GPrioSwap: Towards a Swapping Policy for GPUs

Jens Kehne, Marius Hillenbrand, Jonathan Metter, Mathias Gottschlag, Martin Merkel, Frank Bellosa

Karlsruhe Institute of Technology (KIT)
Operating Systems Group
os@itec.kit.edu

ABSTRACT

Over the last few years, Graphics Processing Units (GPUs) have become popular in computing, and have found their way into a number of cloud platforms. However, integrating a GPU into a cloud environment requires the cloud provider to efficiently virtualize the GPU. While several research projects have addressed this challenge in the past, few of these projects attempt to properly enable sharing of GPU memory between multiple clients: To date, GPUswap is the only project that enables sharing of GPU memory without inducing unnecessary application overhead, while maintaining both fairness and high utilization of GPU memory. However, GPUswap includes only a rudimentary swapping policy, and therefore induces a rather large application overhead.

In this paper, we work towards a practicable swapping policy for GPUs. To that end, we analyze the behavior of various GPU applications to determine their memory access patterns. Based on our insights about these patterns, we derive a swapping policy that includes a developer-assigned priority for each GPU buffer in its swapping decisions. Experiments with our prototype implementation show that a swapping policy based on buffer priorities can significantly reduce the swapping overhead.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Virtual Memory, Swapping*

Keywords

Virtualization; Memory Overcommitment; Oversubscription; Swapping; Profiling; GPU

1. INTRODUCTION

Over the last few years, *graphics processing units* (GPUs) have become 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 computer vision [8], cryptography [16], network packet processing [14] and even processing of general web requests [2]. More recently, cloud offerings including GPUs have emerged, which allow the cloud provider to increase the GPU's utilization and thus offer GPU computation time at competitive prices.

Integrating a GPU into a cloud environment, however, requires the cloud provider to efficiently virtualize the GPU, which creates interesting new challenges. How to share GPUs between mutually-untrusted applications – which is the common case in a cloud environment – has been subject to extensive research. However, these research projects have largely focused on reducing the virtualization overhead [27, 13, 9, 30], on expanding the functionality of virtual GPUs [6, 28, 29, 32, 33] or on providing fairness between mutually untrusted GPU applications [22, 19, 26].

A topic that has been mostly overlooked in previous research, however, is how to deal with the memory of a virtualized GPU. Cloud providers tend to oversubscribe their hardware – e.g., rent out more memory than is actually available – to further increase utilization. If customers decide to actually use all the memory they paid for, however, this kind of oversubscription can easily lead to memory shortages, which are difficult to handle on the GPU. While there are research projects that can handle GPU memory shortages by extending GPU memory with system RAM, these projects either require scheduling of all GPU kernels in software [20, 31] or manage the GPU's memory in a cooperative fashion [17, 3].

To mitigate these issues, our research group developed GPUswap [21], which achieves both fairness and high utilization of GPU memory at the same time, without relying on software scheduling of GPU requests. If memory pressure occurs on the GPU, GPUswap responds by swapping data from applications over-using their fair share of GPU memory to system RAM. Swapping on the GPU is fundamentally different from traditional swapping done in the CPU world: Since modern GPUs have the ability to access system RAM directly over the PCIe bus, swapped data is still accessible to the GPU. Therefore, there is no need to raise a page fault and return swapped pages to the GPU on access, as is typically done with pages swapped to a disk. However, accessing pages over the PCIe bus is significantly slower than accessing pages in GPU memory, and can thus incur high application overhead. The GPU thus resembles a NUMA system, with GPUswap migrating pages between NUMA nodes in response to memory pressure.

SYSTOR 2017, May 22-24, 2017, Haifa, Israel

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-5035-8/17/05…$15.00

DOI: https://dx.doi.org/10.1145/3078468.3078474

GPUswap's current swapping policy selects pages to swap at random, which is clearly far from optimal. Developing a swapping policy for GPUs is not straightforward since current GPUs lack several features found commonly in CPUs. For example, while today's GPUs support virtual memory, they typically do not support page faults, and their page tables do not include reference bits. Traditional page replacement policies known from the CPU world therefore do not apply to most GPUs. However, GPUs are currently growing closer to CPUs in terms of features: The latest generation of Nvidia GPUs includes page fault support, and may be able to set the CPU-side reference bit on DMA operations through the CPU's IOMMU [10]. Unfortunately, it is still unclear to what extent these features can be used for swapping since there is little documentation available about the hardware of those GPUs.

In this paper, we develop a practical swapping policy for GPUs which mitigates the overhead of using system RAM in case of memory pressure without relying on features that are typically not present in GPUs. Our policy, which we call GPrioSwap, is based on self-paging [15, 4, 7] using per-buffer priorities. GPrioSwap requires developers to profile their applications, and to assign priorities to application buffers based on the results of that profiling. GPrioSwap then swaps out pages from low-priority buffers first. Experiments with our prototype implementation show that GPrioSwap reduces the swapping overhead significantly compared to GPUswap's original random selection algorithm.

Specifically, we make the following contributions in this paper:

1. We devise a method for profiling the memory accesses of GPU applications which i) is not limited to a specific type of applications, and ii) is able to profile accesses to buffers not under application control.

2. We develop a policy for swapping data from the GPU to system RAM. To our knowledge, our policy is the first to target a GPU shared between multiple applications, and also the first to explicitly handle buffers allocated by the GPU runtime.

The rest of this paper is organized as follows: First, we present details about the architecture of current GPUs, which are necessary to understand our approach, in Section 2. In Section 3, we present our methodology for profiling the memory access patterns of GPU applications, the access patterns we observed and the implications of these patterns on policy. Next, we describe the design of GPrioSwap in Section 4. In Section 5, we evaluate the benefit of GPrioSwap over a random eviction policy. Finally, we present related work in Section 6, before concluding the paper in Section 7.

2. BACKGROUND

Current GPUs typically operate as asynchronous accelerators. Applications submit high-level commands to the GPU, which tell the GPU to, for example, launch CUDA kernels or initiate DMA operations. The GPU then processes the submitted commands autonomously, while the application is free to perform other work. Optionally, the application can configure the GPU to raise an interrupt after a given command finishes execution, which applications often use to be notified of kernel completion.

