GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping

Jens Kehne Jonathan Metter Frank Bellosa
Operating Systems Group, Karlsruhe Institute of Technology (KIT)
os@itec.kit.edu

Abstract

Over the last few years, GPUs have been finding their way into cloud computing platforms, allowing users to benefit from the performance of GPUs at low cost. However, a large portion of the cloud's cost advantage traditionally stems from oversubscription: Cloud providers rent out more resources to their customers than are actually available, expecting that the customers will not actually use all of the promised resources. For GPU memory, this oversubscription is difficult due to the lack of support for demand paging in current GPUs. Therefore, recent approaches to enabling oversubscription of GPU memory resort to software scheduling of GPU kernels – which has been shown to induce significant runtime overhead in applications even if sufficient GPU memory is available – to ensure that data is present on the GPU when referenced.

In this paper, we present GPUswap, a novel approach to enabling oversubscription of GPU memory that does not rely on software scheduling of GPU kernels. GPUswap uses the GPU's ability to access system RAM directly to extend the GPU's own memory. To that end, GPUswap transparently relocates data from the GPU to system RAM in response to memory pressure. GPUswap ensures that all data is permanently accessible to the GPU and thus allows applications to submit commands to the GPU directly at any time, without the need for software scheduling. Experiments with our prototype implementation show that GPU applications can still execute even with only 20 MB of GPU memory available. In addition, while software scheduling suffers from permanent overhead even with sufficient GPU memory available, our approach executes GPU applications with native performance.

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

Keywords Virtualization; Memory Overcommitment; Oversubscription; Swapping; GPU

1. Introduction

Over the last few years, the use of GPUs as compute accelerators has been constantly growing. Especially in the field of high performance computing (HPC), GPUs are delivering unprecedented levels of performance for certain classes of applications. Most recently, GPUs have also been finding their way into cloud computing platforms, allowing users to benefit from the performance of GPUs at low cost by relieving them of the burden of purchasing and maintaining a dedicated supercomputer.

Cloud providers often oversubscribe the resources of their cloud platforms in order to reduce costs: In case of GPUs, the provider can rent out more GPU memory to customers than is actually available, expecting that the customers will not actually use all of the promised memory. In doing so, cloud providers carefully assign virtual machines to physical hosts such that the actual demand for GPU memory will be close to – but not more than – the physical capacity of the GPU in order to maximize utilization and application performance at the same time. However, customers do not always behave as expected and may choose to fully utilize the promised memory at any time, possibly exceeding the GPU's physical capacity. Though this kind of over-utilization may rarely occur in practice, the cloud platform must ensure that all applications still function correctly and with acceptable performance even if memory is over-utilized.

Currently, handling over-utilization of GPU memory is difficult since current GPUs do not support precise exceptions and therefore cannot seamlessly continue execution after page faults. As a result, these GPUs typically treat page faults as fatal errors. Previous attempts to handle over-utilization of GPU memory, such as Gdev [11] and GDM [18] therefore rely on software scheduling of GPU kernels in order to multiplex the available GPU memory. When dispatching a kernel to the GPU, these solutions copy

the working set of the application that launched the kernel into GPU memory, evicting memory from other kernels if necessary. On the downside, these systems typically software-schedule the GPU even while GPU memory is not fully utilized. Since GPU software scheduling has been shown to induce considerable overhead [11], such software scheduling should be avoided.

In this paper, we present GPUswap, a novel approach to extending GPU memory, which uses the GPU's ability to directly access system RAM to transparently extend applications' GPU address spaces. GPUswap relocates data from applications over-using their memory quota to system RAM, and then redirects the page table entries for the relocated data to the copy in system RAM. Our approach keeps all data permanently accessible to the GPU and does therefore not require software scheduling of GPU kernels. In contrast to previous approaches, our approach does not induce overhead unless the GPU's memory is actually over-utilized. Experiments with our prototype implementation show that when sufficient GPU memory is available, software scheduling suffers from 3.5 – 30 % overhead, while GPUswap executes GPU applications with native performance.

The rest of this paper is organized as follows: We first provide a short overview of current GPUs in Section 2 and introduce our design goals in Section 3. Then, we present our proposed design in Section 4. Section 5 describes the prototypical implementation of our approach, while Section 6 presents our initial performance evaluation of our prototype. Finally, Section 7 presents related work and Section 8 concludes the paper.

2. GPU Background

Modern GPUs are asynchronous in nature: Applications submit small tasks called *GPU kernels* to the GPU and are then free to perform other work while the GPU processes these kernels. Applications can submit work to the GPU by writing commands into *command submission channels*. Recent GPU generations from both AMD and Nvidia feature multiple such command submission channels that can be used by different applications concurrently. To guarantee protection between those applications, these GPUs confine every application to its own *address space*, thus ensuring that an application cannot access memory of other applications.

In this section, we give an overview of the relevant mechanisms for CPU-GPU interaction and protection. We start with a description of GPU memory management, including address spaces, in Section 2.1, followed by a description of the GPU command submission process in Section 2.2.

2.1 Memory management

Modern GPUs assign an address space to each application in order to guarantee protection between GPU kernels from different applications. Each address space is defined by a page table containing GPU-virtual to GPU-physical mappings.

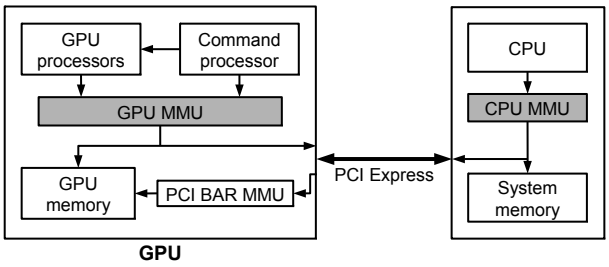


Figure 1: Access paths to GPU memory. GPU kernels access both system RAM and GPU memory through the GPU's MMU. (© 2013 IEEE. Taken from [5] with permission)

For the GPUs we examined, these page tables can contain two types of memory pages: Regular pages of 4 kb and large pages of 128 kb. GPU kernels only operate on GPU-virtual addresses, which a dedicated MMU, depicted in Figure 1, transparently translates to GPU-physical addresses.

Current GPU software stacks manage memory using *Buffer Objects* (BO). BOs are contiguous regions of GPU-virtual memory spanning one or more page table entries. While a BO is always contiguous in virtual memory, the same does not necessarily apply to physical memory, as the page table can map each page of the BO to an arbitrary page in physical memory.

