applications.

5.3 Prototype Policy

To demonstrate the benefits of an eviction policy, we integrated a viable prototype policy into our existing prototype of GPUswap. Since all policies described above suffer from serious drawbacks, our main goal in implementing this policy was not to create a policy that can be used in production environments, but to assess to

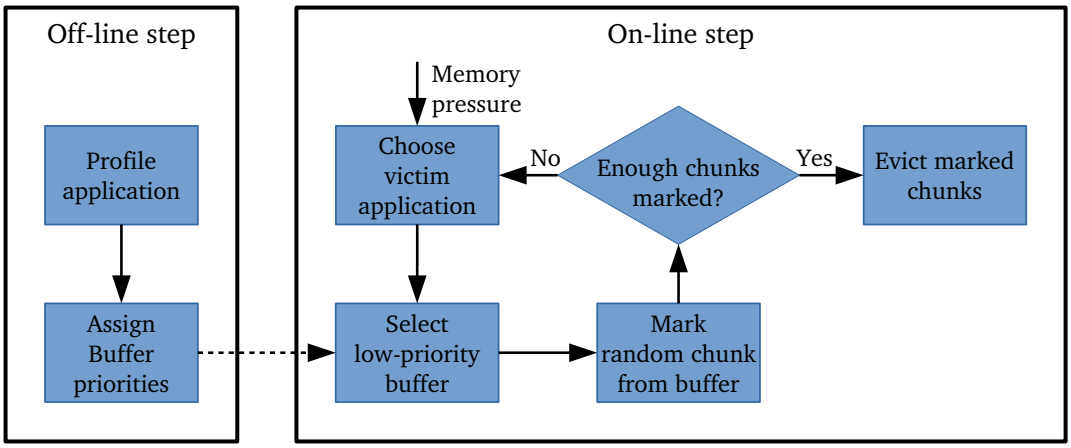


Figure 5.1: Operation of our prototype policy. First, the developer profiles the application and assigns priorities to the application's buffers based on the results of that profiling. Then, when there is insufficient GPU memory to perform an allocation request, the policy performs three steps: First, it selects the application owning most GPU memory as the victim. Second, it selects the lowest-priority buffer from the victim's address space. Finally, it selects a random chunk from that buffer and marks that chunk for eviction. This process continues until the marked chunks free up enough memory to accommodate the allocation request that triggered the policy. At that point, all marked chunks are evicted to system RAM.

what extent an eviction policy can mitigate the slowdown caused by evicting GPU data to system RAM. Consequently, we chose a policy that is reasonably simple to implement, yet likely offers a significant benefit compared to random selection: Our policy is based on memory-based victim selection and priority-based buffer selection.

5.3.1 Overview

Our prototype policy operates in two steps. In the first, off-line step, the developer must profile the application's memory accesses, for example using the profiling mechanism described in Chapter 4, to determine which GPU memory buffers the application accesses frequently. Based on the results of that profiling, the developer then assigns a priority to each of the application's buffers. Finally, the developer modifies the application to pass these priorities to GPUswap as part of its memory allocation requests.

Once a priority has been assigned to each buffer, the second, on-line step of our policy can use these priorities to select memory for eviction: Whenever the GPU driver receives a request for GPU memory, but cannot satisfy this request due to

insufficient resources, our policy first uses memory-based selection as described in Section 5.2.1 to find a victim application: The application owning most GPU memory is selected to give up some of its GPU memory. Once the victim has been selected, our policy uses priority-based buffer selection, described in Section 5.2.2, to choose a chunk from the victim's address space for eviction: The policy finds the lowest-priority buffer from the victim's address space, selects a random chunk from that buffer and marks that chunk for eviction to system RAM. The policy repeats these two steps until all marked chunks combined free up enough memory to accommodate the outstanding allocation request. Once that is the case, the policy returns the list of marked chunks to the relocation mechanism, which subsequently moves the contents of these chunks to system RAM. The entire operation of our policy is shown in Figure 5.1.

While the main purpose of our policy is to assess the benefit of an eviction policy in general, our policy can also serve as a placeholder for production environments for the time being: The policy achieves fairness even in the presence of unmodified applications since memory-based victim selection is independent of priorities, and if application source code is available, our priority-based buffer selection scheme can be applied to any application with only small modifications. On the downside, however, the profiling required for our policy's operation is a time-consuming process. For the time being, there is no alternative to profiling since current GPUs lack features that enable advanced memory management, such as reference bits. Once these missing features have been added to GPU designs, however, we plan to replace our prototype with a policy that does not require modifications to application source code or off-line profiling.

5.3.2 Priority Assignment

To support our policy in selecting chunks for eviction in a meaningful way, the developer must assign priorities to application buffers. To assign these priorities, developers should profile the memory accesses of their applications. This profiling can be achieved, for example, using our profiling mechanism described in Chapter 4.

A principal difference between the profiles collected in Chapter 4 and the profiling required for our policy is the required accuracy: Since only a single priority value per buffer is needed, there is no need to determine the number of accesses to each individual page in the application's address space. Therefore, our profiling mechanism can evict entire application buffers at a time instead of individual pages. In addition, our prototype policy weighs read and write accesses equally, which implies that both can be profiled at the same time. Therefore, our profiling mechanism must only repeat the execution of each GPU kernel once, which speeds up profiling considerably as shown in Section 4.4.3.

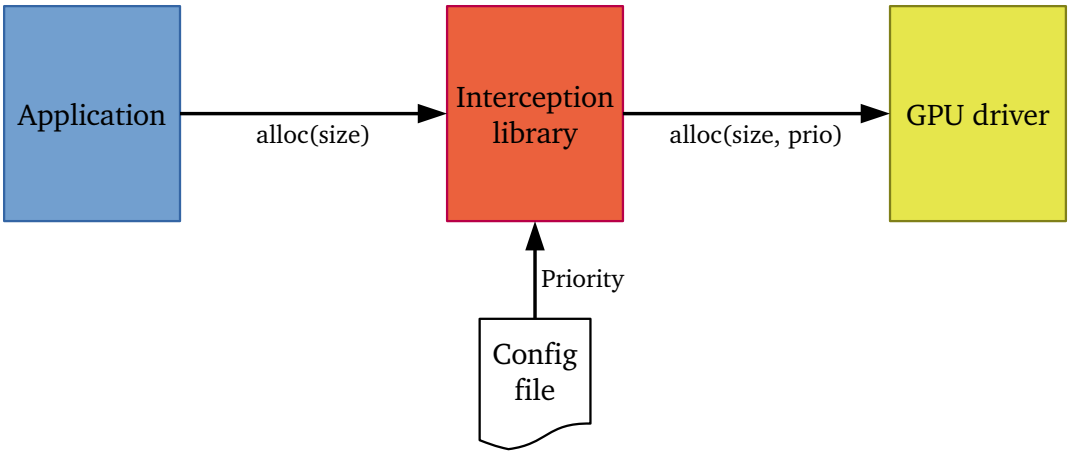


Figure 5.2: Passing priorities using an interception library. The library replaces the original allocation function in the GPU's user space runtime library with its own implementation. When the application attempts to allocate GPU memory, the interception library reads the intended priority for the newly allocated buffer from a configuration file and passes this priority to GPUswap using the appropriate allocation system call. The entire process is transparent to the application.

Once the total number of memory accesses is known for each buffer, the administrator can assign a priority to each buffer. Since priorities are simple integers of arbitrary length, each buffer can be assigned a unique priority, though it is also possible to assign the same priority to multiple buffers if desired. For our prototype, we assign priorities in two steps: First, we calculate each buffer's average number of accesses per page by dividing the number of accesses to the buffer by the number of pages in the buffer. Second, we order all buffers by the number of accesses per page. The buffer with the largest number of accesses per page then receives the highest possible priority, the buffer with the second largest number of accesses per page is assigned the second highest priority and so on. If multiple buffers share the exact same number of accesses per page, we currently assign different but adjacent priorities to those buffers. Note that there is no need to coordinate these priorities between applications since all priorities are local to the respective application.

5.3.3 Passing Priorities to GPUswap

Once a priority has been assigned to each buffer, the application must pass these priorities to GPUswap as part of its allocation requests. To allow the application to pass priorities, GPUswap offers an alternate system call for GPU memory allocation in addition to the GPU driver's original call. GPUswap's alternate memory allocation call implements the same functionality as the driver's original

call, but accepts the priority as an additional parameter. Upon invocation of the alternate call, GPUswap invokes the driver's original memory allocator and subsequently stores the passed priority in the allocated buffer's metadata. Buffers allocated using the original call are assigned a medium priority by default.

To allow applications to use the new system call, GPUswap extends the GPU's user space runtime library with a matching allocation function which accepts the priority as an additional parameter. Since this function is otherwise identical to the runtime's original allocation function, the modifications required in the applications' source code to use the new allocation function are generally small: An additional parameter must be appended to each memory allocation request. If modifying an application's source code is infeasible, a more transparent alternative is to employ an interception library which replaces the GPU runtime's allocation function with its own implementation as shown in Figure 5.2. When the interception library's allocation function is called, it reads the priority for the buffer being allocated from a per-application configuration file and passes the value from that file to GPUswap's allocation system call. In case no priority is found, the library falls back to the original allocation call, thus effectively assigning the default priority to the buffer.

A major challenge when using an interception library is to reliably identify the right entry in the configuration file: Since the size of each buffer, the allocation order and any memory addresses involved are not necessarily constant across executions, it is difficult to determine which buffer the application is trying to allocate with each call. This problem does not occur when modifying the application's source code: The developer knows exactly which buffer is being allocated with each call and can therefore easily pass the correct priority with each allocation request. If a more dynamic configuration of priorities is desired, the application could also pass a unique identifier for each buffer instead of a priority, which would allow an interception library to reliably identify the correct entry in a configuration file. While this hybrid approach is also not fully transparent, it allows the developer to change priorities without recompiling the application.

5.3.4 Selecting Chunks for Eviction

Whenever there is insufficient GPU memory available to satisfy an allocation request, our policy must select an **eviction set** of chunks which should be evicted to system RAM. The policy builds this set in two steps: First, our policy selects a victim application which must give up some of its memory. Then, the policy selects a chunk from the victim's address space and adds that chunk to the eviction set. The policy repeats these two steps until the eviction set is large enough to make room for the outstanding request. Note that both steps must include all chunks from the buffer currently being allocated in their decisions.

Our policy uses memory-based selection to determine the victim application: The application with the largest amount of GPU memory allocated in its address space is always selected as the victim. To distribute the selected chunks fairly across applications owning similar amounts of memory, our policy ignores chunks already in the eviction set when selecting the victim, thus effectively treating these chunks as already evicted. The main advantage of this design is that it automatically converges against each application owning the same amount of memory, thus ensuring fairness between applications. In addition, if an application allocates less than its fair share of memory, the leftover memory is distributed fairly among the other applications.

Once a victim has been selected, our policy uses the priorities determined in the off-line step to select a suitable chunk of memory for eviction. To that end, our policy builds a temporary set of chunks, which we call the **decision set**. This decision set consists of all chunks from the victim's address space sharing the lowest priority present in that address space. As during victim selection, our policy ignores all chunks already in the eviction set while building the decision set, ensuring that the policy continues to the next higher priority if all chunks of the lowest priority are already in the eviction set. Once the decision set has been assembled, our policy selects a random chunk from the decision set and adds that chunk to the eviction set. Note that chunks with the same priority may still have a different impact on the victim's performance when evicted; selecting chunks at random helps to smooth out these differences.

Whenever a chunk has been added to the eviction set, the policy must check whether a sufficient number of chunks has been selected to satisfy the outstanding allocation request: If the sum of the amount of free GPU memory and the combined size of all chunks in the eviction set is larger than the amount of GPU memory requested, the policy terminates and returns the eviction set to GPUswap's eviction mechanism. If not, the policy repeats the entire process, starting with victim selection. Repeating victim selection is necessary since evicting a chunk may change the victim of the next eviction if multiple applications own similar amounts of memory. Note that this design leaves room for optimization – for example, it would be possible to compute the amount of memory each application must give up during victim selection and then select an appropriate number of chunks from each application without repeating victim selection. However, the runtime of our policy is not a major issue since GPUswap's eviction operations are dominated by DMA transfers as shown in Section 6.3.2.

5.3.5 Returning Data to the GPU

Whenever an application frees GPU memory, our policy must select a **return set** of evicted chunks that should be returned to the GPU. As when selecting chunks for eviction, our policy builds the return set in two steps: First, our policy selects a

winner application which will have one of its chunks returned to the GPU. Then, the policy selects an evicted chunk from that winner's address space and adds that chunk to the return set. The policy repeats these two steps until either the return set fills all available GPU memory or all evicted chunks have been added to the return set.

Our policy again employs memory-based selection to determine winner applications: The application owning the least amount of GPU memory is always selected as the winner. Conversely to victim selection, our policy adds the size of each chunk in the return set to the amount of GPU memory allocated by the application owning that chunk during winner selection, effectively treating chunks in the return set as if these chunks were already in GPU memory. This scheme ensures that the available GPU memory is distributed fairly among all applications with evicted chunks.

Once a winner has been selected, the policy builds a decision set containing the evicted chunks with the highest priority in the winner's address space. As when selecting chunks for eviction, the policy ignores all chunks in the return set while building the decision set, ensuring that the policy continues with the next lower priority once all evicted chunks of the highest priority have been added to the return set. The policy then selects a chunk at random from the decision set and adds that chunk to the return set. When the policy eventually terminates, it returns the return set to GPUswap, which transfers the contents of all chunks in the return set back to GPU memory.