2.1 GPU Command Submission

Current GPUs allow applications to submit their commands directly to the GPU, bypassing the operating system to reduce overhead. To that end, these GPUs implement a number of in-memory command queues called *command submission channels*. The GPU driver maps these channels into an application's CPU address space, allowing the application to write new commands directly into the channel. A dedicated device register, which is also mapped into the application's CPU address space, serves as a doorbell for notifying the GPU that new commands have been submitted. Commands from a given channel are then executed in-order, whereas commands from different channels can be arbitrarily interleaved.

2.2 GPU Address Spaces

Since current GPUs can be accessed by multiple applications simultaneously and without operating system intervention, the GPU must implement a mechanism to isolate these applications from each other. To that end, modern GPUs support multiple *address spaces* similar to those found on the CPU. The GPU driver assigns each command submission channel to exactly one GPU address space; all commands originating from that channel then operate only on virtual addresses from that address space. The GPU's address spaces are defined through page tables managed by the GPU driver. These page tables map GPU-virtual addresses to physical addresses, which either reside in GPU memory or system RAM. The driver can thus map system RAM into the GPU address space of arbitrary applications without the application's knowledge.

While GPU virtual memory is similar to virtual memory on the CPU, GPUs typically lack basic features that are ubiquitous on the CPU. First, the GPU driver cannot determine which regions of GPU memory are frequently accessed since the GPU's page table does not include reference bits. Second, currently most GPUs neither support page faults nor preemption. The layout of a GPU address space can therefore be changed only while no code is being executed in that address space.

Fortunately, these limitations are already diminishing: The latest generation of Nvidia GPUs supports both preemption and page faults, which improves GPU memory management on simulated hardware [34]. In addition, recent IOMMUs have the ability to set the reference bit in the CPU's page table on DMA operations, which would allow a swapping policy to detect frequently-accessed pages in system RAM. However, these features are not supported on earlier generations of GPU hardware and are therefore not widely available today.

2.3 GPU Performance Counters

GPU programming often requires extensive tuning to make the code run efficiently. To aid such tuning efforts, current GPUs include performance counters which can count various GPU-internal signals. On Nvidia GPUs, these signals are grouped into domains. Each of these domains consists of up to 256 signal lines, which can originate anywhere on the GPU. Each domain has a distinct set of signal lines, which implies that each signal can be counted by only one domain. Therefore, it is often impossible to count multiple related events simultaneously – for example, read and write requests to memory cannot be counted at the same time.

To accurately count such events, it is therefore necessary to repeat the execution of GPU kernels once for each event to be counted. However, it is possible to combine multiple events into one – for example, the sum of read and write requests can be counted as one event if separate counts for the two are not required. Note that we observed this behavior in Nvidia's profiling tools for GPUs up to the Kepler generation, which indicates that there is indeed no better way.

2.4 GPUswap

GPUswap [21] is part of the GPU driver and operates by intercepting memory allocation requests from applications and migrating application data between GPU memory and system RAM in response to memory shortage. After each allocation request, GPUswap divides the buffer returned by the request into fixed-size chunks. Each chunk is a virtually-contiguous portion of application address space that can be swapped independently of other chunks. By default, each chunk is 2 MiB in size.

When there is insufficient GPU memory to fulfill a given allocation request, GPUswap's default policy evicts chunks from applications using more than their fair share of GPU memory to system RAM. This eviction is done in two steps: In the first step, GPUswap selects a set of chunks at random from the applications currently holding most GPU memory. Once a sufficient number of chunks has been selected to make room for the outstanding allocation request, GPUswap transparently migrates these chunks to system RAM. Since each application's GPU access must be suspended before GPUswap can migrate memory from that application's address space, GPUswap performs this migration one application at a time to prevent the GPU from falling idle.

While selecting chunks from the applications owning most GPU memory effectively maintains fairness, selecting chunks from the applications' GPU address spaces at random is suboptimal. Ideally, GPUswap should swap rarely-accessed chunks first. However, GPUswap has no information about which chunks are accessed frequently. In GPUswap's evaluation, this problem manifests not only as high overhead, but also as a rather large variance in that overhead, depending on the set of chunks selected for swapping. This variance indicates that a swapping policy can in fact improve the swapping overhead.

3. MEMORY ACCESS PATTERNS OF GPU APPLICATIONS

As a first step towards developing a swapping policy for GPUs, we measure the memory access patterns of various GPU applications. To that end, we need to count accesses to individual pages with sufficient accuracy. However, as we explain in Section 2, current GPUs do not support counting accesses to individual pages out of the box.

3.1 Approach

To allow the GPU's performance counters to count accesses to individual pages, we move these pages to system RAM one at a time, as shown in Figure 1. Assuming the application does not otherwise use system RAM, the number of system RAM accesses – which can be counted efficiently using the GPU's performance counters – then equals the number of accesses to the isolated page.

The major disadvantage of this approach is that accesses can be counted for only one page at a time. We therefore run each GPU kernel of the application being observed twice for each allocated page in its address space – once for read and once for write accesses – with a different page in system RAM for each pair of runs. In addition, we restore the input data of each kernel between runs to ensure each run behaves the same way. Note that if multiple pages were in system RAM at once, the combined number of accesses to these pages could still be counted, but there would be no way to distinguish which page accumulated how many accesses.

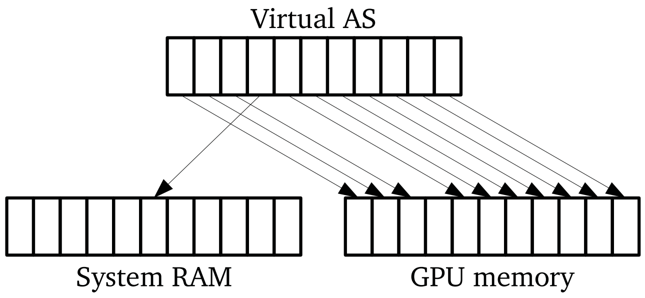


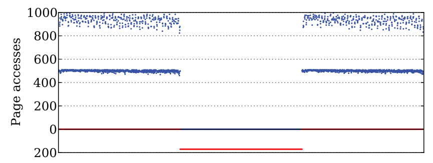
Figure 1: Isolating a single page from a GPU address space by moving that page to system RAM

Our approach assumes three key conditions to be met. First, the location of application buffers in virtual memory must not change between runs. This condition is guaranteed by our design: Since we do not allocate or free any application buffers between runs, but only restore the contents of these buffers, it is impossible for a buffer's location to change. Second, the number of accesses to each page must be deterministic between runs. We argue that this requirement is fulfilled since we restore each GPU kernel's input data before each run. Therefore, each kernel performs the same work on the same input data during each run. Third, we assume that profiling is performed on the same GPU model that is later used in production, as a different GPU model may, for example, have a different cache size and thus produce different access counts. Note that we do not require the GPU kernels' execution to be timing independent since we only record the total number of memory accesses during the execution of each kernel, but not the exact timing of these accesses. We do not consider the exact timing of memory accesses to be relevant in the context of this paper since we can only make swapping decisions at kernel boundaries.