The memory management of current GPUs lacks some features found commonly in CPUs: Current GPUs typically treat page faults as fatal errors, and their page tables do not contain reference- or dirty-bits. Therefore, well-known memory management techniques like demand paging or traditional page replacement algorithms cannot be applied to GPUs.

The GPU's page tables are typically not limited to GPU-physical memory. Instead, they can also contain physical addresses in system RAM. In that case, the GPU's MMU translates any access to a GPU-virtual address mapped to system RAM into a PCI-e bus transaction. Mapping system RAM into GPU address spaces this way is transparent to GPU kernels as these GPU kernels operate on virtual addresses only. The only distinction between GPU memory-backed and system RAM-backed virtual memory is speed: Operations targeting system RAM are limited by the bandwidth of the PCI-express bus, which is about 25x slower than the GPU's native memory bus. In essence, a GPU using both system RAM and it's own memory can thus be considered a NUMA system.

While extending the GPU's memory with system RAM is supported by current GPU software stacks, the application must typically decide at allocation time where to store a given buffer object. This implies that a single BO may not span both GPU memory and system RAM. However, this limitation stems purely from the current GPU software stacks since BOs are a software construct. The GPU hardware can map GPU-virtual addresses to GPU- or system RAM with page granularity, independent of BO boundaries.

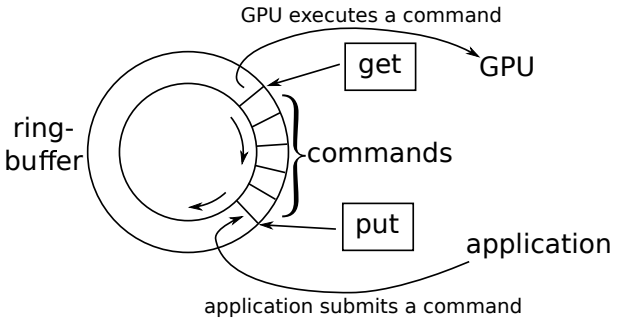


Figure 2: The GPU command submission process (© 2013 IEEE. Taken from [5] with permission)

2.2 Command submission

For most modern GPUs, applications submit GPU commands using command submission channels as depicted in Figure 2. Each channel consists of a ring buffer holding the actual GPU commands, and two pointers – *get* and *put* – which reside in memory-mapped device registers and point to the head and tail of the command queue inside the ring buffer. The GPU driver can memory-map these command submission channels – including both ring buffer and pointers – into the application's CPU address space. Applications can thus submit commands directly to the GPU, without invoking the GPU driver. An application wishing to send commands to the GPU first writes these commands into the next free slot in the command submission channel's ring buffer, and then advances the *put*-pointer, which signals the GPU that a command was just submitted. The GPU processes the commands queued in the ring buffer sequentially and advances the *get*-pointer whenever a command has been consumed.

On the downside, granting applications direct access to the command submission channels also implies that all scheduling decisions for commands from different channels are left to the GPU alone. Therefore, if multiple applications access the GPU concurrently, neither the applications nor the GPU driver have any information about the ordering of commands from different applications. In addition, once a command has been written into the command submission channel, it must execute to completion – it is not possible to un-submit a command even if that command has not yet begun to execute. Since applications cannot be sure when exactly a submitted command will execute, each application must thus ensure uninterrupted access to all data needed by each of its commands until those commands have finished execution.

3. Design goals

The main goal of GPUswap is to enable oversubscription of GPU memory. However, in doing so, GPUswap should not sacrifice application performance or compatibility with existing applications. Besides enabling oversubscription, we thus define the following objectives for our design:

Fairness From the GPU's perspective, system RAM is significantly slower than the GPU's own memory. Relocating application data to system RAM can thus result in significant runtime overhead for the application owning the data. Ideally, this overhead should be distributed fairly among applications. Unfortunately, we can not yet estimate the performance impact of storing a given data structure in system RAM. Therefore, our current goal is to guarantee a fair share of GPU memory to each application. In the future, we will investigate other relocation policies to instead divide the overhead more evenly.

Performance Our goal is thus to optimize for the common case: GPUswap should not induce overhead unless there is a shortage in GPU memory. Our reasoning behind this goal is that we expect relocation of data to system RAM to be relatively rare. The goal of sharing GPU memory is to increase the utilization of that memory. However, such sharing is ineffective when a single application fully utilizes the available memory by itself. Therefore, we expect GPU memory to be shared mainly among applications using relatively small amounts of GPU memory. In that scenario, system RAM is only used to cope with the exceptional case that an application requests more GPU memory than expected. However, if GPUswap must relocate data to system RAM, overhead is unavoidable due to the difference in speed between GPU memory and PCI-e bus. GPUswap is therefore intended as a short-term solution only. If a shortage in GPU memory persists for an extended period of time, the cloud provider should take additional action, such as migrating a VM to a different host.

Transparency Most GPU applications are written under the assumption that the application has exclusive access to the GPU. In addition, modern GPU drivers map the GPU's command submission channels directly into the application to maximize application performance. Our solution should maintain this illusion of exclusive and direct GPU access by keeping any relocation of GPU memory fully transparent to the application in order to remain compatible with existing applications. Specifically, the application should be able to submit commands to the GPU at any time, without being affected by memory relocation. Therefore, from the application's point of view, GPU memory should never become inaccessible, and the contents of GPU memory should never change unexpectedly.

4. Architecture

GPUswap uses system RAM as an extension to GPU memory. Modern GPUs can map system RAM into applications' GPU address spaces, allowing GPU kernels transparent access to system RAM. If applications allocate more GPU memory than available, GPUswap transparently copies data