5.3.6 Policy Implementation

To assess the benefit of our policy, we added a prototype implementation of that policy to GPUswap. For chunk selection, this prototype implements both priority-based buffer selection and random selection, allowing us to evaluate the benefit of priorities by comparing the two policies. Victims are always selected based on the amount of GPU memory allocated in each application's address space.

Passing Priorities to GPUswap

To allow applications to pass priorities to GPUswap, our prototype extends the pscnv driver with an alternate memory allocation system call taking a priority for the allocated buffer as an additional parameter. GPUswap also provides pscnv's original memory allocation call to support legacy applications; however, that call is merely a wrapper around the alternate call using a default priority. To facilitate applications' use of the new allocation system call, we also modified the Gdev CUDA runtime to provide an alternate version of CUDA's memory allocation function which is backed by the alternate allocation system call. This function

takes the priority as an additional parameter, but is otherwise identical to CUDA's original allocation function.

When an application calls either of the two allocation system calls, GPUswap first performs any evictions necessary to make room for the allocation request. Then, GPUswap uses pscnv's original memory allocator to allocate the requested amount of memory minus the combined size of those chunks that were allocated in system RAM during the eviction. Finally, GPUswap's accounting mechanism stores the passed priority – which may be the default priority – in all chunk list entries created from the allocated buffer. Storing the priority with each chunk has the advantage that the policy can subsequently operate directly on chunks, without having to know which chunk originated from which buffer.

To allow for dynamic reconfiguration of priorities, we finally created a separate wrapper function which fulfills the role of the interception library described in Section 5.3.3. This wrapper function takes a unique identifier for each buffer instead of a priority as its additional parameter. The wrapper function then uses this identifier to read the buffer's priority from a configuration file and calls CUDA's priority-enabled allocation function with that priority, or the default priority if no appropriate entry is found in the configuration file. All of our benchmark applications currently use this wrapper function instead of CUDA's original allocation functions.

System buffers, such as the stack buffer, present a challenge for this scheme: Since these buffers are allocated by the GPU runtime library directly through the driver's allocation system calls, we cannot inject priorities for these buffers through an interception library or by modifying the application's source code. Instead, we modified Gdev's user space runtime library to use GPUswap's new allocation call to allocate all system buffers. The priorities for these buffers are currently hardcoded in the runtime library, which may be a problem if different priorities for these buffers are required for different applications. For our benchmark applications, however, we found a set of priorities which is suitable for all applications: The stack buffer is assigned the lowest possible priority, while all other system buffers receive the highest possible priority.

Chunk Selection

While our prototype supports only memory-based victim selection, it includes two policies for chunk selection: Priority-based buffer selection and random selection. Upon loading GPUswap's kernel module, the administrator can choose which of the two policies to use via a module parameter. The chosen policy then governs both eviction and returning data to the GPU for all applications. In the remainder of this section, we only describe selection of chunks for eviction; however, the same principles apply when selecting chunks to return to the GPU.

To avoid frequent recompiling of our benchmark applications, our prototype provides the priority-enabled memory allocation system call even if random selection is in use. The passed priority is still stored in GPUswap's chunk list in that case. However, if random selection is active, our policy skips generation of the decision set altogether and instead selects chunks randomly from the victim application's GPU chunk list. While storing priorities is thus unnecessary work when using random selection, the performance impact of storing a single integer per chunk is negligible.

Our policy must ignore all chunks already in the decision set when selecting a victim to avoid unfairness when building large decision sets. Since processing the entire decision set on each victim selection is time-consuming, our prototype instead updates GPUswap's metadata immediately after adding a chunk to the eviction set: After adding a chunk to the set, the policy removes that chunk from the application's GPU chunk list and adds the chunk to the application's system RAM chunk list. In addition, the policy immediately decrements the amount of GPU memory allocated by the application. While this approach ensures that selected chunks are ignored without requiring the eviction set to be examined frequently, it also causes GPUswap's metadata to briefly become inconsistent during the policy's operation: Selected chunk are effectively treated as already evicted, even though these chunks still reside in GPU memory until the policy terminates. Without mitigation, this inconsistency could cause another application to allocate memory without triggering an eviction if GPUswap's metadata shows a sufficient amount of free GPU memory even though some of that free memory is still occupied by chunks selected for eviction. To prevent this inconsistency from causing problems in practise, we delay all memory allocation requests arriving during the policy's operation until the subsequent evictions are complete.

5.4 Hardware Wishlist

Since the applications we examined showed a high degree of uniformity in their memory accesses within buffers, we expect a priority-based policy to achieve good results for most applications. For production environments, however, we consider our prototype policy to be less than optimal for three reasons: First, our policy requires time-consuming profiling before it can make effective decisions. Second, some applications did show a non-uniform access pattern in some of their buffers, which a single priority per buffer is unable to capture. Third, our prototype requires the applications' source code to be modified, which is infeasible for most commercial software.

For an eviction policy to work well in production, that policy must be able to make sensible decisions for any application out of the box, without requiring modifications to the application or off-line profiling. However, such a policy is not

viable on current GPUs since these GPUs lack several common features related to memory management. Specifically, the following features would help with building a viable eviction policy if added to future GPUs.

Reference Bits are present in the page tables of all contemporary CPUs. Most existing techniques for memory management build on these reference bits in one way or another. Consequently, similar techniques could be applied to GPUs if reference bits were available in the GPU's page table as well: The GPU driver could read these bits periodically to determine which pages in each application's address space are accessed frequently. This periodic process could be either be implemented on the GPU itself, leveraging the GPU's large number of cores and wide memory bus to read many page tables in parallel, or on the CPU, keeping more GPU time available to applications. In the event of memory pressure, an eviction policy could then use the information gained from those reference bits to choose pages for eviction using one of the well-known swapping algorithms [60]. Such an eviction policy would not require any prior knowledge about applications, and would likely surpass our prototype in terms of performance since it could account for non-uniform access patterns within application buffers. Since current GPUs already feature virtual memory and MMUs, we assume that it is possible to add reference bits to these GPUs.

Page Faults are already supported by the latest generation of Nvidia GPUs: If a GPU kernel accesses an address for which no entry exists in the application's page table, the GPU stops the faulting kernel and raises an interrupt. The GPU driver then handles the page fault and signals the GPU to resume the faulting kernel. At the time of this writing, however, details on how to implement page fault handling in the GPU driver are not publicly known since no appropriate documentation is available. Therefore, page faults are only supported by Nvidia's proprietary driver, but not by any of the open-source GPU drivers currently available. Nevertheless, page fault support could be useful to GPUswap if documentation was released in the future.

Currently, GPU page faults are mainly used in the context of Nvidia's unified memory to synchronize data between CPU and GPU in a manner similar to the demand paging used in current operating systems [49]. Since GPU page faults are handled by the GPU driver, the latency of GPU page faults is typically higher than that of CPU page faults [59]. Unless GPU data is accessed very frequently, the overhead of moving accessed pages to the GPU through page faults is thus often larger than that of accessing evicted data directly over the PCIe bus[1]. Therefore, we consider GPU page faults in their current form to be inviable for eviction purposes since the explicit goal of our eviction policy is to keep frequently-accessed pages in GPU memory. Besides demand paging, however, page faults can also be used

[1] Even CUDA applications require careful profiling and tuning to actually benefit from page fault support [55].

to emulate reference bits by intentionally unmapping pages and subsequently counting the number of page faults to each page [12]. While this approach leads to a high number of page faults, it could allow us to build a fully transparent eviction policy even if reference bits are not available by periodically sampling the number of accesses to each page.

Preemption is also supported by the latest generation of Nvidia GPUs: The GPU driver can stop running kernels and seamlessly resume their execution later on. This feature is a prerequisite for page faults since a faulting kernel must be stopped immediately and cannot continue its execution until the page fault is resolved. Like page faults, however, this feature is not publicly documented and therefore inviable to use at the time of this writing.

If appropriate documentation were available, we could use preemption to reduce the memory allocation latency in the presence of memory pressure: Currently, GPUswap must drain the command submission channels of each application before evicting chunks from the application's address space. With preemption, this draining would no longer be necessary: GPUswap could instead stop all running kernels in the application's GPU address space, evict all selected chunks, and subsequently resume all stopped kernels.

Per-Application Performance Monitoring Counters are not supported by any current GPU. Instead, the performance monitoring counters of these GPUs can only count events from one application at a time, which makes monitoring the interaction of multiple applications a cumbersome process since events must be counted separately for each application. If instead there was a set of performance monitoring counters for each application, it would be possible to measure the performance of multiple applications simultaneously in real time. This feature would enable us to build a performance-based policy as described in Section 5.2.1: Determining both each application's baseline performance and its slowdown after evicting chunks could be done in reasonable time if all applications could be measured concurrently. In addition, the policy could also capture interactions between applications after an eviction, such as one application running faster if another application is slowed down as a result of an eviction. Since such performance-based policies are not widely used, this feature opens up far more interesting research opportunities than the other features described in this section; however, we also expect this feature to be the most difficult to integrate, and therefore consider it unlikely to happen in the foreseeable future.

Chapter 6

Performance Evaluation of GPU Memory Extension

GPUswap fulfills the functional goals set in the previous chapters by design: Our accounting and relocation mechanisms operate transparently and are compatible with any type of application, and our eviction policy supports different applications with minimal modifications and achieves fairness even for unmodified legacy applications. In addition, however, it was our goal to minimize the overhead experienced by applications. Since low overhead is a non-functional goal, its fulfillment must be verified through experiments. GPUswap's overhead consists of three main components: i) The overhead of using system RAM in place of GPU memory after eviction, ii) the allocation latency induced by our eviction mechanism if insufficient GPU memory is available, and iii) the latency of our accounting mechanism, which induces latency even if sufficient GPU memory is available. In addition to quantifying GPUswap's overhead, our experiments also demonstrate that GPUswap can keep applications running even in extreme low-memory conditions.

6.1 Experimental Setup

In the experiments presented in this chapter, we used the same setup as for the profiling experiments presented in Section 4.4. The test system used in our experiments consists of a Core i7-4770 CPU clocked at 3.4 GHz, 16 GiB of RAM and an Nvidia GeForce GTX 480 GPU. In all our experiments, both CPU and GPU were locked to the highest available clock frequency. Our machine ran Ubuntu 12.04 and Linux version 3.5.7. Besides the pscnv GPU driver, into which our prototype implementation is integrated, we used Gdev's user space library and CUDA runtime, both modified to pass priorities to our eviction policy. Our prototype includes both the random selection and the priority-based buffer

selection policies; the priorities used by the latter were generated as described in Section 5.3.2. The software stack used for the experiments in this chapter did not include any of the profiling-specific modifications described in Chapter 4.

To study the behavior of GPUswap with varying amounts of available GPU memory, we added a memory limiting mechanism to pscnv. Upon loading pscnv's kernel module, this mechanism allows the user to pass a memory threshold to pscnv as a module parameter. Pscnv's memory allocator then ignores all physical GPU memory above that threshold.

In all experiments presented in this chapter, we used the same eight applications from the Rodinia Benchmark Suite [8] as in our experiments in Section 4.4. We chose these applications because they had been previously modified to support the CUDA driver API [37], which is the only API supported by Gdev's CUDA runtime. In each experiment, we ran two concurrent instances of each application to ensure that were multiple processes competing for GPU memory. Using two instances of the same application has the advantage of both instances running for a similar amount of time – if two applications with different runtimes were used instead, one of the applications would run alone on the GPU for part of its runtime, which would distort any measurements of the overhead experienced by the application. In addition, we extended the runtime of some of our benchmark applications since the original runtimes of some applications were short enough for one instance to finish execution before the second instance was fully started. Specifically, we added a loop around the main computation of those applications that do not naturally execute multiple iterations. This loop repeats the application's GPU kernels, DMA transfers and any CPU computation performed by the application – but not its memory allocations – 100 times. For applications which do execute a number of iterations, we set the iteration count high enough to ensure that the application runs for several seconds. Finally, to obtain runtimes with different sets of evicted chunks, we repeated the entire execution of each application ten times, simultaneously starting both application instances in each repetition. Unless stated otherwise, the numbers reported are the average of these ten runs, and error bars denote the standard deviation over all ten runs.

Since our prototype only supports a single chunk size for the entire system, we chose to conduct all experiments with a single chunk size. Specifically, we configured GPUswap's chunk size to 4 MiB for all experiments, which optimizes the eviction latency in most cases as shown in Section 6.4. While this chunk size is suboptimal for some applications, using the optimal chunk size in each experiment would only improve the results of GPUswap further.

For comparison, we repeated some of our measurements using the Gdev kernel module [36]. Compiling Gdev as kernel module moves the entire code base of Gdev's user space library into the kernel, while the user space library itself becomes a stub translating Gdev's API calls into system calls. The only significant

difference between the kernel module and the user space library is that the former optionally includes Gdev's scheduler and swapping mechanism.

By default, Gdev's scheduler allows applications to have multiple GPU kernels queued in their command submission channels at the same time. While this setting reduces Gdev's scheduling overhead, it can also lead to Gdev's swapping mechanism evicting buffers from an application with kernels still awaiting execution. In our experiments, we therefore configured Gdev to allow only one GPU kernel in the GPU's command submission channels at any given time.

While Gdev claims to support pscnv as the underlying driver even when compiled as a kernel module, we were unable to create a working setup with Gdev's kernel module and the pscnv driver. Therefore, we instead run Gdev on top of the Nouveau GPU driver [65]. We fixed numerous bugs in Gdev's original implementation, but made no functional modifications to Gdev's original scheduler or swapping mechanism. To be able to examine Gdev's behavior with different amounts of available GPU memory, we also added the same memory limiting mechanism to Gdev that we previously added to pscnv.