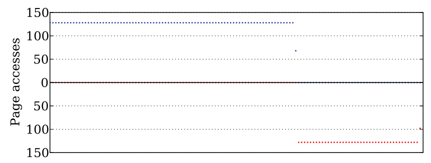
3.2 Implementation

To implement our approach, we added an API to GPUswap that allows userspace applications to specify which page should be in system RAM. GPUswap then moves that page to system RAM and all other pages to GPU memory, ensuring that only one page is in system RAM at a time. Furthermore, the API also includes functions allowing the application to configure and read the GPUs performance counters.

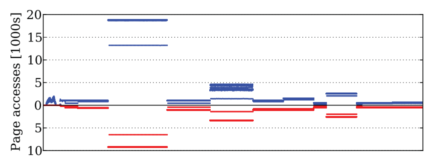
Using the new API, we traced the memory accesses of various GPU applications. Since GPUswap only supports the CUDA driver API, we used a subset of the Rodinia benchmark suite [5] which has previously been ported to that API [18] as our target set of applications. We instrumented each application by adding a loop around each application's kernel launches, which iterates over all allocated pages in the application's address space. In each loop iteration, the application executes each of its GPU kernels twice – once



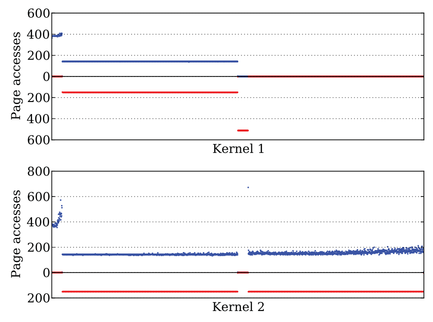
(a) hotspot



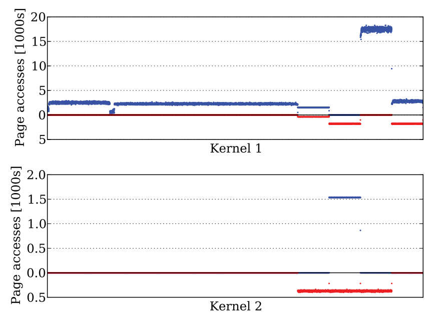
(b) nn



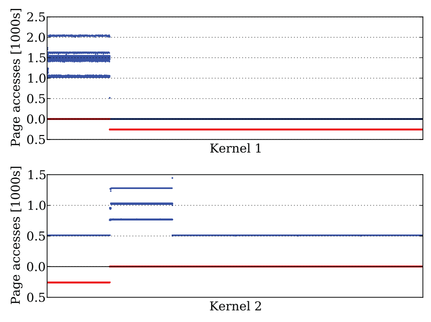
(c) heartwall



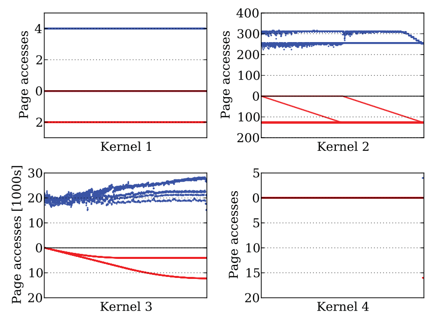
(d) backprop



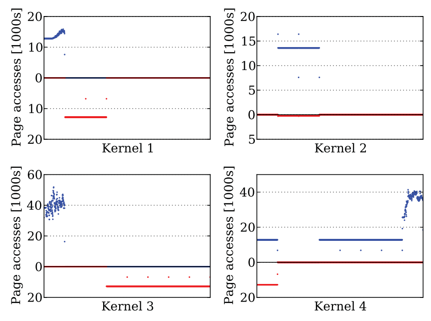
(e) bfs



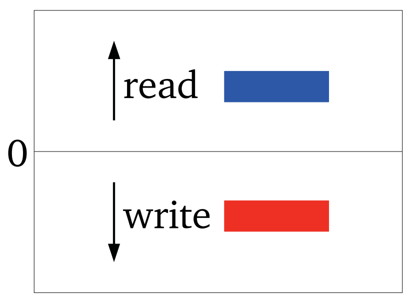
(f) srad_v2



(g) lud



(h) srad_v1



(i) legend

Figure 2: Memory access patterns of various GPU applications. Each graph shows the accesses made by one of the application's GPU kernels. The X-axis is the virtual address space, with virtual addresses increasing from left to right. The Y-axis shows the number of accesses to each allocated page; values above zero show the number of read accesses, while values below zero correspond to the number of write accesses. For better readability, we omitted unallocated regions as well as the 60 MiB stack buffer, which is present for all applications.

for read and once for write accesses. After each finished kernel, the application reads the number of system RAM accesses from the GPU's performance counters, and resets the counters for the next kernel execution.

3.3 Results

Our results are shown in Figure 2. For most applications, the various GPU buffers allocated by the application are clearly visible as steep vertical drops. *Backprop*, for example, allocates four buffers: Two small ones, which are visible at the far left and at the center of the plot, and two larger ones. The same effect is visible in the traces of the other applications, the only exception being *lud* which allocates only one buffer on the GPU. Our observations therefore indicate that the number of accesses varies mostly by buffer rather than by page.

While the access count of a given page depends mostly on the buffer the page belongs to, there is also some variance in the access counts of different pages within the same buffer. In Figure 2, this variance manifests either as multiple parallel lines at the same horizontal position as in *heartwall*, *hotspot* and *srad_v2*, or as slopes as for the leftmost buffer of backprop and *srad_v1*. However, the variance between different buffers is typically much larger than the variance within each buffer. For example, *hotspot*'s first and last buffers display a large variance in the number of reads per page; in any case, however, both of these buffers show a much larger number of accesses than the buffer in the center of the plot.