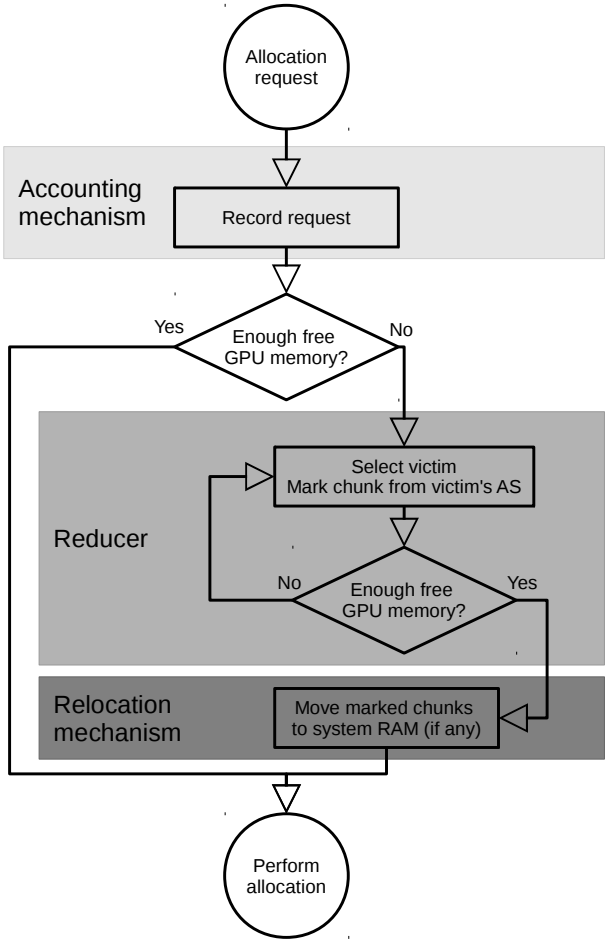


Figure 3: The main components of GPUswap and their operation

from GPU memory to system RAM, and subsequently maps that data in system RAM to the same virtual address it originated from. Applications can thus allocate more GPU memory than is physically available, while GPUswap ensures that all data remains permanently accessible, even if it does not reside in GPU memory. Therefore, in contrast to previous approaches, GPUswap does not depend on software scheduling of GPU kernels to enable oversubscription of GPU memory.

4.1 Overview

GPUswap consists of three main components as depicted in Figure 3: An accounting mechanism which tracks information about each application's allocated memory, a reducer that, based on information from the accounting mechanism, decides which memory to move to system RAM, and a swapping mechanism which executes the reducer's decisions. Whenever an application requests GPU memory, the accounting mechanism takes note of that request in order to track the BOs and the total amount of GPU memory allocated by each application. If there is insufficient free GPU memory to serve the request, the reducer then decides which memory should be relocated to system RAM to free up GPU memory for the request. Finally, the relocation mechanism performs the actual relocations. Conversely, whenever an application frees GPU memory, GPUswap selects relocated memory fitting into the available space on the GPU and moves that memory back onto the GPU in order to maintain good memory utilization. GPUswap's operation is completely transparent to applications, apart from a small delay in the application's GPU command execution during the operation of the relocation mechanism.

GPUswap is designed to operate in the context of the operating system kernel, either as an add-on to an existing GPU driver, or as a separate module wrapping calls from userspace into the GPU driver. Operating inside the kernel gives GPUswap both information about and control over all applications in the system, without requiring cooperation from those applications. Therefore, running in the kernel allows GPUswap to enforce allocation policies even against uncooperative applications.

4.2 Memory accounting

Our accounting mechanism intercepts every request for GPU memory. These requests are sent to the GPU driver by the user-space CUDA runtime whenever that runtime needs additional memory, for example to satisfy a call to `cudaMalloc()`. Once activated, our accounting mechanism logically divides the allocated BO into fixed-size chunks. The chunk size is 2 MB by default, but can be configured to different values if desired. The accounting mechanism maintains a list of all allocated chunks as well as the total amount of memory allocated for each application. Whenever GPU memory is relocated, the reducer informs the accounting mechanism to keep each application's total GPU memory usage accurate.

GPUswap manages GPU memory in chunks rather than pages or entire BOs. Relocating memory with BO granularity could result in poor memory utilization since BOs can be hundreds of megabytes in size: If GPUswap relocates a large BO in order to make room for a small allocation, most of the relocated memory would remain unused. Managing memory with page granularity limits the amount of wasted memory to the page size, but can result in high computational overhead if each page of a large BO must be processed individually. We chose to manage GPU memory in chunks as a compromise between the conflicting goals of minimizing wasted GPU memory and minimizing computational overhead: The amount of wasted memory is limited to the chunk size – which is much smaller than the typical BO size – while at the same time, the computational overhead is limited since there are far fewer chunks than pages to process. Note that the chunk size can be configured to the page size or the size of the GPU's memory – the latter resulting in each BO consisting of one chunk – should the need arise.

4.3 Victim selection

If there is not enough free GPU memory for a request, the reducer decides which memory to relocate to system RAM in two steps: i) Select which application should give up GPU memory (*victim selection*), ii) selecting a chunk of GPU memory owned by that application for relocation (*chunk selection*). The reducer repeats these two steps until the size of the selected memory plus the size of the pre-existing free memory is larger than or equal to the size of the new request.

The reducer uses data from the accounting mechanism to select a victim. Since it is our goal to spread the available GPU memory evenly among applications, the reducer currently chooses the application consuming most GPU memory as the victim. To determine which application uses most GPU memory, the reducer considers all data currently residing in GPU memory as well as the newly allocated BO. Considering that new BO as well as existing memory ensures that the reducer can relocate memory owned by the requesting application in case the new request causes that application to exceed its fair share of GPU memory.

Once the reducer has selected a victim, the second step is to select a chunk of GPU memory owned by that victim for relocation. Ideally, since any access to the selected memory will incur a significant performance penalty after relocation, the reducer should select a chunk that the application will not use in the near future. Unfortunately, typical algorithms used to achieve a good selection – such as LRU or LFU – are unusable on GPUs since the GPU's MMU does not implement a reference bit and is therefore unable to track page accesses. Consequently, we currently revert to selecting a random chunk owned by the victim. Once a chunk has been selected, the reducer marks that chunk for relocation and reduces the victim's accounted GPU memory consumption by the size of the selected chunk.

4.4 Relocation mechanism

Once the reducer has selected appropriate chunks, the relocation mechanism actually moves these chunks to system RAM. During relocation, the mechanism must ensure the consistency of the chunks' contents. To that end, the relocation mechanism performs the following steps:

1. Temporarily suspend GPU access for the application owning the chunk
2. Copy the chunk's contents to system RAM
3. Modify the application's GPU page tables so that the chunk's location in the application's GPU-virtual address space maps to the new location of the BO in system RAM
4. Return to 2 if the application owns another marked chunk
5. Restore GPU access for the application

Suspending GPU access for the application is necessary to ensure the consistency of the chunks' contents during relocation: If the application is allowed to execute GPU kernels during copying, one of these kernels could write to a chunk while that chunk is being copied. If such a write occurs in a region of the chunk that has already been copied, this modification is lost upon switching the page table to the now outdated version of the chunk in system RAM. Unfortunately, we cannot apply the classical techniques for solving this problem to GPUs due to the limited capabilities of the GPU's MMU. For example, using write faults to detect changes to the copied memory, as is typically done for virtual machine migration [2], does not work on current GPUs since these GPUs treat write faults as fatal errors. Therefore, there is currently no alternative to suspending the application's GPU access altogether.