6.2 Application Runtime

Arguably the most important metric for the performance of GPUswap is the runtime overhead induced in applications. To quantify this overhead, we measured the runtime of each application's computation phase, which includes all GPU kernels, DMA operations and CPU computation launched by the application. In this series of experiments, we excluded the time taken by the application's memory allocation requests; GPUswap's overhead during memory allocation requests is evaluated separately in the next section.

Figures 6.1 and 6.2 show the total runtime of all repetitions of each application's computation and I/O with different amounts of GPU memory available. We measured these runtimes for the random selection and the priority-based chunk selection policies of GPUswap as well as for Gdev. For comparison, we ran each application another 100 times under the random selection policy, recording the best runtime we encountered during these 100 runs. This best runtime can serve as an approximation of the best possible result that any policy could obtain since given enough runs, the random selection policy will eventually select the optimal set of chunks for each application.

Scheduling Overhead

One of the main goals of GPUswap is not to induce any overhead if sufficient GPU memory is available, which we expect to be the common case in produc-

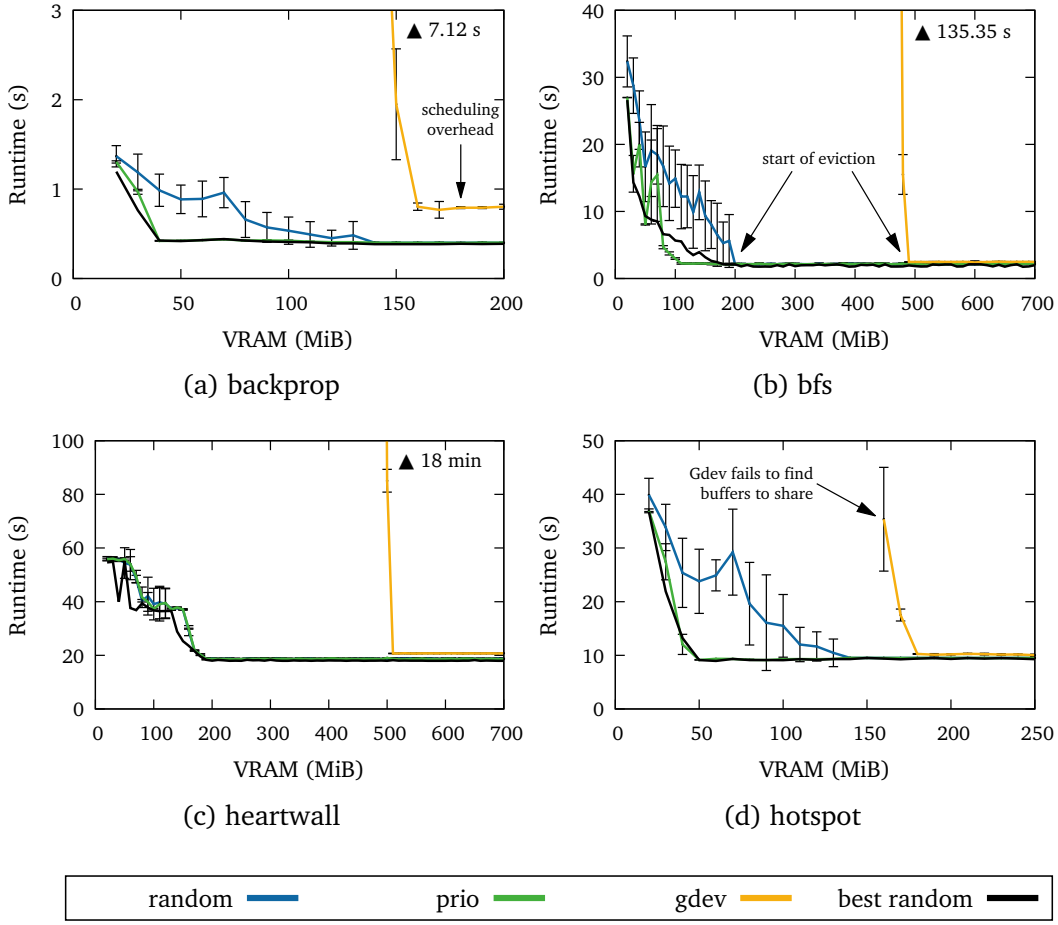


Figure 6.1: Runtime of various applications under GPUswap and Gdev. The times in this figure are the total runtime of all repetitions of the application's computation and I/O. "Random" shows the runtime of each application using GPUswap's random selection policy, while "prio" shows the same runtime under the priority-based chunk selection policy. "Gdev" shows the runtime under Gdev's original scheduler and swapping mechanism. For comparison, "best random" shows the best runtime we encountered during 100 runs of the application under our random chunk selection policy, which serves as an approximation of the best runtime obtainable by any policy.

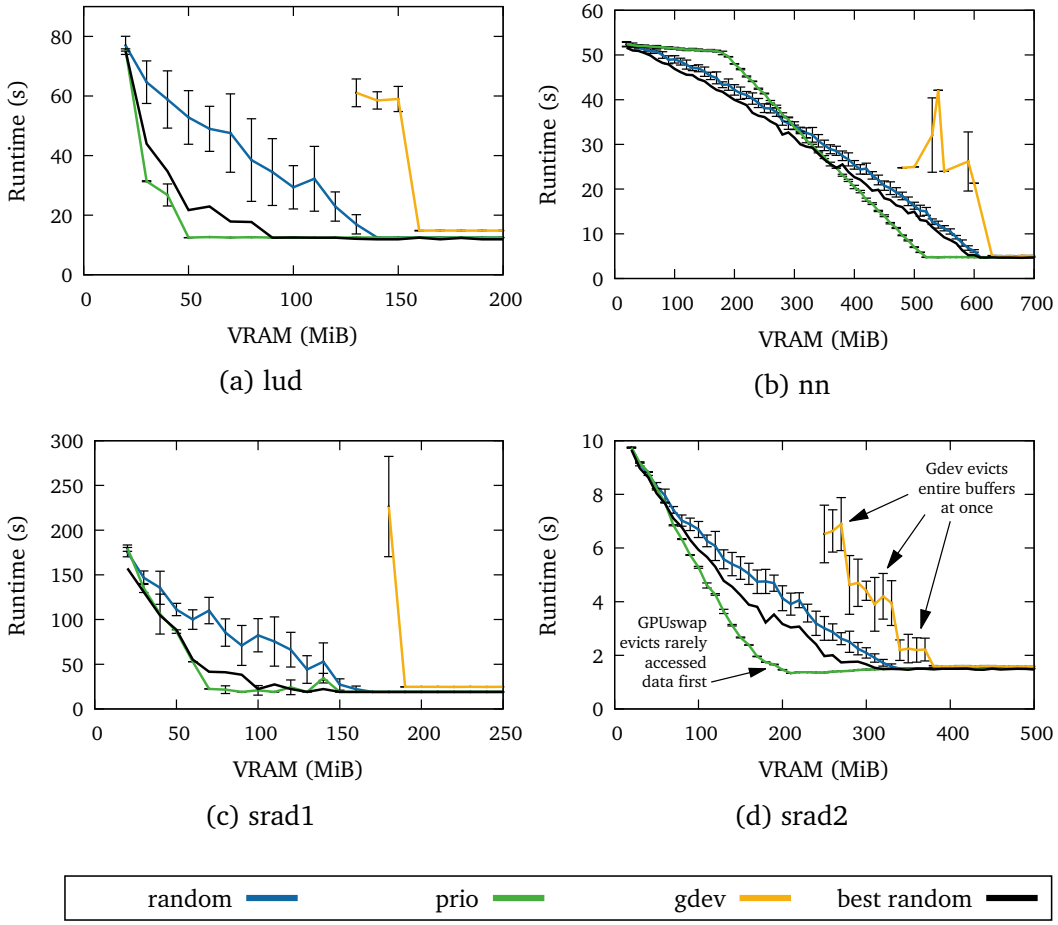


Figure 6.2: Runtime of various applications under GPUswap and Gdev. The times in this figure are the total runtime of all repetitions of the application's computation and I/O. "Random" shows the runtime of each application using GPUswap's random selection policy, while "prio" shows the same runtime under the priority-based chunk selection policy. "Gdev" shows the runtime under Gdev's original scheduler and swapping mechanism. For comparison, "Best Random" shows the best runtime we encountered during 100 runs of the application under our random chunk selection policy, which serves as an approximation of the best runtime obtainable by any policy.

tion systems. This condition corresponds to the right-hand side of the plots in Figures 6.1 and 6.2. The runtime of all applications is constant in that region since no evictions take place. Compared to GPUswap, however, Gdev causes a slowdown between 4.6 % (nn) and 95 % (backprop) in this region. The cause of this result is Gdev's software scheduling of GPU kernels, which is active regardless of the amount of GPU memory available: When we disabled Gdev's scheduler at compile time – which completely removes the scheduler from Gdev's code base, but also disables Gdev's swapping mechanism – Gdev did not exhibit any slowdown compared to pscnv. Note that Gdev's scheduling overhead is inflated by the settings used in our experiments: We allowed Gdev to keep only one GPU kernel in the GPU's command submission channels at any time since any other setting would be incompatible with Gdev's swapping mechanism. However, even when we allowed Gdev to keep multiple kernels in flight concurrently, we still measured scheduling overheads of up to 17 %.

Eviction Overhead

When GPU memory becomes scarce, both Gdev and GPUswap must evict data from the GPU to system RAM. Since these evictions unavoidably cause application overhead, the runtime of all applications eventually starts to increase as the amount of available GPU memory is reduced. However, the runtime of all applications increases much more steeply under Gdev than it does under GPUswap even when using the random selection policy. This result indicates that Gdev's strategy of copying data to the GPU before launching GPU kernels is inefficient since it causes Gdev to frequently copy large amounts of data – which may not even be needed by the next kernel to run – between CPU and GPU. In contrast, GPUswap causes only data that is actually accessed by the application to be transferred over the PCIe bus. Consequently, the runtime of all applications rises less steeply with GPUswap, regardless of the policy in use.

While GPUswap causes the applications' runtimes to increase linearly as the amount of available GPU memory decreases, the increase in runtime caused by Gdev occurs in steps, which can be seen most clearly for srad2 in Figure 6.2(d). This stepping is caused by the fact that Gdev can only evict entire buffers: Once Gdev detects a shortage of GPU memory, it evicts the contents of an entire application buffer, even if the amount of missing GPU memory is smaller than the buffer's size. Since the buffer's contents are then copied between GPU memory and system RAM on each kernel launch, this eviction causes a steep rise in the application's runtime. Subsequently, Gdev does not need to evict more data until the entire memory freed by evicting the first buffer has been allocated for other buffers. Once that is the case, however, Gdev must evict the contents of a second buffer, causing another steep increase in runtime.

From the figures, it can also be seen that GPUswap can apparently tolerate lower amounts of available GPU memory than Gdev without evicting data: For all applications, Gdev causes the runtime of all applications to start increasing with a larger amount of available GPU memory than GPUswap. This result is an artifact of our experimental setup: Both Gdev and the underlying GPU driver allocate some GPU memory internally, which is subsequently unavailable to applications. However, Nouveau and the Gdev kernel module appear to allocate a larger amount of memory than pscnv and the Gdev library, thus leaving less GPU memory to applications and causing Gdev to start swapping earlier than GPUswap. We do not consider this result to be our contribution since GPUswap would likely suffer from the same problem if it were integrated into Nouveau instead of pscnv.

Low Memory Conditions

Even if large overheads due to eviction of data to system RAM are unavoidable, keeping applications running slowly is preferable to not running them at all. In Figures 6.1 and 6.2, however, it can be seen that Gdev is unable to execute any applications if the amount of available GPU memory falls below a certain threshold. The cause of this result is that Gdev requires all data in the address space of an application to be in GPU memory before launching one of the application's kernels. By design, Gdev is therefore unable to launch any GPU kernels if the amount of available GPU memory is insufficient to hold all data of at least one application. In contrast, GPUswap does not strictly require any data to be in GPU memory since evicted data remains accessible to the application at any time. Consequently, GPUswap was able to execute all applications with as little as 20 MiB of available GPU memory.

Since we ran two instances of the same application – both of which consume the same amount of memory – Gdev should be able to keep applications running with up to half the amount of GPU memory at which swapping first became necessary. In the figures, however, it can be seen that Gdev was often unable to execute applications with more than half of the amount of GPU memory where swapping first occurred still available. This result is caused by a flaw in Gdev's swapping algorithm: If one of the applications attempts to allocate a small buffer for which insufficient GPU memory is available, Gdev may select a larger buffer from another application as the allocated buffer's sharing partner. If the first application subsequently allocates a larger buffer, Gdev may be unable to find a sufficiently large buffer to share with since the large buffer in the second application's address space is already shared. Even if this flaw was corrected, however, Gdev would still require larger amounts of available GPU memory than GPUswap since in contrast to GPUswap, Gdev requires the working set of at least one running application to fit in GPU memory.

Our prototype does not currently support amounts of GPU memory smaller than 20 MiB since it keeps system data structures, such as command submission channels, in GPU memory. In principle, however, it is possible to allocate some of these data structures in system RAM as well. Therefore, GPUswap will be able to run applications even with even lower amounts of GPU memory available once support for these system data structures is added to our prototype.

The Benefit of Policy

While our experiments have shown that GPUswap significantly outperforms Gdev even when using our random selection policy, that policy still induces considerable overhead as soon as any data is evicted from the GPU. In comparison, our priority-based buffer selection policy reduces the eviction overhead significantly: For most applications, the average runtime under the priority-based policy was not only consistently lower than under the random selection policy, but generally came close to or even outperformed the best runtime seen in 100 runs of random selection. For all applications except heartwall and srad1, the priority-based policy was even able to evict around 100 MiB of data without causing any slowdown at all: In all our experiments, our priority-based policy evicted the stack buffer first, which has only a negligible impact on the application's performance.