In addition to the buffers shown in Figure 2, all applications allocate another buffer of 60 MiB, which we excluded from the figure for legibility. This buffer is allocated by the CUDA runtime and used as stack space for the application's GPU threads. Since GPU threads typically keep most of their data in registers rather than on the stack, this buffer shows only a small number of accesses. Beside the stack buffer, the runtime allocates several other buffers – e.g., for application code – which are included in Figure 2, but are too small to be clearly visible. Note that even though these buffers are not under direct application control, their pages can be swapped to system RAM just like pages from any other buffer.

To verify that the results of our profiling are representative, we repeated this experiment multiple times with varying input sizes. While the total number of memory accesses made by each kernel did change with the input size, the number of accesses per buffer relative to other buffers stayed the same for all applications – for example, the same buffer always

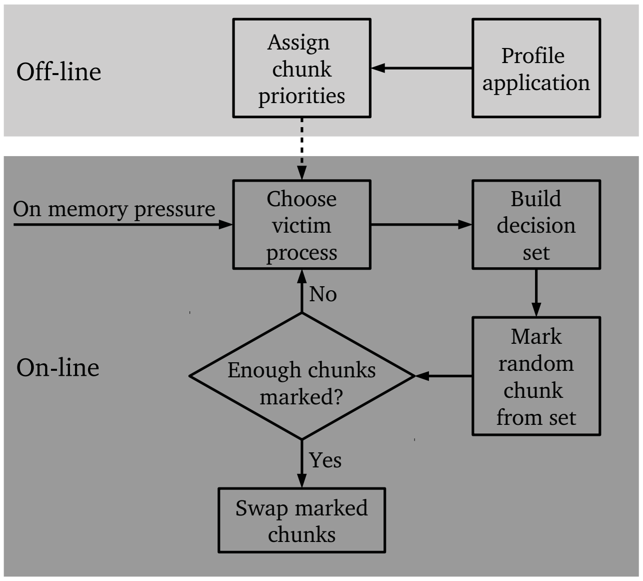


Figure 3: Operation of GPrioSwap

received the largest number of memory accesses, regardless of input size. However, we did observe some changes in the access counts of individual pages relative to other pages in the same buffer. We therefore consider the results of our profiling to be representative on the buffer-, but not on the page level for these applications.

While profiling worked well for the applications we examined, we do assume that some workloads cannot be profiled representatively using our method. When executing SQL queries on a GPU [25], for example, the query being executed does not only determine the input data, but also the exact processing that needs to be done. One would thus need information about what queries to expect in production to obtain representative profiling results. We therefore expect our method to yield representative results only for applications performing the same task – though possibly on a different input – each time they are run.

4. A SWAPPING POLICY FOR GPUS

In this section, we describe GPrioSwap, a practical swapping policy for GPUs based on the insights gained from our traces presented in Section 3. Since making swapping decisions at runtime is not viable on current GPUs, GPrioSwap takes a self-paging approach similar to that found in past works like Exokernel [7], Nemesis [15] and SPIN [4]. The key difference between these works and GPrioSwap is that the application is never explicitly asked to return memory to the operating system. Instead, applications only set the policy for a swapping mechanism which otherwise operates transparently.

The basic operation of GPrioSwap is shown in Figure 3. GPrioSwap operates in two steps: An off-line step, in which priorities are assigned to an application's buffers, and an on-line step, which uses these priorities to find suitable chunks of memory to swap in response to memory pressure.

In the off-line step, which we describe in Section 4.1, the application developer defines a priority for each buffer. To that end, the developer first profiles the application's memory accesses to each buffer, and assigns priority values to buffers such that rarely-accessed buffers will be swapped first. Then, the developer modifies the application to pass these priority values to GPrioSwap as part of the application's memory allocation requests.

In the on-line step, presented in Section 4.2, GPrioSwap uses the priorities defined in the off-line step to find suitable candidates for swapping if memory pressure occurs. Whenever an allocation request cannot be satisfied due to insufficient GPU memory, GPrioSwap first chooses a set of low-priority chunks from applications using more than their fair share of GPU memory. These chunks are then moved to system RAM using GPUSwap's original swapping mechanism. Conversely, when an application frees memory, GPrioSwap moves high-priority chunks back to GPU memory.

4.1 Priority Assignment

The first step in the operation of GPUPrioSwap is to assign a priority value to each application buffer. To that end, the developer profiles the application to determine the average number of accesses per chunk for each buffer. To speed up this profiling process, we modified our profiling method from Section 3.1 to swap entire buffers instead of individual pages. In addition, since we currently weigh read and write accesses equally, we can count both kinds of access in one pass. The application's kernels must thus be repeated once per buffer rather than twice per page. The immediate result of profiling is the total number of accesses per buffer, which we then divide by the number of chunks in the buffer to obtain the average number of accesses per chunk.

Once that information is known, the developer can convert these numbers into a priority value for each buffer by ordering the buffers in descending order of their average chunk access counts: The buffer with most accesses per chunk receives the highest priority value, the buffer with the second most accesses per chunk receives the second highest priority value and so on. Note that there is no need to coordinate priority values across applications. GPrioSwap uses the buffers' priorities only to select which chunks from an application's address space to swap, but not to select the application itself.

Once a priority has been determined for each buffer, the application passes each buffer's priority to GPrioSwap as part of the allocation request for the buffer. To avoid breaking compatibility with existing applications, GPrioSwap assumes a default priority if the application does not explicitly pass a priority value for a buffer. If users cannot – or choose not to – profile and modify their applications, these applications can therefore continue to run unmodified, although without the full benefit of GPrioSwap.