Our swapping mechanism uses the same technique as LoGV [5] to suspend GPU access for an application: The mechanism transparently unmaps all command submission channels from the application's address space, replaces those channels with shadow copies in system RAM, and waits for all commands in the unmapped command submission channels to finish execution. For the application, the shadow copies are indistinguishable from regular command submission channels, which maintains the illusion of uninterrupted GPU access and thus keeps memory relocation transparent to the application. In particular, the application can still submit commands without blocking, since these commands are transparently written to a shadow copy. After the relocation finishes, the mechanism restores the application's GPU access by synchronizing the contents of the shadow copies with the physical channels – causing all GPU commands submitted to a shadow copy to begin execution – before remapping the physical channels back into the application's address space. Our relocation mechanism only suspends GPU access for one application at a time to prevent the GPU from idling, and copies all marked buffers in the disabled application's address space at once while the application is suspended.

Our relocation mechanism assumes that there is sufficient system RAM to hold all selected chunks. We consider this assumption reasonable since current server machines typically contain much more system RAM than GPU memory. However, the mechanism cannot use ordinary application memory for relocating chunks: The memory holding these chunks must be non-pageable and DMA-accessible to make the relocated chunks accessible to the GPU. Since appropriate memory is easy to allocate for a device driver, we chose to allocate the memory for relocated chunks in the driver's memory space for simplicity. In principle, however, GPUswap can use any memory with the desired properties for relocation. Allocating memory for relocation in the driver is thus not a strict requirement if appropriate memory can be obtained by other means.

4.5 Returning memory to the GPU

Before any data is relocated, the GPU memory must be fully utilized to minimize the overhead associated with the use of

system RAM. Therefore, GPUswap attempts to move suitable chunks from system RAM back to GPU memory whenever an application frees GPU memory. GPUswap considers a chunk suitable if i) that BO resides in system RAM, and ii) the chunk's size is less than or equal to the amount of free GPU memory. GPUswap chooses which chunks to move back to the GPU in two steps: First, GPUswap selects one application owning at least one suitable chunk as the winner. That winner is currently the application owning the least amount of GPU memory to distribute GPU memory fairly among applications. Then, GPUswap randomly chooses one of that winner's suitable chunks and marks that chunk for relocation back onto the GPU. GPUswap repeats these two steps until no suitable chunks remain.

Once GPUswap has selected a set of chunks, we employ the relocation mechanism described in Section 4.4 to move the marked chunks back to the GPU. In essence, the relocation mechanism repeats the same steps as for relocating chunks to system RAM, only this time using GPU memory as the destination. First, the mechanism chooses an application owning at least one marked chunk and suspends that application's access to the GPU. Then, the mechanism copies all marked chunks owned by that application back into GPU memory, and modifies the application's GPU page tables to keep the chunks accessible in the same (virtual) locations. Finally, the swapping mechanism restores the application's GPU access, and advances to the next application owning marked chunks.

5. Prototype implementation

We integrated our prototype implementation of GPUswap into the PathScale GPU driver (pscnv) [13]. Pscnv is currently the only open-source GPU driver capable of mapping the GPU's command submission channels into user space. Unfortunately, pscnv limits our current implementation to Nvidia Fermi GPUs. In principle, however, our approach applies to all GPUs that feature virtual address spaces and multiple command submission channels, which includes newer GPU generations from both Nvidia and AMD. We are currently working on an implementation based on the Nouveau driver which will support Nvidia Kepler- and Maxwell-generation GPUs.

Most of GPUswap's functionality is implemented as an add-on separate from the main components of the original pscnv driver. GPUswap is activated through a hook in pscnv's memory allocator which invokes GPUswap once for each memory allocation request. GPUswap then operates as described in Section 4.1: The reducer selects appropriate chunks of GPU memory to relocate, before the relocation mechanism copies the selected chunks into system RAM. This process frees up enough memory to subsequently allow pscnv's original memory allocator to serve the original request. Since this entire process is implemented inside pscnv without requiring changes to the driver's API, GPUswap's operation is completely transparent to applications.

5.1 Memory accounting and management

The original pscnv driver manages memory as BOs, which are contiguous in virtual GPU memory and can be hundreds of megabytes in size. However, we prefer a smaller entity of memory management as explained in Section 4.2. Therefore, we split each allocated BO into fixed-size chunks. If the size of the BO cannot be evenly divided by the chunk size, the remainder is put into a separate chunk that is smaller than the configured size. We ensure that these chunks appear as contiguous BOs towards user space to keep this modification transparent to applications. Internally, we compose the chunks of large pages whenever possible to minimize the amount of page table manipulation.

Since each chunk may be placed in GPU or system memory individually, our accounting mechanism maintains per-application lists of GPU memory chunks which GPUswap can potentially move to system RAM, as well as chunks that have already been moved. Since pscnv lacked any per-application resource accounting, we introduced a new data structure that contains these two lists as well as general memory consumption statistics for each application. Whenever an application allocates a BO, the allocator adds all chunks of that BO to the application's list of chunks that can be moved to system RAM. Similarly, whenever our relocation mechanism moves chunks to system RAM, it also moves the appropriate entries to the list of already relocated chunks.

5.2 Reducer

On each request for GPU memory, our reducer executes in the kernel context of the thread that submitted the request. Since relocating memory requires suspending GPU access for the application owning that memory – which is a costly operation – to guarantee the consistency of the relocated memory's contents, the reducer does not perform the relocation immediately. Instead, it adds each selected chunk to a list attached to the state data structure of the application owning the selected chunk. After enough memory has been selected, the reducer triggers the relocation mechanism which then performs the actual relocations. The reducer then waits for all enqueued relocation operations to complete, in order to ensure that pscnv's memory allocator will subsequently find sufficient free GPU memory.

5.3 Suspending applications

GPUswap uses a similar mechanism as LoGV [5] to suspend GPU access for applications: GPUswap unmaps all GPU command submission channels from the application's address space and replaces those channels with identical shadow copies in system RAM. To restore GPU access after relocation, GPUswap copies all newly submitted GPU commands from the shadow copies to the physical channels and

subsequently maps the physical channels back into the application's address space.

To avoid creating inconsistencies while a shadow copy is being created or synchronized with a physical channel, we must prevent the application from modifying both the physical channel and the shadow copy while the operation is in progress. Therefore, we perform these operations while neither the command submission channel nor the shadow copy are mapped into the application's address space. If the application accesses the unmapped channel, the resulting page fault is directed to GPUswap, which stalls its response until the operation is complete. Except for a possible delay in the page fault handler, both suspending and resuming GPU access are thus completely transparent to applications.