Once the stack buffer had been fully evicted, application runtimes did increase for the priority-based policy as well. However, this increase tended to be less steep initially than with random selection, only growing steeper towards the left of the figures as the policy was eventually forced to evict chunks with higher priorities. Finally, with only small amounts of available GPU memory – corresponding to the far left of the figures – the runtimes under the two policies became nearly indistinguishable. At that point, almost all application data resides in system RAM, leaving neither policy with much choice about which chunks to evict.

Throughout our experiments, the applications' runtimes under the priority-based policy showed much smaller standard deviations than under the random selection policy. Since our priority-based policy selects chunks from the decision set – all of which share the same priority – at random, these small standard deviations indicate that which chunks are chosen for eviction has little impact on the application's performance as long as the selected chunks share a similar number of accesses.

Overall, these results show that an eviction policy can significantly reduce the overhead associated with evicting GPU data to system RAM. However, some applications – such as heartwall and bfs – did show large standard deviations under the priority-based policy as well when low amounts of GPU memory were available. In such low-memory situations, the best runtime observed with random selection was also lower than the average runtime under our priority-based policy for some applications. These results indicate that our priority-based buffer selection policy may still leave room for improvement in some cases.

Outliers

While the results of most of our benchmark applications were remarkably consistent, our measurements for heartwall and nn stand out. For heartwall, the priority-based policy appears to have no effect on the application's runtime. The reason for this result is that heartwall is heavily DMA-bound, spending only a small fraction of its total runtime on computation. Since we currently assign priorities only based on the number of memory accesses during computation, the impact of our policy on heartwall's runtime is therefore limited. However, our policy did not increase the application's runtime either compared to random selection, and GPUswap significantly outperforms Gdev regardless of the policy in use.

Our results for nn are more complex to interpret. As for most other applications, nn's runtime initially remains constant, indicating that the stack buffer is a good candidate for eviction for this application. Once the stack buffer is fully evicted, however, the application's runtime starts to increase more steeply than with random selection, causing random selection to eventually outperform the priority-based policy. With less than 180 MiB of GPU memory available, the runtime then starts to increase less steeply. These results suggest that nn's buffers are evicted in the wrong order: The first buffer to be evicted after the stack is apparently more important to the application's performance, thus causing a steeper increase in runtime than the last buffer to be evicted.

To understand this result, we re-examind nn's memory access profile shown in Figure 4.4(d). From that profile, it can be seen that nn allocates only two buffers: An input buffer which is only read, and an output buffer which is only written. The number of accesses per page is almost identical for these two buffers, except for a single page at the end of the input buffer which is not filled to capacity and therefore receives a lower number of accesses. Consequently, the input buffer was assigned a lower priority than the output buffer in our experiments and was thus evicted first. Since the number of accesses per page is almost identical for both buffers, this decision should have only a negligible impact on the application's performance; however, the results in Figure 6.2(b) appear as if evicting the input buffer has a much larger impact on the application's performance than evicting the output buffer. At first glance, this result suggests that read requests to system RAM have a larger impact on application performance than write requests.

To test whether read requests really have a larger impact on application performance than write requests, we repeated the previous experiment with the priorities of the input and output buffers manually exchanged. The results, depicted in Figure 6.3, show that with this change, the priority-based policy outperforms random selection in all cases. However, the application's runtime now increases more steeply during the eviction of the output buffer, though the difference between the two buffers is not as pronounced as in the previous experiment. We therefore

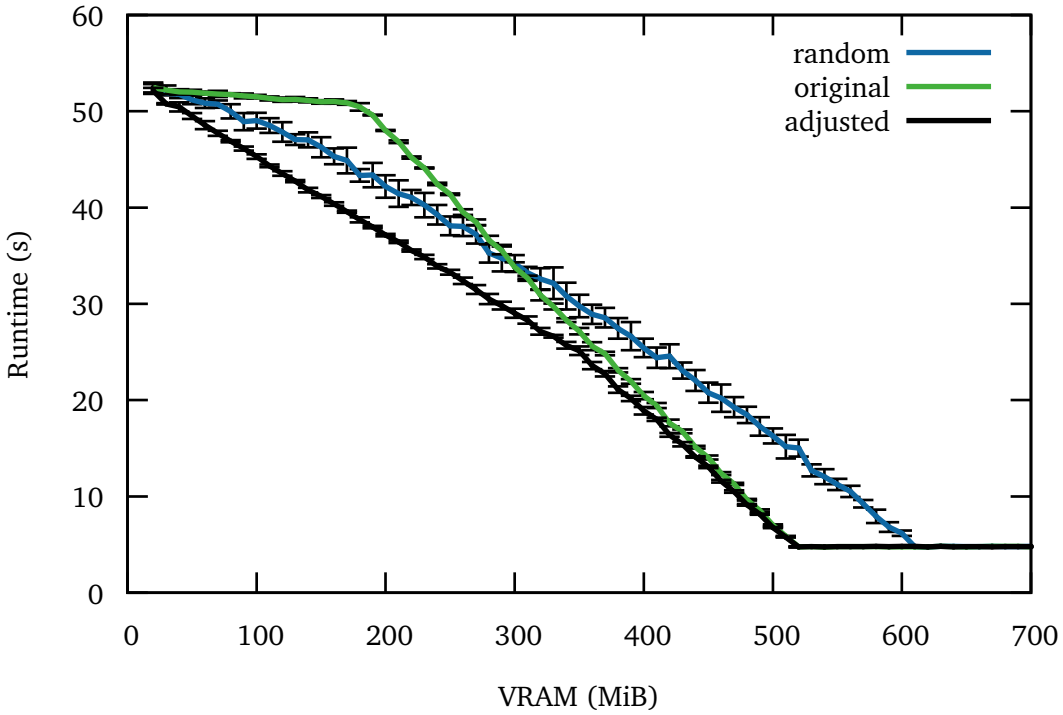


Figure 6.3: Runtime of nn with manually adjusted priorities. "Original" shows the results of our previous experiment shown in Figure 6.2(b). The steep increase in runtime between 200 and 500 MiB corresponds to the eviction of the input buffer, while the moderate increase below 200 MiB is caused by the eviction of the output buffer. "Adjusted" shows the runtime with the priorities of nn's input and output buffers manually exchanged – the output buffer is thus evicted first, causing the application's runtime to rise more steeply between 500 and 350 MiB than below 350 MiB. For comparison, "random" shows the application's runtime under the random selection policy.

conclude that there must be additional factors other than the type of access influencing the performance impact of each memory access. In any case, our results highlight that the number of memory accesses to a buffer may not accurately reflect the impact of the buffer's eviction on the application's performance.

6.3 Latency

In addition to the runtime overhead discussed in Section 6.2, GPUswap can induce delays in the application's execution on memory allocation requests: Our accounting mechanism must divide application buffers into chunks upon alloca-

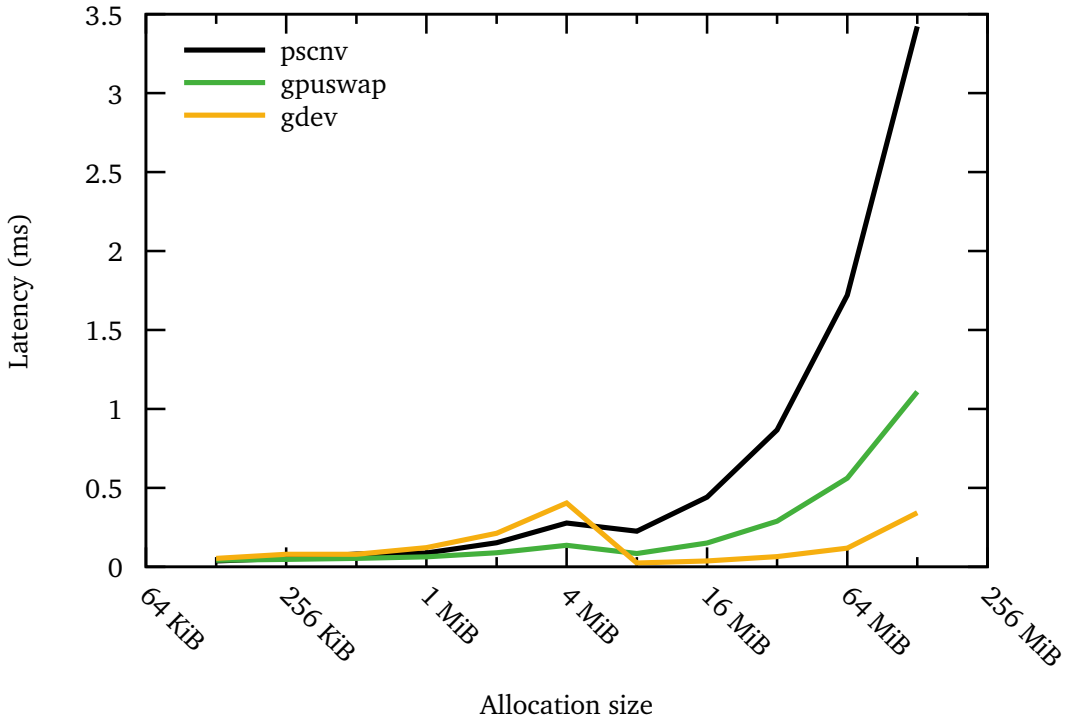


Figure 6.4: Latency of memory allocations of different sizes without eviction taking place. At each size, we measured the total time for 10000 allocations of the respective size. Since we did not measure individual requests, we cannot report standard deviations for these measurements. The numbers reported are the average latency of an individual allocation request of each size. For all measurements shown in this figure, there was a sufficient amount of GPU memory available.

tion, applications requesting memory may have to wait for an eviction to complete before their allocation requests are fulfilled, and other applications may have their GPU access suspended during an eviction if these applications are chosen as victims. While we expect allocation requests to be much less frequent than kernel launches, long delays could nonetheless be a problem for applications with real-time requirements. To quantify GPUswap's latency, we conducted experiments using the same eight benchmark applications as in the last section. In all experiments presented in this section, we limited the amount of available GPU memory to 100 MiB since eviction occurs in all applications at that amount.

6.3.1 Allocation Latency

Since GPUswap divides all buffers into chunks during allocation, GPUswap could potentially increase the latency of memory allocations compared to the unmodified

pscnv driver. To quantify the latency induced by GPUswap, we measured the latency of memory allocation if no eviction takes place. To that end, we created a microbenchmark which measures the time needed to allocate buffers of different sizes. At each size, this benchmark performs 10000 allocations, freeing the allocated buffer immediately after allocation to avoid running out of GPU memory. The benchmark measures the total time taken for all 10000 allocation/free cycles using the CPU's timestamp counter.

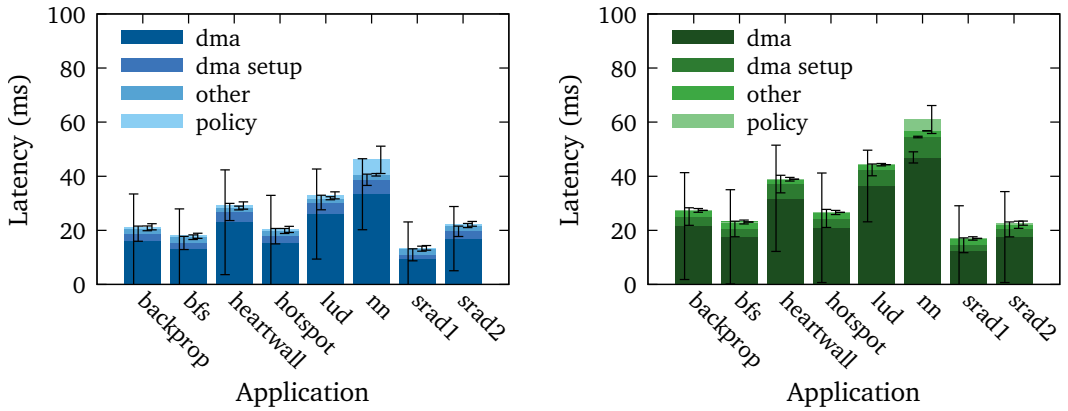
Figure 6.4 shows the average time per allocation/free cycle for GPUswap, Gdev and the unmodified pscnv driver. The results show that GPUswap actually outperforms the unmodified pscnv driver for allocations larger than 128 KiB. This result is counterintuitive since GPUswap performs additional work during allocations to divide buffers into chunks and should therefore take more time to allocate memory than pscnv. However, we made extensive modifications to pscnv and fixed numerous bugs while integrating GPUswap. Therefore, it is possible that we unknowingly fixed a performance bottleneck in the original implementation of pscnv during GPUswap's development. In any case, our results show that GPUswap's accounting mechanism does not lead to any significant performance degradation during memory allocation.

In addition to pscnv, GPUswap also outperformed Gdev for allocations smaller than 8 MiB. Between 4 and 8 MiB, however, Gdev's allocation latency drops significantly, causing Gdev to outperform both GPUswap and the unmodified pscnv driver from that point on. This result indicates that the memory allocator of the Nouveau driver, which the Gdev kernel module is built on, is more efficient than pscnv's for large allocation sizes.

Our results show that the allocation latency of all three drivers drops between 4 and 8 MiB. This drop is caused by an optimization for DMA transfers implemented in both the Gdev kernel module and the Gdev user space library: For small transfers, Gdev executes a copy loop on the CPU instead of submitting a DMA operation to the GPU. To speed up this copy loop, Gdev proactively maps all buffers of 4 MiB or less into the application's CPU address space at allocation time. Since setting up this mapping takes time, however, this optimization also increases the latency of allocations smaller than 4 MiB for all drivers.