In addition to buffers without explicit priorities, GPrioSwap must also handle buffers allocated by the GPU runtime, such as application code or stacks. The runtime allocates these buffers through the driver's regular allocation mechanism. From the driver's point of view, these buffers are thus not fundamentally different from application buffers; specifically, they can be swapped just like any other buffer. However, since these buffers are not under application control, it is impossible for the developer to assign a priority value for these buffers. Simply assigning the default priority to all runtime buffers would be suboptimal since our application traces have shown that some of these buffers are better candidates for swapping than others. We therefore modify the GPU runtime to assign a different default priority to each of these

```
input  : Chunk list of a victim process
output : Decision set
decision set ← empty array
minPrio ← largest allowed priority value
foreach chunk c in victim's chunk list do
    if priority(c) == minPrio then
        Append c to decision set
    else if priority(c) < minPrio then
        Empty decision set
        Append c to decision set
        minPrio ← priority(c)
    end
end
return decision set
```

Listing 1: Algorithm for computing the decision set from the victim's chunk list

runtime buffers. Note that the access counts for these buffers were highly consistent for our benchmark applications, but that may not be the case for other applications.

4.2 Chunk Selection

Once all buffers have been assigned a priority, GPrioSwap uses these priorities to select chunks for swapping. When the GPU driver receives a request for GPU memory that cannot be satisfied, GPrioSwap first selects a victim which must give up some of its memory. As in the original implementation of GPUswap, that victim is the application owning most GPU memory to ensure fairness. Next, GPrioSwap creates a list of the lowest-priority chunks from the victim's address space which we call the decision set. The algorithm for creating this decision set is shown in Listing 1. Finally, GPrioSwap chooses a chunk from the decision set at random and marks that chunk for swapping. The entire process then repeats until swapping all marked chunks frees up enough GPU memory to service the allocation request which triggered the swap operation. Finally, the marked chunks are moved to system RAM one application at a time using GPUswap's original swapping mechanism.

Our algorithm re-builds the entire decision set once for each swapped chunk. We chose this approach for simplicity since re-building the entire set implicitly handles two corner cases: i) the next chunk to be swapped may have a different priority, or ii) the next chunk to be swapped may belong to a different victim. In addition, we assume the cost for re-building the set to be small compared to the cost of the DMA transfer moving the marked chunks to system RAM.

To maintain a high utilization of GPU memory, GPrioSwap returns swapped chunks to the GPU whenever an application frees chunks in GPU memory. The process of selecting chunks to return to the GPU is similar to that for finding swapping candidates: GPrioSwap first chooses the application consuming the least amount of GPU memory as the winner, and then builds a decision set consisting of the highest-priority chunks from the winner's address space using an algorithm similar to that in Listing 1. GPrioSwap then picks a chunk from the decision set at random and marks it for returning to GPU memory. This process continues until the marked chunks consume all available GPU memory, at which point the marked chunks are migrated to the GPU one application at a time.

5. EVALUATION

To verify that GPrioSwap reduces the overhead induced by GPU data in system RAM, we conducted experiments using our prototype implementation. In this section, we present the results of these experiments.

5.1 Experimental Setup

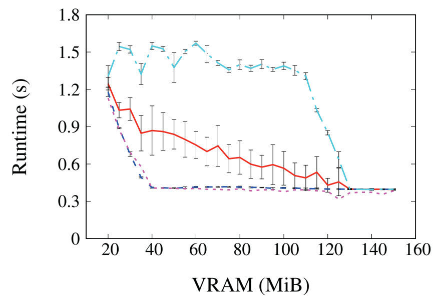
We conducted all experiments described in this section on an Nvidia GeForce GTX 480, which is based on the Fermi microarchitecture and features 480 cores and 1.5 GiB of GDDR5 memory. Our test system further includes an Intel Core i7-4470 clocked at 3.4 GHz and 16 GiB of system RAM. In all our experiments, we locked both CPU and GPU to the highest possible clock frequency.

Our benchmark system runs Ubuntu 12.04.5, using the default kernel version 3.5. GPUswap is built into the pscnv driver [24], which is the only open-source driver for Nvidia GPUs supporting memory-mapped command submission channels. We used the Gdev userspace libraries [20], which to our knowledge include the only implementation of CUDA supporting pscnv. The applications used in our benchmarks were originally taken from the Rodinia benchmark suite [5], but have been modified to work with the CUDA driver API implemented by Gdev [18]. In all experiments, we used a chunk size of 2 MiB, which has been shown to be the optimal size for GPUswap [21].

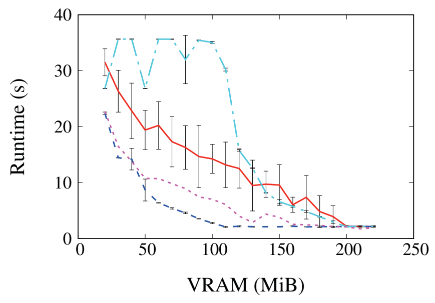
5.2 Swapping Overhead

In our first experiment, we measure the effect of our policy on the overhead associated with swapping GPU data to system RAM. To that end, we compare the average execution time of several benchmark applications under GPUswap to the average and best-case execution times under random selection for different amounts of available GPU memory. Initially, we granted each application a sufficient amount of GPU memory so that no swapping occurs, and then gradually reduced the amount of available GPU memory until only 20 MiB remained. To decrease the available GPU memory, we used the same limiting mechanism as in the evaluation of GPUswap, which causes the GPU driver to ignore all GPU memory above a configurable threshold. To ensure that applications actually compete for memory, we ran two instances of each application concurrently. We ran each application 10 times at each memory size we tested. The numbers reported are the average runtime of both instances over all 10 runs. To obtain an approximation of the best-case runtime, we ran all applications another 100 times under GPUswap's original implementation, and recorded the shortest execution time observed during these 100 runs.