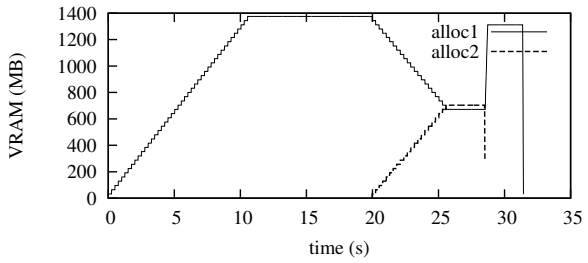
5.4 Relocation mechanism

After our reducer finishes operation, the relocation mechanism iterates over all applications owning at least one selected chunk, suspends each application's GPU access using the mechanism described in Section 5.3, and copies all chunks from the application's selected chunk list to system RAM. Note that only chunks that already exist in GPU memory are copied in this way: If chunks from the newly allocated BO are selected for relocation, these chunks are allocated in system RAM directly.

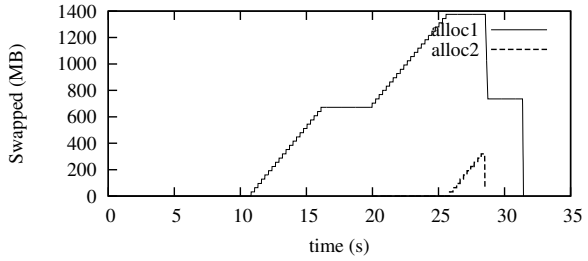
Since GPUswap is a part of pscnv, our relocation mechanism can use utility functions that are part of pscnv, for example to manipulate the GPU's page tables. However, pscnv assumes that the GPU's command submission channels are mapped into the application, and thus does not include important pieces of application logic. For example, pscnv itself cannot submit commands to the GPU, which is necessary to initiate DMA transfers. We therefore backported the missing pieces necessary for DMA transfers from gdev and pscnv's userspace library into the pscnv kernel module. Our current implementation supports asynchronous DMA, which allows our relocation mechanism to relocate all queued chunks in parallel while the application is suspended.

5.5 Returning memory to the GPU

GPUswap performs return operations in a separate thread executing in kernel space. Since the GPU applications we examined tend to free multiple BOs in short succession, that thread checks for unused GPU memory in regular intervals. These intervals should be as long as necessary to capture each set of free operations in a single interval with high probability, but otherwise as short as possible to ensure that GPU memory does not remain unused for extended periods of time. We currently set the interval to 50 ms, which fulfills both conditions to our satisfaction. If our thread detects unused GPU memory at the end of an interval, the thread chooses a set of chunks to relocate back into GPU memory and executes the appropriate relocation operation, which involves the same steps as relocation into system RAM.



(a) Total amount of allocated GPU memory



(b) Total amount of allocated system RAM

Figure 4: Total amount of memory allocated by two memory-intensive processes over time

6. Experimental Evaluation

GPUswap's main goal is to enable oversubscription of GPU memory while maintaining fairness, performance and transparency. As described in Section 5, our approach achieves transparency by operating completely inside the driver, not requiring any changes to applications. To show the fairness and performance of our approach, we conducted a number of experiments using our prototype implementation.

6.1 Experimental setup

We used a Nvidia GeForce GTX 480 GPU as a testbed for our experiments. The GPU is based on the Fermi microarchitecture and features 480 cores and 1.5 GB of GDDR5 memory. Our host system consists of a Intel Core i7-4470 CPU and 16 GB of system RAM. For our experiments, we locked both CPU and GPU at the highest available clock frequency. For our benchmarks, we used Gdev's CUDA implementation (ucuda) [9], which supports both Gdev and pscnv. Our host system ran Ubuntu 12.04.5, which is based on Linux 3.5.7.33.

6.2 Fairness

Since GPU memory offers much higher performance than system RAM, the available GPU memory should be distributed fairly among applications. To that end, GPUswap attempts to guarantee a fair amount of GPU memory to each application. We created a synthetic benchmark application called *alloc* to evaluate whether our approach fulfills that guarantee. Alloc performs no computation, but instead con-

Application	Area
backprop	Machine learning
bfs	Graph computation
heartwall	Image processing
hotspot	Physics simulation
lud	Linear algebra
srad2	Image processing

Table 1: The benchmarks used in our evaluation

tinuously allocates BOs of 32 MB until it possesses a total of 2 GB of virtual GPU memory. Alloc then waits a pre-defined amount of time before freeing all allocated memory at once. Our reasoning behind this benchmark was to simulate the behavior of a program containing a memory leak. In our experiment, we started two instances of alloc – named alloc1 and alloc2 – with alloc2 starting 20 seconds after alloc1. We then recorded the total amount of both GPU memory and system RAM allocated by both instances combined. We set the chunk size to 32 MB to make the relocation operations more visible.

The results are shown in Figure 4. Alloc1 starts at time 0 and gradually fills up all available GPU memory. When no more GPU memory is available, our reducer starts moving data to system RAM in response to new allocations. As a result, the amount of allocated system RAM starts to increase until alloc1 owns a total of 2 GB of memory. 20 seconds after the launch of alloc1, alloc2 starts and attempts to allocate memory. As the entire GPU memory has been allocated by alloc1, our reducer chooses chunks owned by alloc1 – which obviously exceeds its fair share of memory – for relocation. As a result, alloc1's amount of GPU memory decreases at the same rate as the amount of GPU memory allocated to alloc2 increases until both instances own approximately the same amount of GPU memory. Note that the amounts of memory owned by both instances are not exactly equal: Since the 1400 MB of GPU memory available to applications do not divide equally by 32 MB, alloc2 ends up with one more chunk than alloc1. Since this kind of imbalance is limited to one chunk, we consider the amount of unfairness acceptable.

6.3 Performance

We used six benchmark applications from the rodinia benchmark suite [1] – which have been previously adapted to ucuda [8] – to evaluate the performance of our prototype. The resulting set of benchmarks is listed in Table 1.