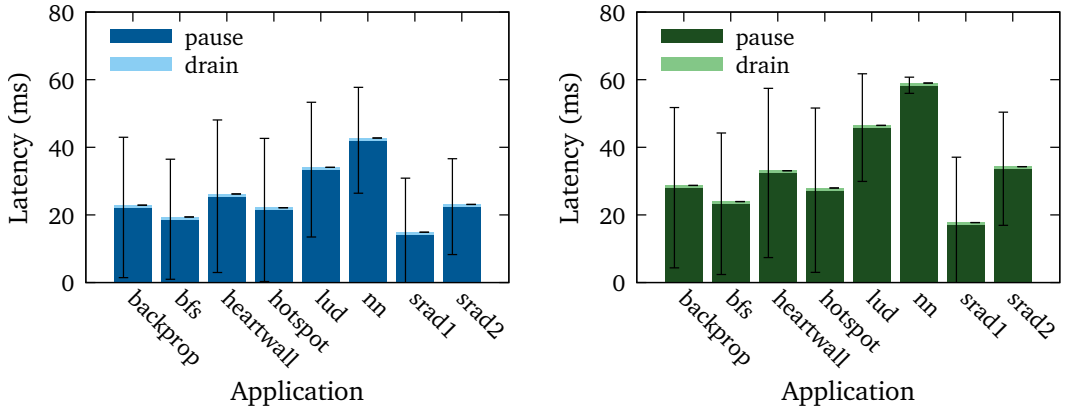
6.3.2 Eviction Latency

If there is insufficient GPU memory available to service an allocation request, GPUswap must evict data from GPU memory to system RAM, which adds additional latency to the allocation request. To quantify this latency, we used the CPU's time stamp counter to measure the time spent in GPUswap during evictions for both allocating and victim applications. For applications allocating memory, we measured the runtime of the policy in use and the runtime of GPUswap's eviction



(a) Random selection (b) Priority-based buffer selection

Figure 6.5: Average latency of eviction operations without contention from the allocating application's perspective. "Policy" shows the runtime of the respective eviction policy. "Dma setup" gives the time spent on setting up DMA transfers – such allocating space for evicted chunk in system RAM and mapping those chunks into the driver's address space – while "Dma" shows the time spent on the DMA transfers themselves. "Other" is the time spent in other parts of our eviction mechanism, such as waiting for command submission channels to drain or allocating chunks directly in system RAM.



(a) Random selection (b) Priority-based buffer selection

Figure 6.6: Latency of eviction operations without contention from a victim application's perspective. "Drain" shows the time needed to drain each application's command submission channels prior to eviction, while "pause" shows the time the application was unable to run kernels on the GPU due to the ongoing eviction.

mechanism itself; the latter includes waiting for victim applications' command submission channels to drain as well as the DMA transfers needed to relocate the chunks selected by the eviction policy. For victim applications, we measured the time needed to drain each application's command submission channels and the time during which the application's GPU access was suspended due to chunk transfers taking place. Note that draining the command submission channels does not constitute overhead from the victim application's point of view since the GPU still performs useful work on the victim's behalf during the draining phase. We did not perform any measurements for Gdev in this experiment: Since Gdev does not evict any data from the GPU during allocation requests, a comparison between Gdev and GPUswap would be meaningless.

In a first experiment, we quantified the latency of eviction in the absence of contention of any resources other than GPU memory. To that end, we created a dummy application allocating 80 MiB of memory – thus leaving 20 MiB for the benchmark application – which ensures that data from both the dummy and the benchmark application is evicted since the dummy application owns more than half of the available GPU memory. Apart from allocating memory, our dummy application does nothing else until the benchmark application terminates. We then ran each of our benchmark applications concurrently with our dummy application under both the random selection and the priority-based buffer selection policies, repeating each application's execution 100 times under each policy. Since our dummy application does not compete for PCIe bandwidth or any of the GPU's engines, the latency measured in this experiment constitutes a lower bound on the eviction latency in any scenario.

Figure 6.5 shows the results from an allocating application's perspective, while Figure 6.6 shows the results from the perspective of a victim application. As can be seen, the delay induced by GPUswap was typically below 60 ms for both allocating and victim applications, indicating that GPUswap can achieve acceptable latencies. The results also show that the latency measured in this experiment was heavily dominated by DMA transfers. However, our prototype leaves some potential for DMA latency optimization: The time spent on setting up DMA transfers – which could potentially be overlapped with other transfers if our prototype supported fully asynchronous DMA – accounts for about 14 % of the total latency. Since our dummy application never submits any commands to the GPU and all of our benchmark applications perform their allocations before launching the first kernel, no draining of command submission channels takes place. Finally, the runtime of both policies was negligible for all applications except nn. For nn, the ratio of policy runtime to eviction mechanism runtime is larger than for the other applications since nn causes larger amounts of data to be evicted at once than any other application. As a result, our policy must select a large number of chunks during the evictions triggered by nn, increasing the policy's runtime compared to other applications. Furthermore, most of the allocations performed by nn are larger

than the amount of GPU memory available in this experiment. As a consequence, GPUswap must always allocate some chunks of the newly allocated buffer directly in system RAM, which shortens the runtime of our eviction mechanism compared to other applications since no DMA is required to relocate those chunks.

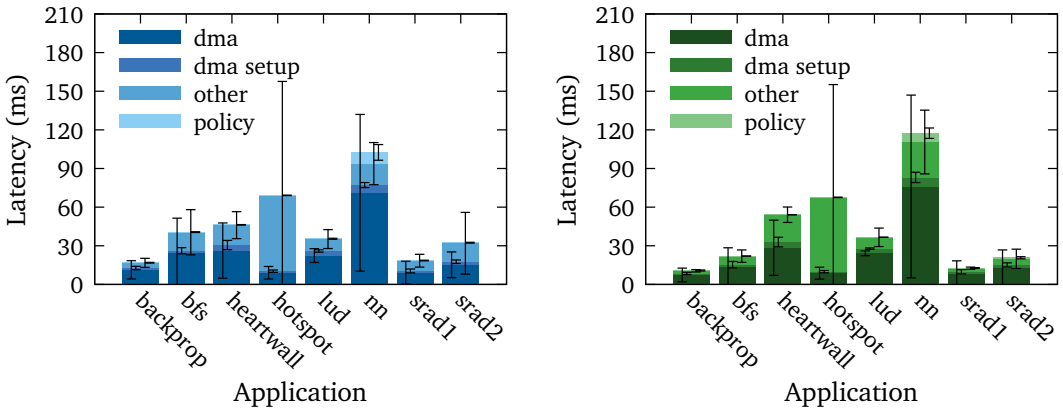
Throughout our measurements, we observed a large variance in the latency of the allocation requests, indicated by the large error bars in the figures. Three factors contribute to this latency: First, the application may allocate buffers of different sizes, which can lead to differences in latency between allocation requests since GPUswap may have to evict more data for larger buffers. Second, however, the amount of data that must be evicted does not only depend on the size of the allocated buffer, but also on the amount of available GPU memory: If some GPU memory is still available, the amount of data that must be evicted may be smaller than the requested allocation. As a consequence, GPUswap may induce different latencies in allocations of the same size. Third, the eviction policy may select chunks from the newly allocated buffer for eviction, which are then allocated directly in system RAM without the need for a DMA transfer. The total latency of the allocation request thus depends on the number of chunks from the newly allocated buffer that are chosen for eviction.

In contrast to our dummy application, applications in a production system do launch GPU kernels which compete with other applications for the GPU's engines or – if some of the application's data has been evicted – PCIe bandwidth. To study GPUswap's behavior in presence of such competition, we repeated the previous experiment with a second instance of each application replacing our dummy application. This setup replicates the one used in Section 6.2.

The results of this second experiment are shown in Figure 6.7 from an allocating application's perspective and Figure 6.8 from the perspective of a victim application. As can be seen, the latency experienced by most of the applications when allocating memory is increased under contention, but generally remains below 100 ms. While latencies of 100 ms can be problematic for real-time applications, applications are free to choose when to perform their memory allocations and can therefore allocate memory at times when latency is acceptable.

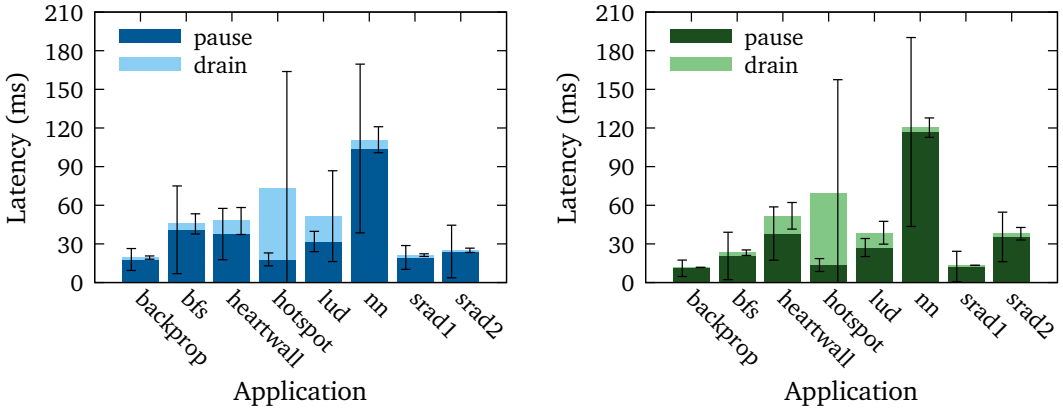
As in the previous experiment, the latency experienced by allocating applications was dominated by the runtime of the eviction mechanism, while the runtime of both policys was negligible. We therefore conclude that the runtime of the eviction policy is generally not an issue.

The latency experienced by victim applications is potentially more problematic since victim applications have no control over when their GPU access is suspended. However, the latency seen by victim applications was lower than that experienced by allocating applications: For most applications, the latency was below 60 ms, which we consider unproblematic since human observers typically perceive response times of less than 100 ms as instantaneous [45]. The main exception



(a) Random selection (b) Priority-based buffer selection

Figure 6.7: Average latency of eviction operations in the presence of contention from the allocating application's perspective. "Policy" shows the runtime of the respective eviction policy. "Dma setup" gives the time spent on setting up DMA transfers – such allocating space for evicted chunk in system RAM and mapping those chunks into the driver's address space – while "Dma" shows the time spent on the DMA transfers themselves. "Other" is the time spent in other parts of our eviction mechanism, such as waiting for command submission channels to drain or allocating chunks directly in system RAM.



(a) Random selection (b) Priority-based buffer selection

Figure 6.8: Average latency of eviction operations in the presence of contention from a victim application's perspective. "Drain" shows the time needed to drain each application's command submission channels prior to eviction, while "pause" shows the time the application was unable to run kernels on the GPU due to the ongoing eviction.

to this result was nn which causes particularly large amounts of memory to be evicted at once. Note that the time needed to drain the application's command submission channels does not constitute latency for victim applications since the GPU still performs useful work on behalf of the victim while the application's channels are being drained.

Since the command submission channels of victim applications are suspended only while chunks are being evicted from the application's address space, the latency experienced by victim applications consists almost exclusively of DMA. Therefore, we can imagine two potential ways to further reduce this latency: First, implementing support for fully asynchronous DMA would overlap some of the time spent on setting up DMA transfers – which accounts for 13 % of the total latency in this experiment – with the transfer of other chunks. Second, porting GPUswap to a newer GPU supporting PCIe 3.0 would speed up the DMA transfers themselves since PCIe 3.0 provides a higher bandwidth than the PCIe 2.0 interfaced used in our experiments.

Another notable difference to the previous experiment is an even larger variance in allocation latency. In addition to the three factors influencing latency described above, two more factors were present in this experiment: First, allocating applications may have to wait for the command submission channels of a victim application to drain. This draining can not only take longer than the actual eviction, but can also take varying amounts of time depending on the number of commands queued in the victim's command submission channels. This effect is most clearly visible for hotspot, which tends to build particularly long queues of kernels in its command submission channel. Second, since we ran two concurrent instances of each application in each repetition, it is possible for the two instances to compete for PCIe bandwidth: One instance may still be allocating memory while the second instance has already finished its allocations and launched its first GPU kernel, which may access evicted data over the PCIe bus. Our measurements have shown that the duration of individual chunk transfers can increase by up to a factor of 10 in that situation. Since this problem does not affected all transfers equally, it causes additional variance in the latency observed by applications. This second problem particularly affects bfs, nn and srad2.

6.4 Chunk Size

GPUswap operates on larger chunks instead of individual pages to reduce the processing overhead of eviction operations. However, the optimal size of these chunks is not intuitively clear. While our prototype allows the user to configure the size of these chunks through a module parameter, GPUswap must provide a default chunk size in case that parameter is not explicitly set. Ideally, this default size should yield acceptable results for all applications.