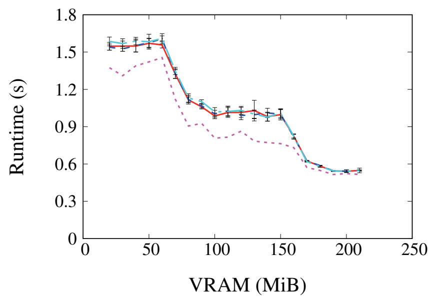
The results are shown in Figure 4. Using random selection, the application runtime starts to increase immediately when the amount of available GPU memory becomes too small to hold both application instances since some important data is always selected for swapping. In addition, random selection causes a high jitter in the applications' runtimes depending on whether a good or bad set of chunks is randomly selected for swapping. In contrast, GPrioSwap swaps the relatively unimportant stack buffer first and therefore initially has no effect on the runtimes of all applications except heartwall (Figure 4c) and srad_v1 (Figure 4g), which appear to be limited by something other than the speed of the memory used. When large amounts of memory are swapped, GPrioSwap eventually has no choice but to select important data for



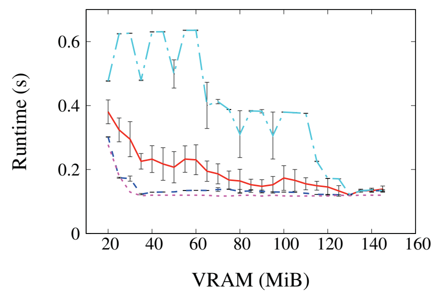
(a) backprop



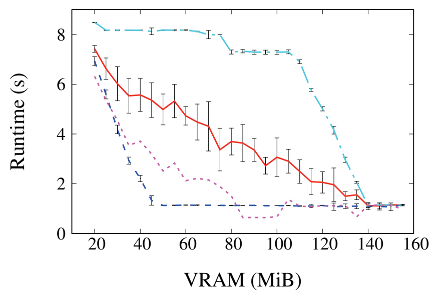
(b) bfs



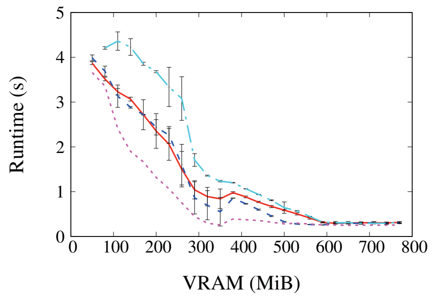
(c) heartwall



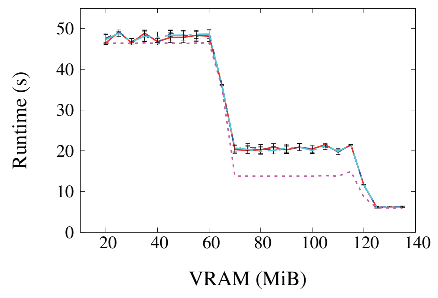
(d) hotspot



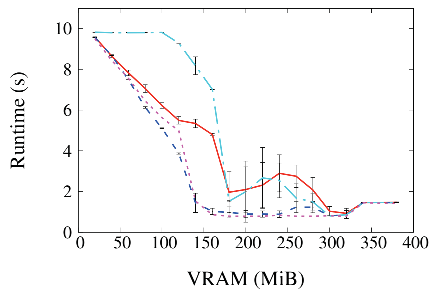
(e) lud



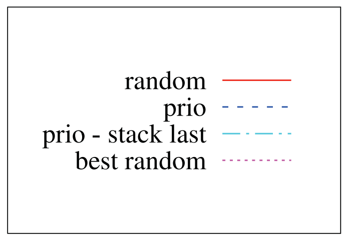
(f) nn



(g) srad_v1



(h) srad_v2



(i) legend

Figure 4: Average runtime over 10 runs of various GPU applications depending on the amount of GPU memory available. "Random" shows the runtime when chunks for swapping are selected randomly, while "prio" denotes the runtime when chunks are chosen by GPrioSwap. "Prio – stack last" shows the runtime under GPrioSwap with the stack buffer's priority set to the highest value, which mimics the behavior of a policy ignoring buffers allocated by the GPU runtime. For comparison, "best random" indicates the best application runtime seen in 100 runs using random chunk selection. The error bars indicate the standard deviation.

swapping, prompting a steep increase in the applications' runtimes as the amount of GPU memory available decreases further. However, these increased execution times are close to, and in some cases even below the best-case execution time with random selection. At 20 MiB of available memory, finally, virtually all application data is in system RAM, leaving both policies with little choice about what to swap. As a result, the application runtimes at 20 MiB of GPU memory are virtually identical for both policies. In all cases, however, GPrioSwap does not increase the runtime of any application compared to random selection. Overall, we therefore conclude that GPrioSwap improves the swapping overhead compared to random selection.

To verify that swapping the stack buffer is indeed the reason why swapping initially has no effect on the runtime of most applications, we repeated the previous experiment with the stack buffer's priority set to the highest value, ensuring that this buffer is swapped last. Note that the remaining system buffers, such as code segments, have a high priority already since we found these buffers to be important during profiling. In this experiment, we therefore expect GPrioSwap to mimic the behavior of a solution that ignores system buffers until only system buffers remain in GPU memory. The results, labeled "prio – stack last" in Figure 4, show that the application runtime increases even more steeply than using random selection as soon as GPU data is swapped. This result is expected since random selection likely chooses at least some chunks from the stack buffer for swapping.

It is noteworthy that there are cases where the best-case execution time using random selection is below the execution time using GPrioSwap, and in some cases even below the execution time with all application data in GPU memory. This result indicates that the swapping overhead can be further reduced if hot individual pages inside buffers can be reliably

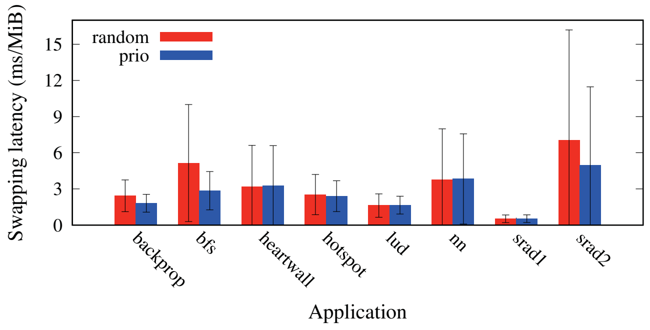


Figure 5: Average swapping latency per megabyte of data swapped for various applications.

detected. Recent research suggests that the performance of GPU applications can improve if some application data is placed in system RAM [1], and it is possible that we are observing a similar effect.

5.3 Policy Overhead

In a second experiment, we measured the latency induced by GPrioSwap in cases where there is not enough GPU memory readily available. This latency includes selecting chunks for swapping, suspending of victim applications and the transfer of selected chunks to system RAM, but not the time spent in the driver's original memory allocator. In addition to the time, we also recorded the amount of memory that GPUswap had to migrate to satisfy each request. Since the delay experienced by the allocating application naturally increases with the amount of swapped memory, we then calculated the latency induced by GPrioSwap per megabyte of data swapped.