We ran each of our benchmark applications multiple times while gradually increasing the amount of available GPU memory in 50 MB increments to evaluate the effect of using system RAM instead of GPU memory. To limit the available GPU memory, we modified the pscnv kernel module, forcing pscnv's memory allocator to ignore all GPU memory above a configurable address. At each memory size we tested, we started two instances of each benchmark application simultaneously. Since our reducer selects 2 MB chunks for relocation at random, we ran each benchmark application 10 times to allow our reducer to select different combinations for chunks. In addition, we modified our benchmark applications to repeat their main computation step 100 times in order to make the effect of using system RAM in place of GPU memory more visible. To that end, we added a loop comprising all GPU kernel launches and as much of the application's I/O as possible to each of our benchmark applications. The only exception to this modification was hotspot: Hotspot already executes its GPU kernels for a number of iterations, which allowed us to repeat the main computation step by simply setting the desired number of iterations appropriately.

For comparison, we ran this experiment on both GPUswap and gdev. We chose gdev since it is currently the only freely available tool for enabling oversubscription of GPU memory that we are aware of. Gdev enables oversubscription by transparently sharing BOs between applications: Whenever memory pressure occurs, Gdev selects two BOs of similar size owned by different applications. Gdev then copies the contents of one of those BOs into system RAM, and maps the other BO into the GPU address space of both applications. Whenever a GPU kernel of one of the two applications is then scheduled for execution, Gdev copies that application's data into the shared BO before starting the kernel. To implement limiting of GPU memory, we added the same modification we made to pscnv to the gdev kernel module.

Figure 5 shows the runtime for each of our benchmark applications for various GPU memory sizes on both GPUswap and gdev. For each application, we recorded the runtime of the slower of the two instances. As expected, our solution induces less runtime overhead than gdev for most of our benchmark applications if all application data resides on the GPU, which is the case for all applications at 500 MB GPU memory. Most notably, gdev's overhead for backprop was more than 30% compared to GPUswap, followed by srad2, lud and heartwall with 14%, 10% and 3.5% overhead, respectively. The only notable exception was hotspot, for which GPUswap showed an average overhead of 1.5% compared to gdev. Bfs did not show any significant difference in runtime between GPUswap and gdev. Overall, we conclude that GPU software scheduling, as implemented by gdev, causes significant runtime overhead for most applications, while GPUswap – which only interrupts GPU computation in case of memory pressure – does not induce overhead for most applications as long as sufficient GPU memory is available.

On the downside, our benchmarks also show the cost of relocating application data to system RAM: At a total GPU memory size of 20 MB – at which almost all application data resides in system RAM – the runtime of all applications increases significantly – from 50% for backprop to 786% for bfs. For most applications, however, the overhead ap-

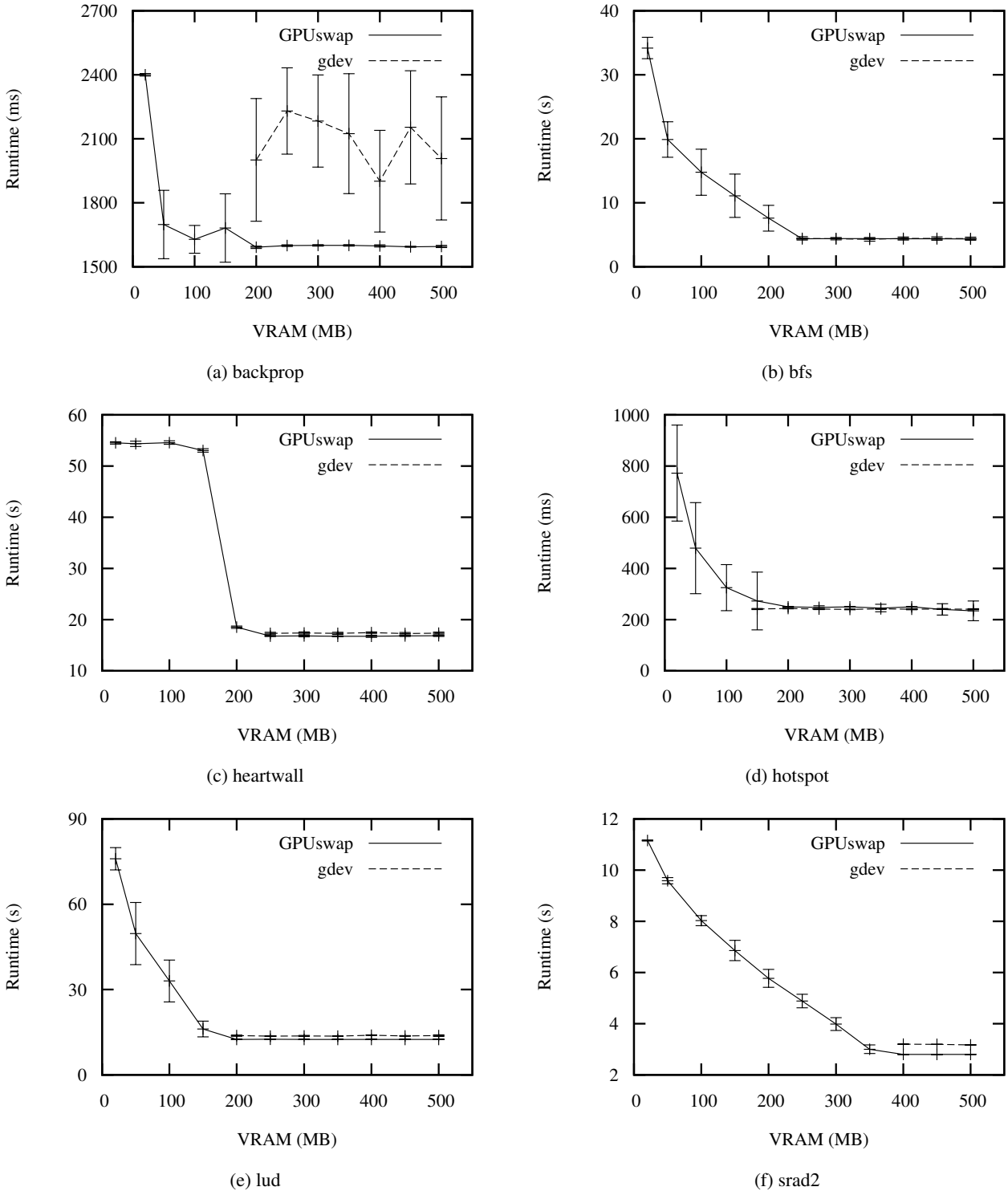


Figure 5: The runtime of two instances of each benchmark application for various amounts of GPU memory. The x-axis shows the amount of GPU memory available to applications, while the y-axis shows the average runtime of the slower of the two instances. Unfortunately, Gdev caused all benchmark applications to crash under memory pressure. Therefore, results for Gdev are only shown for cases where sufficient GPU memory is available.

pears to decrease exponentially as we add more GPU memory. Consequently, if only a small amount of application data is relocated, the overhead becomes relatively small as well. Unfortunately, we could not compare these results to Gdev since all benchmark applications invariably crashed as soon as Gdev detected memory pressure. However, since we expect all application data residing in system RAM to be an exceptionally rare case and GPUswap is intended mainly as a short-term solution, we consider GPUswap's overhead acceptable overall.