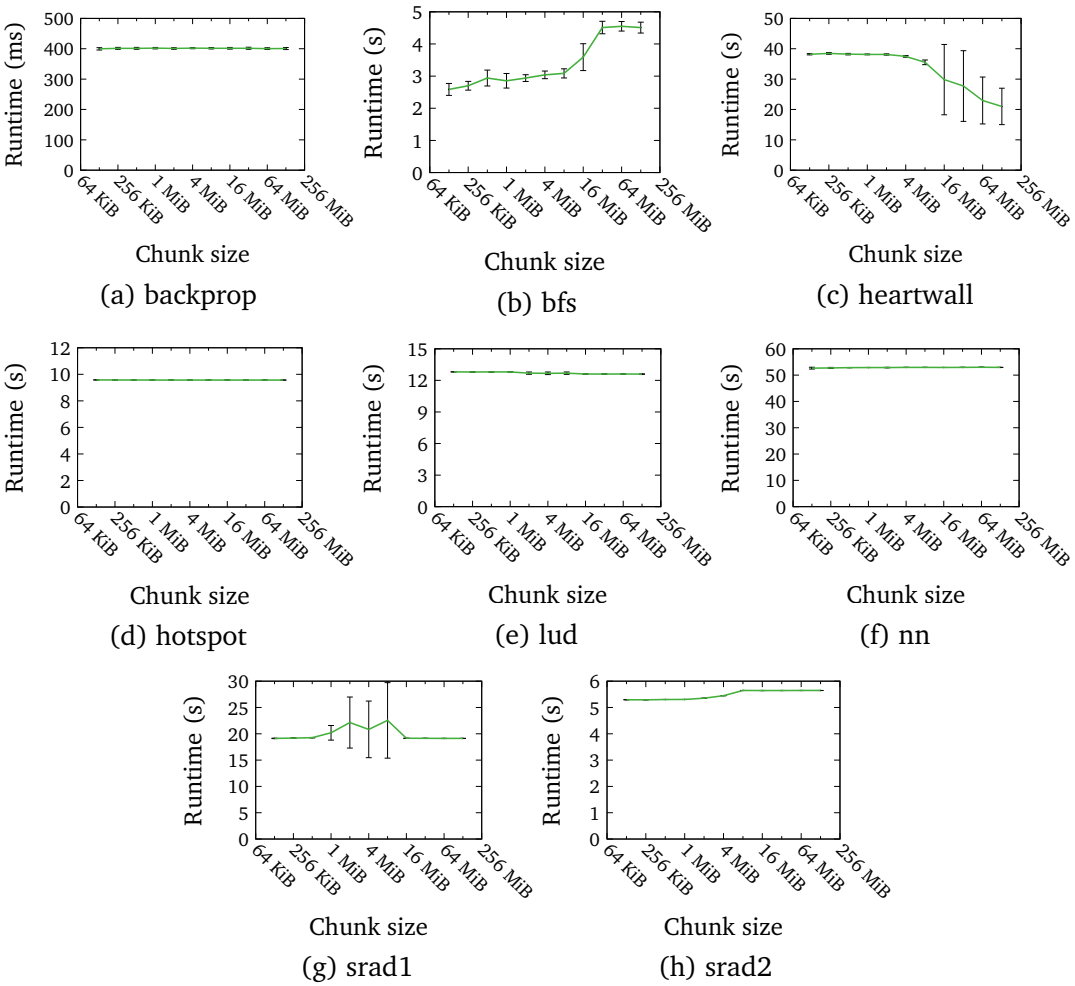


Figure 6.9: Runtime of various applications for different chunk sizes under the priority-based buffer selection policy with 100 MiB of GPU memory available. At each chunk size, we repeated each application's execution ten times; the numbers reported are the average of these ten runs, while the error bars indicate the standard deviation.

To determine the default chunk size, we measured the runtime as well as the allocation latency of our benchmark applications for different chunk sizes with the amount of available GPU memory limited to 100 MiB. As in the previous experiments, we repeated each application's execution ten times at each chunk size, running two concurrent application instances in each repetition.

Figure 6.9 shows the applications' runtimes for different chunk sizes ranging from 128 KiB – which corresponds to one large page and is currently the smallest chunk size supported by our prototype – to 128 MiB. The results show that the chunk size has little impact on the runtime of most applications, with the exception of bfs whose runtime increases at large chunk sizes, and heartwall for which large chunk

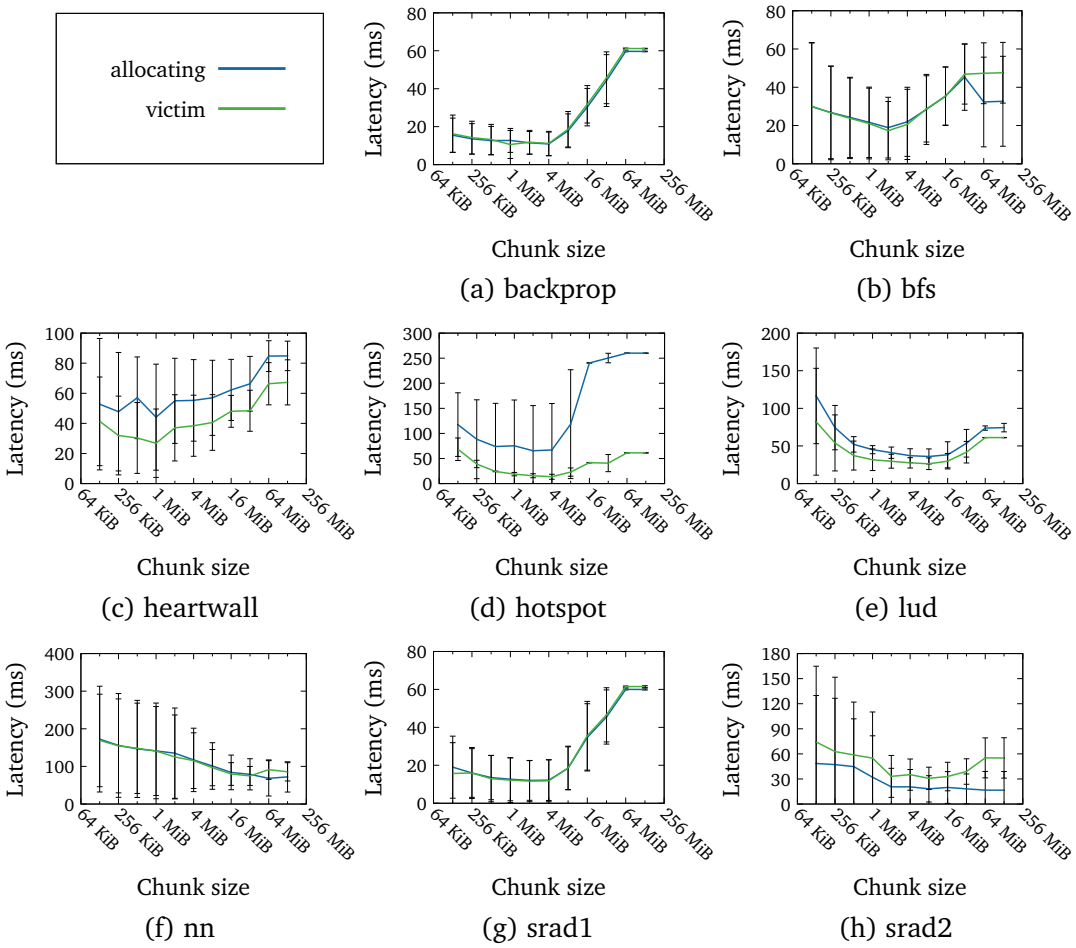


Figure 6.10: Average eviction latency experienced by various applications at different chunk sizes under the priority-based buffer selection policy with 100 MiB of GPU memory available. At each chunk size, we repeated each application's execution 100 times; the numbers reported are the average of these runs.

sizes cause the runtime to decrease. For bfs, the increase in runtime is caused by the fact that GPUswap may leave up to one chunk of GPU memory unused if a small allocation request triggers the eviction of a larger chunk. As a consequence, the amount of GPU memory available to the application is decreased at large chunk sizes, resulting in runtime overhead higher than necessary. For heartwall, the decrease in runtime is caused by the fact that GPUswap permits the amount of GPU memory allocated to each application instance to differ by up to one chunk. At large chunk sizes, one of the two instances in our experiment therefore received a much larger amount of GPU memory than the other, causing that instance to finish its execution early and subsequently leave the entire GPU to the second instance. While GPUswap's behavior thus caused a speedup in both instances at

large chunk sizes, the long error bars at large chunk sizes indicate that the two instances were not affected evenly. This result highlights that a fair distribution of resources does not necessarily optimize overall system performance.

Besides bfs and heartwall, srad1 also exhibited some sensitivity to the chunk size in use. While the majority of executions for this application ran for the same amount of time at each chunk size, we observed a small number of executions with increased runtimes at chunk sizes between one and eight MiB. At some point during these executions, our policy selected a remainder chunk for eviction. Since remainder chunks are smaller than the configured chunk size, GPUswap was then forced to evict an additional chunk to free up a sufficient amount of GPU memory. As a result, the application ran with a smaller amount of data in GPU memory than if no remainder chunk had been selected. The same effect is also visible in our previous measurements shown in Figure 6.2(c): During the eviction of the stack buffer of srad1, the application's runtime is increased for some memory sizes, while for others, it is identical to the runtime with all data in GPU memory. Since this effect is a corner case caused by a specific combination of memory size, chunk size and amount of memory used by the application, it can principally manifest in any application using any chunk size given the right circumstances. However, it is also possible that this effect results in a speedup: If selecting a remainder chunk brings the total amount of data evicted closer to the size of the allocation request that triggered the eviction than evicting only full chunks, the application will run with up to one chunk of additional GPU memory. In any case, the impact of this corner case is constrained to one chunk in either direction.

In addition to the applications' total runtime, we also measured the allocation latency experienced by the applications at different chunk sizes. Figure 6.10 shows the average latency experienced by both allocating and victim applications under the priority-based buffer selection policy for chunk sizes ranging from 128 KiB to 128 MiB. Our results show that for three of our benchmark applications – backprop, hotspot and srad1 – a chunk size of 4 MiB appears to optimize the allocation latency for both allocating and victim applications. For three more applications – bfs, lud and (for victim applications) srad2 – the latency is optimal at either 2 or 8 MiB, with 4 MiB giving good results for these applications. The remaining two applications prefer either smaller (heartwall) or larger (nn) chunks. Overall, these results indicate that 4 MiB is the optimal chunk size in terms of latency for this set of applications.

Based on our results, we chose a chunk size of 4 MiB for all our experiments to optimize the eviction latency. We chose to optimize for eviction latency since the chunk size appears to have no significant effect on the runtime of most applications. In addition, in those cases where the chunk size did cause significant changes in runtime, these changes were generally insignificant at a chunk size of 4 MiB, indicating that this chunk size does not cause fairness or the GPU memory's utilization to degrade. Note that using a single chunk size in all experiments

implies that some of our experiments were conducted using a sub-optimal chunk size. However, even using potentially sub-optimal chunk sizes, GPUswap outperformed Gdev significantly for all applications. Choosing the optimal chunk size for each application would only improve the results of GPUswap further.

6.5 Summary

Overall, our evaluation has shown that GPUswap fulfills our performance goals: First, GPUswap induces no application overhead as long as sufficient GPU memory is available since we do not rely on software scheduling of GPU kernels. In addition, when GPU memory becomes scarce, GPUswap induces less overhead than Gdev, indicating that GPUswap's strategy of keeping evicted data directly accessible to applications is more efficient than copying data to the GPU prior to kernel launch. Finally, GPUswap is able to keep applications running with much smaller amounts of available GPU memory than Gdev, which requires that all GPU data of each individual application must fit in GPU memory. The downside of our approach is that GPUswap adds significant latency to memory allocation requests, since applications must potentially wait for an eviction operation to complete before an allocation request can be served. This problem does not affect Gdev since Gdev never evicts any data from the GPU during memory allocation.

Our experiments have also shown that an eviction policy can further reduce the overhead associated with evicting GPU data to system RAM. The performance of our priority-based buffer selection policy is far superior to random selection. However, some of our results indicate that the policy still leaves room for improvement: In some cases, application runtimes have shown large standard deviations, and individual runs under the random selection policy have achieved lower runtimes than the priority-based policy did on average. These results indicate that application performance can still depend on the exact set of chunks chosen for eviction even though these chunks all share a similar number of accesses. In addition, the behavior of nn has shown that the number of memory accesses alone does not always fully capture the importance of pages to the application's performance. Further research into how memory bandwidth and latency affect the performance of GPU applications could thus yield further improvements to our eviction policy. However, the main problem of our prototype policy remains that the policy requires off-line profiling. Ideally, an eviction policy should operate purely on-line, without requiring any effort beforehand. We expect more research opportunities in this area to open up once GPU vendors include more hardware support for memory management – such as reference bits – in their hardware.

Finally, our experiments have shown that the chunk size used by GPUswap has no significant effect on application runtime in most cases, though large chunk sizes can be problematic in terms of both fairness and GPU memory utilization for

some applications. However, our results also show that the chunk size can greatly influence the latency of memory allocations, and that there is no single chunk size that optimizes this latency for all applications. Therefore, it may be worthwhile for GPUswap to support a different chunk size for each application. Currently, however, the optimal chunk size for each application can only be determined experimentally. Further investigation may yield a way to derive an application's optimal chunk size from profiling data or to tune each application's chunk size at runtime.

Chapter 7

Conclusion

Over the last few years, GPUs have become increasingly popular in computing since their massively parallel computing capability can bring tremendous speedups to a variety of applications. Consequently, all major cloud providers have included GPUs in their platforms. These platforms are typically virtualized, which increases the utilization of the underlying physical hardware by sharing this hardware between multiple customers. Since customers tend to under-utilize their hardware, virtualization also allows the providers to oversubscribe the physical hardware – i.e., offer their customers a larger amount of resources than physically available – to increase the physical hardware's utilization even further. The increased utilization in turn allows providers to offer access their hardware at low cost. If the physical hardware is oversubscribed, however, the provider must be capable of dealing with overload in case the customers do use all the resources they were promised.