Figure 5 shows the delay induced per megabyte of swapped data for GPrioSwap as well as the original implementation of GPUswap. The results indicate that GPrioSwap does not cause any significant delays beyond those induced by the original implementation of GPUswap. In fact, the delays induced by GPrioSwap appear to be shorter than those of the original implementation for some applications, though not by a significant amount. This result is consistent with our expectations: Since the delay induced by GPUswap is generally dominated by the DMA transfers needed for migrating data to system RAM [21], the delay induced by any policy tends to be negligible.

6. RELATED WORK

The problem of sharing a GPU between multiple applications or virtual machines has been addressed by several research projects. GViM [13], gVirtuS [9], rCUDA [6], VOCL [32], vCUDA [27] and VGVM [30] employ an approach called *API remoting*: These projects intercept GPU commands in a modified GPU runtime library and forward these commands to the hypervisor. As a consequence, these projects are generally limited to applications using that specific library. TimeGraph [19] and PTask [26] take this approach a step further, defining new, more efficient APIs at the operating system level. However, either the user space libraries or the applications themselves must be modified to support these new APIs. With GPUvm [28] and gVirt [29], the hypervisor instead exposes a virtual GPU device that is identical to the physical GPU, allowing the VM to use the GPU's original device driver without modification. VGRIS [33] and LoGV [12] take a paravirtual approach, intercepting commands in a guest device driver which exposes the same interface as the native GPU driver towards user space, allowing applications and user space libraries to remain unmodified. While all of these projects are able to share a GPU between multiple applications and/or virtual machines, none of them support oversubscription of GPU memory.

Gdev [20] implements a kernel-level command scheduler for GPUs: Application commands are queued within Gdev, with a software scheduler deciding when to forward these commands to the GPU. This software scheduling allows Gdev to share GPU buffers between applications: Whenever the scheduler selects an application to execute on the GPU, Gdev copies the contents of any shared buffers in that application's address space to the GPU before the application's GPU kernel begins execution; any data already present in that buffer is copied to system RAM. GDM [31] later generalized Gdev's approach by making all GPU buffers implicitly shared: When a GPU kernel is selected for execution, any data that kernel might touch is copied to the GPU before the kernel is executed; data from other applications may be evicted to system RAM in the process. While both Gdev and GDM are able to extend GPU memory, both rely on software scheduling, which defeats the GPU's own, highly efficient scheduling and context switching and thus induces considerable application overhead even in the absence of memory pressure. In addition, both Gdev and GDM may copy data that the application does not actually need, while with GPUswap, only data that is actually accessed is transferred over the PCIe-Bus.

RSVM [17] places GPU memory management in a user-space runtime library that implements swapping through explicit API calls: Applications must map GPU buffers into their address space explicitly, and unmap these buffers after use. On each map call, the runtime copies the mapped data to the GPU and swaps unmapped buffers if necessary. Swapping is thus cooperative – if applications do not actively unmap their GPU buffers, these buffers will stay on the GPU indefinitely. The work of Becchi et al. [3] instead ties the swap operation to the kernel launch: When the application launches a GPU kernel, the runtime copies all data given to that kernel as parameters to the GPU, potentially evicting other data from the same application's address space in the process. Swapping data from other applications is also possible, but only in a cooperative fashion: On memory shortage, applications receive requests to swap data, but may choose not to obey that request. In contrast, GPrioSwap is completely transparent to applications and can thus swap data even against the application's will.

Agarwal et al. [1] propose another mechanism for extending GPU memory with system RAM. Their allocation policy is similar to GPrioSwap in that it based on per-buffer hints generated from application profiles. However, their work is different from ours in three ways: First, their method of profiling is based on compiler modification, and therefore only considers application buffers, but not buffers allocated by the GPU runtime. In addition, relying on compiler modification restricts their profiling to CUDA applications, while our performance counter-based approach can be applied to any type of application. Second, the authors do not target a

GPU shared by multiple applications, but instead focus on HPC applications with a working set larger than the available GPU memory. Third, the authors target future machines in which the CPU and GPU form a CC-NUMA system without the bandwidth constraints of the PCIe-Bus. In contrast, GPrioSwap assumes that the GPU is connected via PCIe, resulting in a much larger difference in bandwidth between GPU memory and system RAM.

Finally, current CUDA SDKs [23] include a function called Unified Memory, which enables automatic data transfers between CPU and GPU. Unified memory creates a shared address space between CPU and GPU. When data in unified memory is accessed, the runtime transparently copies that data to the location of the access (CPU or GPU). More recently, Heterogeneous Memory Management (HMM), which offers a similar functionality in a device-agnostic way, has been proposed for inclusion into the Linux kernel [11]. However, to the best of our knowledge, neither Unified Memory nor HMM enable oversubscription of GPU memory since both reserve memory for all data that might be on the GPU.

7. CONCLUSION

In this paper, we have presented GPrioSwap, a practical swapping policy for GPUs. First, we have analyzed the behavior of various GPU applications, and determined that the importance of pages in the address spaces of those applications varies mostly by buffer rather than by individual page. Following that insight, GPrioSwap allows developers to determine the relative importance of their applications' buffers through profiling, and to assign a priority to each buffer based on the buffer's importance. In the event of memory pressure, GPrioSwap then selects buffers for swapping based on these priorities: Buffers critical to application performance stay in GPU memory, while less important buffers are moved to system RAM. Our experiments with our prototype have shown that GPrioSwap significantly reduces the overhead associated with using system RAM in place of GPU memory compared to the original implementation of GPUswap.

There are currently two main drawbacks to GPrioSwap. First, our current prototype only assigns a single priority value to each buffer, which is then valid for the entire lifetime of the buffer. However, different GPU kernels of the same application can sometimes display different memory access patterns. In the future, we plan to investigate ways to assign separate buffer priorities for each kernel. The second drawback of our policy is the required profiling of GPU applications, which is a complicated and time-consuming process. GPU vendors could alleviate this requirement by adding appropriate hardware support, such as reference bits in the GPU's page tables. We believe that such hardware support has a significant potential to improve memory management on the GPU, and hope that our work will inspire GPU vendors to include this functionality in future generations of their hardware.

8. REFERENCES

[1] Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., and Keckler, S. W. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2015), ASPLOS '15, pp. 607–618.