Since GPUswap selects memory chunks for relocation randomly, and different chunks may have a different impact on performance when relocated, the runtimes of most of our benchmark applications fluctuate heavily as soon as any application data is relocated to system RAM. This fluctuation is shown by the error bars in Figure 5, which show the standard deviation of the benchmark runtimes across all runs at each memory size. It is important to note that for some benchmarks, the low end of this standard deviation comes close to the application's runtime with sufficient GPU memory available, which indicates that GPUswap's random selection sometimes selects chunks for relocation which have only a negligible impact on performance. Unfortunately, it is currently not possible to detect these chunks beforehand since current GPUs lack appropriate hardware support, such as reference bits in the GPU's page tables. However, we believe that hardware support for detecting rarely accessed chunks has the potential to greatly improve GPUswap's performance.

We repeated some of our experiments with up to five concurrent instances of our benchmark applications to assess the scalability of GPUswap. Our results indicate that the increase in computational overhead due to the larger number of applications can be neglected – in practice, the runtime overhead appears to depend only on the amount of relocated application data. Our experiments also show that the overhead is spread evenly among all running applications. These results are consistent with our expectations: Since Fermi GPUs run GPU kernels from different applications sequentially, there is no additional contention on the PCI-e bus if more applications are started in parallel. However, bus contention could be a problem on GPUs capable of running kernels from different applications concurrently.

6.4 Relocation delay

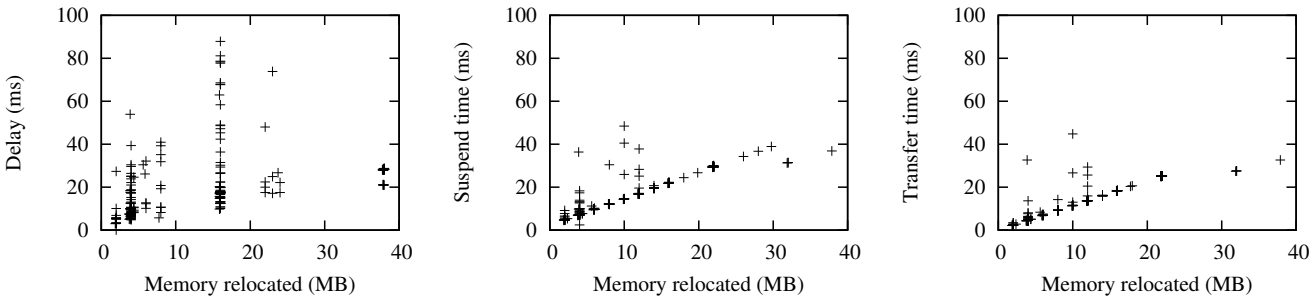
In addition to the direct cost of using system RAM in place of GPU memory, GPUswap can also cause delays in applications while memory is being relocated. Specifically, each allocation request can cause two types of delay: i) The allocating application itself may have to wait for a relocation to complete before the actual allocation can take place, and ii) the GPU access for another application may be suspended if memory owned by that application is relocated in response to an allocation request.

We ran the same benchmark applications as in the previous experiment with the total amount of GPU memory limited to 100 MB to measure the impact of these delays. We ran each benchmark application five times, again starting two instances of each application simultaneously. While the applications were running, we recorded the amount of memory relocated and the delay caused by each individual relocation operation. The results are depicted in Figure 6. Figure 6a shows the delay experienced by applications requesting GPU memory, depending on the amount of GPU memory that must be relocated to make room for the request. Note that each of our benchmark applications allocates the same buffers in each run, which is why the memory sizes appear quantized in this figure. Figure 6b shows the delay experienced by applications that must give up memory in response to an allocation request from another application. Finally, Figure 6c shows the time taken for the raw DMA transfers taking place during relocation, depending on the size of the transfer. The chunk size in this experiment was set to the default of 2 MB, which causes the apparent quantization in Figures 6b and 6c.

The allocation requests we observed were generally delayed by less than 100 ms. We consider this delay acceptable since most GPU applications tend to re-use previously allocated memory instead of allocating and freeing buffers frequently. However, the delay did not always mirror the time needed for a DMA transfer of the same size. The reasons for this result are twofold: First, GPUswap performs work of its own in addition to the DMA transfer – for example to select which chunks should be relocated – which can cause allocation requests to be delayed longer than the time needed for the DMA transfer. Second, the delay may be shorter than the time for a DMA transfer if chunks from the newly requested memory are selected for relocation, which results in those chunks being allocated in system RAM directly, without the need for a DMA transfer.

The delays we observed in applications not allocating memory themselves were even shorter on average than the delay for allocation requests. This shorter delay is owed to the fact that GPUswap performs almost all of its computation – such as chunk selection – in the context of the application requesting memory. Therefore, the delay experienced by other applications is dominated by the time needed for the DMA transfer, which can be seen from the mostly linear relationship between the amount of relocated memory and the delay in Figure 6b and Figure 6c. However, we observed that some relocations delayed the application longer than average. We assume that these relocations ran concurrently with computation on previously relocated memory and thus suffered from contention on the PCI-e bus. However, even these longer relocations generally completed in under 50 ms.

We again repeated these experiments with more than two applications. Our results indicate that multiple concurrent applications can introduce considerable jitter to the dura-



(a) Memory allocation delay (b) Delay for applications giving up memory (c) DMA transfer only

Figure 6: The results of our measurements of relocation delay. The plots show the relocation delay experienced by an application allocating memory (left) and an application that must give up memory due to an allocation request from another application (center). The plot on the right shows only the time for the DMA transfer that takes place during relocation.