In principle, the memory of modern GPUs can be oversubscribed easily since these GPUs support virtual memory. The cloud provider can thus offer a larger amount of virtual memory than is physically available, only allocating physical memory that is actually used by a customer. If the customers allocate a larger amount of memory than physically available, the provider can evict excess data from the GPU to system RAM. However, oversubscription of GPU memory is complicated by the asynchronous nature of contemporary GPUs: To reduce overhead, these GPUs typically allow the user to submit GPU kernels for execution directly to the GPUs command queues, bypassing the operating system. In addition, current GPUs assume all data that has been allocated in GPU memory to be accessible at any time. If a kernel attempts to access data that is not available, the resulting page fault is consequently treated as a fatal error, resulting in the termination of the faulting GPU kernel. Traditional methods of oversubscribing memory – such as swapping – thus cannot be applied to GPUs.

To overcome this issue, previous work has relied on software scheduling of GPU kernels to regain control over GPU kernel execution: A scheduler executing on the CPU selects the next GPU kernel to run when the previous kernel finishes execution. The operating system can then enable oversubscription by copying any data the next kernel might need into GPU memory prior to kernel launch. While this approach can successfully oversubscribe GPU memory, it comes with two main disadvantages: First, large amounts of data may be copied between CPU and GPU whenever a GPU kernel is started, including data that the kernel does not actually need. Second, software scheduling of GPU kernels disables the GPU's internal, highly efficient scheduling and context switching, thus inducing significant application overhead even if sufficient GPU memory is available.

In this thesis, we propose GPUswap, a novel eviction mechanism for GPU memory. In contrast to previous work, GPUswap triggers eviction operations on memory allocation requests instead of kernel launches: When an application attempts to allocate more GPU memory than is available, GPUswap evicts data from GPU memory to system RAM to make room for the allocation request. GPUswap then uses the virtual memory system present in modern GPUs to map the evicted data directly into the virtual address spaces of the application owning the data. Since all evicted data is thus directly accessible at any time, GPUswap can subsequently allow applications to submit their kernels directly to the GPU. Since GPUswap does not rely on software scheduling, GPUswap does not induce any overhead as long as sufficient GPU memory is available. In addition, GPUswap eliminates unnecessary copying since evicted data is only transferred over the PCIe bus when a GPU kernel actually accesses that data. Our evaluation has shown that the overhead induced by GPUswap is significantly lower than that of previous work regardless of the amount of GPU memory available.

While GPUswap should ideally evict rarely-accessed pages first, such pages are difficult to identify on current GPUs since these GPUs lack common features related to memory management, such as reference bits. Therefore, we instead use off-line profiling to identify rarely-accessed pages in the applications' address spaces. Previous work in this area relied on instrumenting the profiled application through a modified compiler, which has two main disadvantages: First, profiling is limited to the type of application supported by the modified compiler – e.g., CUDA applications. Second, only application code compiled by the modified compiler can be profiled, which can be a problem if an application uses shared libraries for which no source code is available. In contrast, our own profiling is based on the GPU's performance monitoring counters: By evicting a single page to system RAM and then counting the total number of accesses to system RAM from the profiled application, we can obtain the exact number of accesses to the evicted page. Repeating this process once for each page in the application's address space yields a complete access profile for the application's entire memory. Our profiler is not limited to a specific type of application and captures memory accesses from all

code in the application's address space, including shared libraries. While profiling using this method is slow since each of the application's GPU kernels must be repeated many times, we have shown that profiling time can be reduced either by exploiting application knowledge or by reducing profiling granularity.

Using our profiler, we examined several applications from the Rodinia Benchmark Suite. Our profiling showed that pages in the same application buffer tend to share a similar number of accesses, but the number of accesses often varies greatly between buffers. Consequently, we augmented GPUswap with an eviction policy which evicts rarely-accessed buffers first. Our policy operates in two steps: First, application developers profile their applications to determine which buffers are good candidates for eviction and assign a priority to each buffer allocated by the application based on the results of that profiling. In the second step, our policy then uses these priorities to select pages from rarely-accessed buffers for eviction whenever memory pressure occurs. Our evaluation has shown that compared to selecting pages randomly, this policy can greatly reduce the overhead associated with using system RAM in place of GPU memory.

7.1 Future Work

While GPUswap overcomes the limitations of previous work, our work has raised questions and uncovered challenges which have not yet been addressed.

Hardware and Application Support

Our current prototype builds on the pscnv driver, which has been unmaintained since 2012. The driver thus introduces two limitations to our prototype: First, our prototype is currently limited to CUDA applications. In principle, GPUswap supports any type of application, and measurements with different applications could yield additional insight into the strengths and weaknesses of our approach. Second, the pscnv driver limits our prototype to the Nvidia Fermi generation of GPUs, which only support PCI Express version 2. Newer versions of PCI Express provide higher bandwidth, which would likely reduce both the overhead associated with using system RAM in place of GPU memory and the eviction latency. Since neither of these limitations is conceptual, both can be overcome by integrating GPUswap into a different GPU driver.

Integration with Scheduling

While scheduling GPU kernels in software can induce significant application overhead, we expect some form of scheduling to be necessary in a shared environment to ensure fairness between different users. Recent works on GPU scheduling have

shown that fairness can be guaranteed with low overhead by leaving a degree of control to the GPU [44]. However, since such schedulers do not have full control over GPU kernel execution, they are typically incompatible with Gdev's approach of returning data to the GPU prior to kernel launch. In contrast, GPUswap does not depend on scheduling and should therefore be compatible to any GPU scheduler. Integrating GPUswap with scheduling also opens a range of new research opportunities in terms of policy, such as whether the scheduling priority provides information useful for an eviction policy.

Chunk Size

Our evaluation has shown that the latency of memory allocations under GPUswap is sensitive to the chunk size in use. However, there is no single chunk size that is optimal for all applications. While our current prototype only supports a single chunk size for all running applications, it is possible to extend GPUswap with support for individual chunk sizes for each application. Finding the correct chunk size for each application, however, remains an open problem. Further analysis may reveal application characteristics predicting the optimal chunk size. Alternatively, it may be possible to tune each application's chunk size at runtime.

Policy

While our policy significantly improves the overhead of GPUswap compared to random selection, our evaluation has revealed shortcomings in the way priorities are assigned: For some applications, the number of accesses to each page does not appear to reflect the impact of each page on the application's performance. In addition, we currently assign priorities based only on the number of memory accesses during computation, but disregard DMA. However, our priority-based policy is oblivious to the way priorities are generated. More detailed application profiling, including more factors than just the number of memory accesses, could lead to more accurate priorities, which our policy can then use without modification. In any case, however, we consider our priority-based policy to be a placeholder due to the manual effort required for profiling and assigning priorities. Ideally, an eviction policy should operate purely on-line, without requiring manual intervention. We therefore hope to replace our policy with a fully transparent policy once memory management-related features like reference bits are added to GPU designs.

Deutsche Zusammenfassung

Grafikkarten (Graphics Processing Units, GPUs) nehmen in der heutigen Informatik eine wichtige Rolle ein, da sie für bestimmte Arten von Anwendungen große Leistungsgewinne bei gleichzeitig hoher Energieeffizienz ermöglichen. Aus diesem Grund haben alle großen Cloudanbieter in den letzten Jahren GPUs in ihre Angebote integriert. Die Plattformen dieser Anbieter verwenden üblicherweise Virtualisierung, um physische Ressourcen zwischen mehreren Kunden aufzuteilen. Dieses Aufteilen erhöht die Auslastung der Ressourcen und verschafft dem Cloudanbieter so einen Kostenvorteil gegenüber dedizierter physischer Hardware. Um die Auslastung noch weiter zu erhöhen, vermieten heutige Cloudanbieter häufig mehr Ressourcen, als tatsächlich physisch zur Verfügung stehen. Für den Fall, dass die Kunden die angebotenen Ressourcen tatsächlich vollständig auslasten wollen, muss der Anbieter in diesem Fall aber in der Lage sein, das Funktionieren der Kundenanwendungen zu garantieren, selbst wenn der Ressourcenbedarf der Kunden die Kapazität der physischen Ressourcen übersteigt.

Der Speicher aktueller Grafikkarten lässt sich vergleichsweise einfach zwischen mehreren Kunden aufteilen, da diese Grafikkarten virtuellen Speicher ähnlich dem der CPU unterstützen. Der Anbieter kann so jedem Kunden einen großen, virtuellen Adressraum zur Verfügung stellen, muss aber nur so viel physischen Speicher bereitstellen, wie die Kunden tatsächlich verwenden. Falls der Anbieter mehr Speicher anbieten will, als physisch vorhanden ist, ist es grundsätzlich auch möglich, im Fall einer Überlastung des Grafikspeichers Daten in den Hauptspeicher des Systems auszulagern. Dieses Auslagern wird aber durch die asynchrone Arbeitsweise aktueller GPUs erschwert: Anwendungen können GPU-Kernels zur Ausführung direkt an die GPU senden, ohne dafür das Betriebssystem aufrufen zu müssen. Das Betriebssystem hat so keine Kontrolle über den Ausführungszeitpunkt der GPU-Kernels. Darüber hinaus gehen aktuelle GPUs davon aus, dass sämtlicher Grafikspeicher, der einmal von einer Anwendung angefordert wurde, jederzeit zugänglich ist. Sollte ein Kernel versuchen, auf eine nicht zugängliche Adresse zuzugreifen, behandelt die GPU diesen Zugriff als fatalen Fehler und beendet die Ausführung des Kernels.

Bisherige Ansätze umgehen dieses Problem, indem sie einen Software-Scheduler für GPU-Kernels einsetzen, um die Kontrolle über den Ausführungszeitpunkt der Kernels zurückzugewinnen. Bei dieser Methode wird nach Beendigung jedes Kernels der nächste Kernel auf der CPU in Software ausgewählt und an die GPU gesendet. Sind Daten, auf die der nächste Kernel möglicherweise zugreift, von der GPU in den Hauptspeicher ausgelagert worden, kopiert der Scheduler diese Daten zurück auf die GPU, bevor der Kernel gestartet wird. Der entscheidende Nachteil dieses Ansatzes ist, dass der Software-Scheduler das extrem effiziente interne Scheduling und Context Switching der GPU ersetzt, ohne das gleiche Maß an Effizienz zu erreichen. Ansätze, die auf Software-Scheduling basieren, verursachen daher einen hohen Overhead, und zwar auch dann, wenn eine ausreichende Menge Grafikspeicher zur Verfügung steht. Da der Scheduler darüber hinaus keine Möglichkeit hat, festzustellen, auf welche Daten ein GPU-Kernel tatsächlich zugreift, werden mit diesem Ansatz häufig Daten kopiert, die gar nicht benötigt werden.

In der vorliegenden Arbeit entwickeln wir einen alternativen Ansatz, um Auslagern von GPU-Daten zu ermöglichen. Unser Auslagerungsmechanismus, genannt GPUswap, blendet alle ausgelagerten Daten direkt in den GPU-Adressraum der jeweiligen Anwendung ein. Da auf diese Art alle Daten jederzeit zugänglich sind, kann GPUswap den Anwendungen weiterhin erlauben, Kommandos direkt an die GPU zu senden. Da unser Ansatz ohne Software-Scheduling auskommt, verursacht GPUswap keinerlei Overhead, solange Grafikspeicher in ausreichender Menge zur Verfügung steht. Falls tatsächlich Daten in den Hauptspeicher ausgelagert werden müssen, eliminiert GPUswap außerdem unnötige Datentransfers zwischen Hauptspeicher und GPU, da nur ausgelagerte Daten, auf die Anwendung tatsächlich zugreift, über den PCIe-Bus übertragen werden.

Auch wenn GPUswap im Vergleich zu vorherigen Ansätzen deutlich weniger Overhead verursacht, ist der Overhead, der durch die Verwendung von Hauptspeicher anstelle von Grafikspeicher verursacht wird, immer noch erheblich: Anwendungen greifen auf ausgelagerte Daten über den PCIe-Bus zu, der über eine erheblich geringere Bandbreite verfügt als der Grafikspeicher. Um diesen Overhead zu reduzieren, sollten bevorzugt Speicherseiten ausgelagert werden, auf die selten zugegriffen wird. Solche Seiten zu identifizieren ist auf aktuellen GPUs allerdings nicht ohne Weiteres möglich, da die Hardwarefunktionen, die auf der CPU zu diesen Zweck normalerweise eingesetzt werden – z.B. Referenzbits – auf aktuellen GPUs nicht zur Verfügung stehen.

In der vorliegenden Arbeit verwenden wir stattdessen Profiling, um selten verwendete Speicherseiten zu identifizieren. Bisherige Ansätze zum Profiling von GPU-Speicher basierten auf modifizierten Compilern, die alle Speicherzugriffe der analysierten Anwendung transparent instrumentieren. Dieser Ansatz hat allerdings zwei Nachteile: Erstens können nur Anwendungen untersucht werden, die vom modifizierten Compiler unterstützt werden, und zweitens muss sämtlicher

Code der untersuchten Anwendung – inklusive verwendeter Bibliotheken – mit dem modifizierten Compiler übersetzt werden, da ansonsten Speicherzugriffe aus Anwendungsteilen, die mit einem anderen Compiler übersetzt wurden, für den Profiler nicht sichtbar sind.

Unser Ansatz verwendet die Performancezähler der GPU anstelle eines modifizierten Compilers. Unser Profiler lagert einzelne Seiten aus dem Grafikspeicher in den Hauptspeicher aus und verwendet anschließend die Performancezähler, um die Zahl der Hauptspeicherzugriffe der Anwendung zu zählen. Wird dieser Vorgang einmal für jede Seite im Adressraum der Anwendung wiederholt, so erhält man ein vollständiges Zugriffsprofil des gesamten Speichers in diesem Adressraum. Im Gegensatz zu vorherigen Arbeiten funktioniert dieser Ansatz mit beliebigen Anwendungen und erfasst automatisch sämtliche Bibliotheken im Adressraum der Anwendung. Eine Untersuchung von mehreren Anwendungen aus der Rodinia Benchmark Suite mithilfe unseres Profilers zeigt, dass sich die Zahl der Zugriffe pro Seite bei den meisten Anwendungen vor allem zwischen verschiedenen Speicherpuffern der Anwendung unterscheidet, während Seiten innerhalb desselben Puffers meist eine ähnliche Zahl von Zugriffen aufweisen.