[2] Agrawal, S. R., Pistol, V., Pang, J., Tran, J., Tarjan, D., and Lebeck, A. R. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2014), ASPLOS '14, pp. 19–34.

[3] Becchi, M., Sajjapongse, K., Graves, I., Procter, A., Ravi, V., and Chakradhar, S. A Virtual Memory Based Runtime to Support Multi-tenancy in Clusters with GPUs. In *Proceedings of the ACM International Symposium on High-Performance Parallel and Distributed Computing* (June 2012), HPDC '12, pp. 97–108.

[4] Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Dec. 1995), SOSP '95, pp. 267–283.

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J., Lee, S.-H., and Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization* (Oct. 2009), IISWC '09, pp. 44–54.

[6] Duato, J., Peña, A. J., Silla, F., Mayo, R., and Quintana-Ortí, E. S. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the IEEE International Conference on High Performance Computing Simulation* (June 2010), HPCS '10, pp. 224–231.

[7] Engler, D. R., Kaashoek, M. F., and O'Toole, J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Dec. 1995), SOSP '95, pp. 251–266.

[8] Fung, J., and Mann, S. Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. In *Proceedings of the IEEE International Conference on Multimedia and Expo* (June 2008), ICME '08, pp. 9–12.

[9] Giunta, G., Montella, R., Agrillo, G., and Coviello, G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proceedings of the Springer European Conference on Parallel Processing* (Sept. 2010), Euro-Par '10, pp. 379–391.

[10] Glisse, J. Using process address space on the GPU, Sept. 2013. https://www.x.org/wiki/Events/XDC2013/XDC2013JeromeGlisseUsingProcessAddressSpaceGPU/xdc2013-glisse.pdf.

[11] Glisse, J. LKML: HMM (Heterogeneous Memory Management) v18, Mar. 2017. https://lkml.org/lkml/2017/3/16/596.

[12] Gottschlag, M., Hillenbrand, M., Kehne, J., Stoess, J., and Bellosa, F. LoGV: Low-Overhead GPGPU Virtualization. In *Proceedings of the IEEE International Workshop on Frontiers of Heterogeneous Computing* (Nov. 2013), FHC '13, pp. 1721–1726.

[13] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V., and Ranganathan, P. GViM: GPU-accelerated Virtual Machines. In *Proceedings of the ACM Workshop on System-Level Virtualization for High Performance Computing* (Mar. 2009), HPCVirt '09, pp. 17–24.

[14] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM Conference* (Aug. 2010), SIGCOMM '10, pp. 195–206.

[15] Hand, S. Self-Paging in the Nemesis Operating System. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Feb. 1999), OSDI '99, pp. 73–86.

[16] Jang, K., Han, S., Han, S., Moon, S. B., and Park, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation* (Mar. 2011), NSDI '11.

[17] Ji, F., Lin, H., and Ma, X. RSVM: A Region-based Software Virtual Memory for GPU. In *Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques* (Sept. 2013), PACT '13, pp. 269–278.

[18] Kato, S. Gdev benchmarks. https://github.com/shinpei0208/gdev-bench.

[19] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the USENIX Annual Technical Conference* (June 2011), USENIX ATC '11, pp. 17–30.

[20] Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. A. Gdev: First-Class GPU Resource Management in the Operating System. In *Proceedings of the USENIX Annual Technical Conference* (June 2012), USENIX ATC '12, pp. 401–412.

[21] Kehne, J., Metter, J., and Bellosa, F. GPUswap: Enabling Oversubscription of GPU Memory Through Transparent Swapping. In *Proceedings of the ACM International Conference on Virtual Execution Environments* (Mar. 2015), VEE '15, pp. 65–77.

[22] Menychtas, K., Shen, K., and Scott, M. L. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (June 2014), ASPLOS '14, pp. 301–316.

[23] Nvidia Corporation. CUDA Toolkit, 2013-07-02T10:16:48-07:00. https://developer.nvidia.com/cuda-toolkit.

[24] PathScale, Inc. Pscnv. https://github.com/pathscale/pscnv.

[25] Rauhe, H., Dees, J., Sattler, K.-U., and Faerber, F. Multi-level Parallel Query Execution Framework for CPU and GPU. In *Proceedings of the Springer East European Conference on Advances in Databases and Information Systems* (Sept. 2013), ADBIS '13, pp. 330–343.

[26] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the ACM Symposium on Operating Systems Principles* (Sept. 2011), SOSP '11, pp. 233–248.

[27] Shi, L., Chen, H., Sun, J., and Li, K. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers 61*, 6 (June 2012), 804–816.

[28] Suzuki, Y., Kato, S., Yamada, H., and Kono, K. GPUvm: Why not virtualizing GPUs at the hypervisor? In *Proceedings of the USENIX Annual Technical Conference* (June 2014), USENIX ATC '14, pp. 109–120.

[29] Tian, K., Dong, Y., and Cowperthwaite, D. A full GPU virtualization solution with mediated pass-through. In *Proceedings of the USENIX Annual Technical Conference* (June 2014), USENIX ATC '14, pp. 121–132.

[30] Vasilas, D., Gerangelos, S., and Koziris, N. VGVM: Efficient GPU capabilities in virtual machines. In *Proceedings of the IEEE International Conference on High Performance Computing & Simulation* (July 2016), HPCS '16, pp. 637–644.

[31] Wang, K., Ding, X., Lee, R., Kato, S., and Zhang, X. GDM: Device Memory Management for GPGPU Computing. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems* (June 2014), SIGMETRICS '14, pp. 533–545.

[32] Xiao, S., Balaji, P., Zhu, Q., Thakur, R., Coghlan, S., Lin, H., Wen, G., Hong, J., and c Feng, W. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *Proceedings of the IEEE Conference on Innovative Parallel Computing* (May 2012), InPar '12, pp. 1–12.

[33] Yu, M., Zhang, C., Qi, Z., Yao, J., Wang, Y., and Guan, H. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proceedings of the ACM International Symposium on High-Performance Parallel and Distributed Computing* (June 2013), HPDC '13, pp. 203–214.

[34] Zheng, T., Nellans, D., Zulfiqar, A., Stephenson, M., and Keckler, S. W. Towards high performance paged memory for GPUs. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture* (Mar. 2016), HPCA '16, pp. 345–357.