Ausgehend von den gesammelten Profilen untersuchen wir mehrere mögliche Auslagerungsstrategien und ihre Anwendbarkeit auf aktuellen GPUs. Unser Prototyp enthält zwei dieser Strategien: Eine wählt auszulagernde Seiten zufällig aus, während die andere einen prioritätsbasierten Ansatz verwendet. Bei der prioritätsbasierten Strategie weist der Benutzer ausgehend von einem Zugriffsprofil der Anwendung jedem Puffer der Anwendung eine Priorität zu. Die Auslagerungsstrategie wählt dann bevorzugt Speicherseiten aus Puffern niedriger Priorität. Experimente mit beiden Strategien zeigen, dass der prioritätsbasierte Ansatz den Overhead von GPUswap im Vergleich zu zufälliger Auswahl nicht nur deutlich reduziert, sondern sogar in der Lage ist, größere Datenmengen ohne jeden Overhead auszulagern.

List of Figures

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, and M. Isard. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, Savannah, GA, USA, Nov. 2016. USENIX Association. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual, Volume 2: System Programming*. Sept. 2012. URL: https://developer.amd.com/wordpress/media/2012/10/24593_APM_v2.pdf.

[3] Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Specification, Rev. 3.00, Dec. 2016. URL: https://support.amd.com/TechDocs/48882_IOMMU.pdf.

[4] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 607–618, Istanbul, Turkey, Mar. 2015. ACM. URL: https://dx.doi.org/10.1145/2694344.2694381.

[5] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 19–34, Salt Lake City, UT, USA, Mar. 2014. ACM. URL: https://dx.doi.org/10.1145/2541940.2541956.

[6] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A Virtual Memory Based Runtime to Support Multi-tenancy in Clusters with GPUs. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 97–108, Delft, The Netherlands, June 2012. ACM. URL: https://dx.doi.org/10.1145/2287076.2287090.

[7] S. Bhattacharya and D. Mukhopadhyay. Who Watches the Watchmen?: Utilizing Performance Monitors for Compromising Keys of RSA on Intel Platforms. In *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293 of *Lecture Notes in Computer Science*, pages 248–266. Springer, Sept. 2015. URL: https://dx.doi.org/10.1007/978-3-662-48324-4_13.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, IISWC '09, pages 44–54, Austin, TX, USA, Oct. 2009. IEEE. URL: https://dx.doi.org/10.1109/IISWC.2009.5306797.

[9] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '05, pages 273–286, Berkeley, CA, USA, May 2005. USENIX Association. URL: https://www.usenix.org/legacy/events/nsdi05/tech/full_papers/clark/clark.pdf.

[10] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, Nov. 2012. URL: https://dx.doi.org/10.1016/j.jpdc.2012.01.020.

[11] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Operating Systems Review*, 43(3):73–82, July 2009. URL: https://dx.doi.org/10.1145/1618525.1618534.

[12] R. Draves. Page Replacement and Reference Bit Emulation in Mach. In *Proceedings of the 1991 USENIX Mach Symposium*, pages 201–212, Monterey, CA, USA, Nov. 1991. USENIX Association. URL: https://www.usenix.org/publications/library/proceedings/mach91/draves.pdf.

[13] 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 *Proceedings of the 2010 International Conference on High Performance Computing Simulation*, HPCS '10, pages 224–231, Caen, France, June 2010. IEEE. URL: https://dx.doi.org/10.1109/HPCS.2010.5547126.

[14] Envytools. Tools for People Envious of Nvidia's Blob Driver, Nov. 2017. URL: https://github.com/envytools/envytools.

[15] J. Fotheringham. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *Communications of the ACM*, 4(10):435–436, Oct. 1961. URL: https://dx.doi.org/10.1145/366786.366800.

[16] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro. Data Transfer Matters for GPU Computing. In *Proceedings of the 2013 International Conference on Parallel and Distributed Systems*, ICPADS '13, pages 275–282, Seoul, South Korea, Dec. 2013. IEEE. URL: https://dx.doi.org/10.1109/ICPADS.2013.47.

[17] J. Fung and S. Mann. Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. In *2008 IEEE International Conference on Multimedia and Expo*, ICME '08, pages 9–12, Hannover, Germany, June 2008. IEEE. URL: https://dx.doi.org/10.1109/ICME.2008.4607358.

[18] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proceedings of the 2010 European Conference on Parallel Processing*, Euro-Par '10, pages 379–391, Naples, Italy, Sept. 2010. Springer. URL: https://dx.doi.org/10.1007/978-3-642-15277-1_37.

[19] Google, Inc. Deepdream, Nov. 2017. URL: https://github.com/google/deepdream.

[20] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-Overhead GPGPU Virtualization. In *Proceedings of the 4th International Workshop on Frontiers of Heterogeneous Computing*, FHC '13, pages 1721–1726, Zhangjiajie, China, Nov. 2013. IEEE. URL: https://dx.doi.org/10.1109/HPCC.and.EUC.2013.245.

[21] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing*, HPCVirt '09, pages 17–24, Nuremburg, Germany, Mar. 2009. ACM. URL: https://dx.doi.org/10.1145/1519138.1519141.

[22] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the 2010 ACM SIGCOMM Conference*, SIGCOMM '10, pages 195–206, New Delhi, India, Aug. 2010. ACM. URL: https://dx.doi.org/10.1145/1851182.1851207.

[23] S. Hand. Self-Paging in the Nemesis Operating System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 73–86, New Orleans, Louisiana, USA, Feb. 1999. USENIX Association. URL: https://www.usenix.org/legacy/events/osdi99/full_papers/hand/hand.pdf.

[24] M. Harris. Unified Memory in CUDA 6, Nov. 2013. URL: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[25] L. He, K. Li, L. Shi, J. Sun, and H. Chen. A Fast RPC System for Virtual Machines. *IEEE Transactions on Parallel & Distributed Systems*, 24(7):1267–1276, July 2013. URL: https://dx.doi.org/10.1109/TPDS.2012.199.

[26] M. Hillenbrand, M. Gottschlag, J. Kehne, and F. Bellosa. Multiple Physical Mappings: Dynamic DRAM Channel Sharing and Partitioning. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 21:1–21:9, Mumbai, India, Sept. 2017. ACM. URL: https://dx.doi.org/10.1145/3124680.3124742.

[27] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, Austin, TX, USA, June 2009. ACM. URL: https://dx.doi.org/10.1145/1555754.1555775.

[28] Intel Corporation. Intel® Xeon® Processor E7-8800/4800 v4 Product Families Brief, May 2016. URL: https://www.intel.com/content/www/us/en/processors/xeon/xeon-e7-8800-4800-v4-product-families-brief.html.

[29] Intel Corporation. Intel Virtualization Technology for Directed I/O – Architecture Specification, Rev. 2.5, Nov. 2017. URL: https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf.

[30] International Business Machines Corporation. *IBM Power Systems Performance Guide: Implementing and Optimizing*. IBM Redbooks. May 2013. URL: https://www.redbooks.ibm.com/abstracts/sg248080.html.

[31] B. Jacob and T. Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, July 1998. URL: https://dx.doi.org/10.1109/40.710872.

[32] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '11, Boston, MA, USA, Mar. 2011. USENIX Association. URL: https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Jang.pdf.

[33] JEDEC Solid State Technology Association. Graphics Double Data Rate (GDDR5X) SGRAM Standard, Aug. 2016. URL: https://www.jedec.org/sites/default/files/docs/JESD232A.pdf.

[34] F. Ji, H. Lin, and X. Ma. RSVM: A Region-based Software Virtual Memory for GPU. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 269–278, Edinburgh, Scotland, Sept. 2013. IEEE. URL: https://dx.doi.org/10.1109/PACT.2013.6618823.

[35] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, pages 675–678, Orlando, FL, USA, Nov. 2014. ACM. URL: https://dx.doi.org/10.1145/2647868.2654889.

[36] S. Kato. Gdev, Nov. 2014. URL: https://github.com/shinpei0208/gdev.

[37] S. Kato. Gdev Benchmarks, Apr. 2014. URL: https://github.com/shinpei0208/gdev-bench.

[38] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference*, USENIX ATC '11, pages 17–30, Portland, OR, USA, June 2011. USENIX Association. URL: http://static.usenix.org/events/atc11/tech/final_files/atc11_proceedings.pdf#page=27.

[39] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC '12, pages 401–412, Boston, MA, June 2012. USENIX Association. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final319.pdf.

[40] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Computer Architecture Letters*, 1(1):7–11, Jan. 2002. URL: https://dx.doi.org/10.1109/L-CA.2002.8.

[41] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. de Lara. VMM-independent Graphics Acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 33–43, San Diego, CA, USA, June 2007. ACM. URL: https://dx.doi.org/10.1145/1254810.1254816.

[42] J. Lawley. Understanding Performance of PCI Express Systems, Oct. 2014. URL: https://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

[43] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008. URL: https://dx.doi.org/10.1109/MM.2008.31.

[44] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, Salt Lake City, UT, USA, June 2014. ACM. URL: https://dx.doi.org/10.1145/2541940.2541963.

[45] R. B. Miller. Response Time in Man-Computer Conversational Transactions. In *Proceedings of the Fall 1968 AFIPS Joint Computer Conference, Part I*, AFIPS FJCC '68, pages 267–277, San Francisco, CA, USA, Dec. 1968. ACM. URL: https://dx.doi.org/10.1145/1476589.1476628.

[46] A. Mordvintsev, C. Olah, and M. Tyka. Inceptionism: Going Deeper into Neural Networks, June 2015. URL: https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html.

[47] Nvidia Corporation. CUDA Programming Guide. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#data-transfer-between-host-and-device.

[48] Nvidia Corporation. CUPTI Documentation. URL: https://docs.nvidia.com/cuda/cupti/index.html.

[49] Nvidia Corporation. NVIDIA Tesla P100, May 2016. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[50] Nvidia Corporation. Nvidia Tesla V100 GPU Architecture, Aug. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[51] Nvidia Corporation. Nvidia Tesla V100 GPU Accelerator, Mar. 2018. URL: https://images.nvidia.com/content/technologies/volta/pdf/437317-Volta-V100-DS-NV-US-WEB.pdf.

[52] J. J. K. Park, Y. Park, and S. Mahlke. Dynamic Resource Management for Efficient Utilization of Multitasking GPUs. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 527–540, Xi'an, China, Apr. 2017. ACM. URL: https://dx.doi.org/10.1145/3037697.3037707.

[53] PathScale, Inc. PathScale NVIDIA Graphics Driver, Aug. 2012. URL: https://github.com/pathscale/pscnv.

[54] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, Cascais, Portugal, Sept. 2011. ACM. URL: https://dx.doi.org/10.1145/2043556.2043579.

[55] N. Sakharnykh. Beyond GPU Memory Limits with Unified Memory on Pascal, Dec. 2016. URL: https://devblogs.nvidia.com/beyond-gpu-memory-limits-unified-memory-pascal/.

[56] L. Shi, H. Chen, J. Sun, and K. Li. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers*, 61(6):804–816, June 2012. URL: https://dx.doi.org/10.1109/TC.2011.112.

[57] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 11–22, New Orleans, LA, USA, Feb. 2012. ACM. URL: https://dx.doi.org/10.1145/2145816.2145819.

[58] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why not virtualizing GPUs at the hypervisor? In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 109–120, Philadelphia, PA, June 2014. USENIX Association. URL: https://www.usenix.org/system/files/conference/atc14/atc14-paper-suzuki.pdf.

[59] I. Tanasic, I. Gelado, M. Jorda, E. Ayguade, and N. Navarro. Efficient Exception Handling Support for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '17, pages 109–122, Cambridge, MA, USA, Oct. 2017. ACM. URL: https://dx.doi.org/10.1145/3123939.3123950.

[60] A. S. Tanenbaum and H. Bos. Page Replacement Algorithms. In *Modern Operating Systems*, pages 209–221. Pearson, fourth edition, 2015.

[61] K. Tian, Y. Dong, and D. Cowperthwaite. A full GPU virtualization solution with mediated pass-through. In *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 121–132, Philadelphia, PA, June 2014. USENIX Association. URL: https://www.usenix.org/system/files/conference/atc14/atc14-paper-tian.pdf.

[62] D. Vasilas, S. Gerangelos, and N. Koziris. VGVM: Efficient GPU capabilities in virtual machines. In *Proceedings of the 2016 International Conference on High Performance Computing & Simulation*, HPCS '16, pages 637–644, Innsbruck, Austria, July 2016. IEEE. URL: https://dx.doi.org/10.1109/HPCSim.2016.7568395.

[63] K. Wang, X. Ding, R. Lee, S. Kato, and X. Zhang. GDM: Device Memory Management for GPGPU Computing. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 533–545, Austin, TX, June 2014. ACM. URL: https://dx.doi.org/10.1145/2591971.2592002.

[64] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. c Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Proceedings of the 2012 IEEE Conference on Innovative Parallel Computing*, InPar '12, pages 1–12. IEEE, May 2012. URL: https://dx.doi.org/10.1109/InPar.2012.6339609.

[65] X.org foundation. Nouveau, Oct. 2017. URL: https://nouveau.freedesktop.org/.

[66] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC '16, pages 579–590, Denver, CO, USA, June 2016. USENIX Association. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/xue.

[67] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '13, pages 203–214, New York, NY, USA, June 2013. ACM. URL: https://dx.doi.org/10.1145/2462902.2462914.