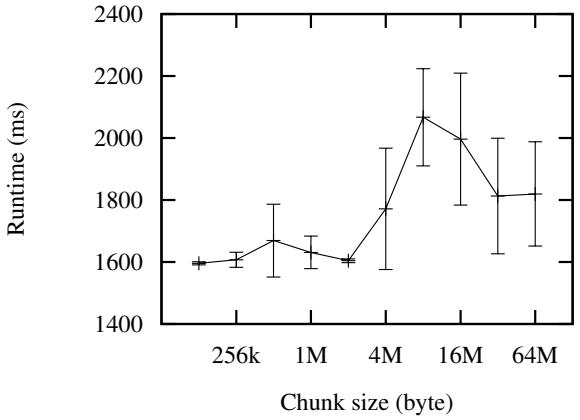


Figure 7: Total runtime of backprop for various chunk sizes

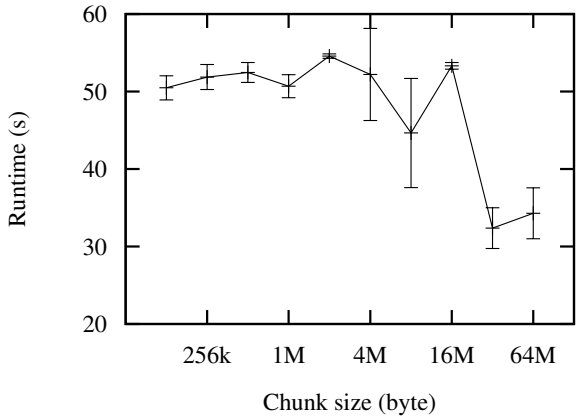


Figure 8: Total runtime of heartwall for various chunk sizes

tion of the DMA transfers, and hence to both applications requesting memory and applications losing memory due to allocation requests from other applications. This increased jitter was to be expected since more concurrent applications increase the potential for contention on the PCI-e bus.

6.5 Chunk size

GPUswap manages GPU memory in chunks as a compromise between granularity and computational overhead. In order to study the runtime effects of the chunk size, we ran two instances of each of our benchmark applications with the available GPU memory limited to 100 MB, while gradually increasing the chunk size from 128 kb – which equals one large page – to 64 MB. For each chunk size, we started each application ten times. We first measured the total runtime of each application's main computation step as described in our second experiment (Section 6.3). Our results indicate that the effect of different chunk sizes on the application runtime depends heavily on the application. Some applications, such as backprop (Figure 7), benefit from small chunk sizes, while others, such as heartwall (Figure 8), prefer larger chunks. As the effect of the chunk size on the other applications was generally weaker than on backprop and heartwall, we chose not to show these results for brevity. However, the results do not indicate a clear trend towards a specific chunk size. We therefore conclude that there is no single chunk size that is optimal for all applications in terms of total runtime. In the future, we plan to investigate why specific chunk sizes are optimal for certain applications, which will allow us to select an appropriate chunk size dynamically for each application.

We also measured the average delay experienced by memory allocation requests across all applications as described in Section 6.4 for chunk sizes from 128 kb to 64 MB. Our results, which are depicted in Figure 9, indicate that a chunk size of 2 MB minimizes the allocation delay. At smaller chunk sizes, GPUswap must consider a large number of chunks, which leads to a high computational overhead if a large amount of memory must be relocated, while at larger chunk sizes, the delay is dominated by long DMA transfer times. For the moment, we therefore chose 2 MB as GPUswap's default chunk size.

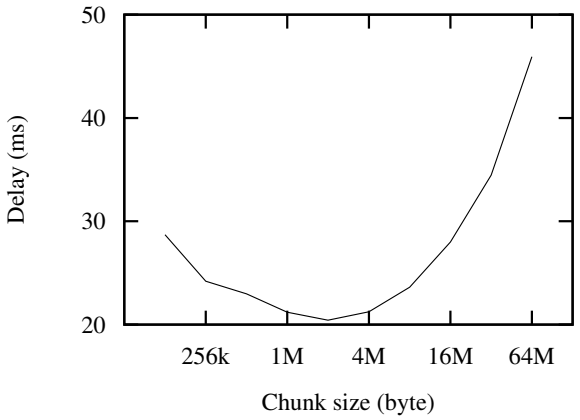


Figure 9: The average allocation delay across all benchmark applications for various chunk sizes

7. Related Work

Several previous projects have considered the use of GPUs in a shared environment. GViM [6], gVirtuS [4], rCUDA [3] and vCUDA [15] all work by intercepting high-level (e.g., CUDA or OpenCL) GPU commands in the guest VM and forwarding those commands to the hypervisor. TimeGraph [10] and PTask [14] instead intercept GPU commands at the operating system level, either by trapping accesses to the GPU's command submission channels, or by offering a custom API. GPUvm [16], gVirt [17] and LoGV [5] take a hybrid approach, intercepting only commands related to GPU resource allocation, while granting the application direct access to the GPU's command submission channels. Though all of these projects enable sharing of a GPU between multiple guest VMs, to our knowledge, none of them specifically address GPU memory. Note that while GPUswap re-uses some techniques from LoGV – most notably unmapping of command submission channels – the two are completely separate projects: LoGV does not address memory fairness, while GPUswap is not aware of guest applications. However, the two could likely be integrated for use in production cloud environments.

Gdev [11] implements software scheduling of low-level GPU commands, and uses its control over GPU command execution to enable a limited form of GPU buffer sharing between applications. Gdev extends the available GPU memory by alternating the contents of shared buffers between applications during scheduling, always copying the content of the next application to run onto the GPU. GDM [18] later removed virtually all of Gdev's restrictions by instead copying the entire application memory to the GPU during scheduling. Both Gdev and GDM can thus allocate more GPU memory than physically available; however, both also rely on software scheduling, thus inducing considerable runtime overhead even if enough free GPU memory is available.

Nvidia's Unified Memory, which was integrated into the CUDA SDK [12] in version 6, creates a single address space shared between CPU and GPU by transparently synchronizing the contents of system RAM and GPU memory. However, to the best of our understanding, Unified Memory does not enable oversubscription of GPU memory since all GPU memory shared in this fashion must be allocated on the GPU. In addition, Unified Memory is implemented in the user-space CUDA runtime and can thus not evict other applications' memory from the GPU, which makes it impossible to enforce fairness.

RSVM [7] also manages GPU memory on the application level. RSVM consists of a shared library which hides the complexity of GPU memory management from the application and is able to extend GPU memory using system RAM. However, applications must use the RSVM library explicitly. RSVM thus depends on cooperation from the application and is therefore not suitable for a cloud environment containing multiple, mutually untrusted applications.

8. Conclusion

In this paper, we presented GPUswap, a novel approach to enabling oversubscription of GPU memory that does not depend on software scheduling of GPU kernels. GPUswap transparently relocates application data from the GPU into system RAM, while keeping this data permanently accessible to the application. Therefore, GPUswap allows applications to submit commands to the GPU directly at any time, without involving the GPU driver or GPUswap. Experiments with our prototype implementation indicate that the avoidance of software scheduling in GPUswap increases performance while sufficient GPU memory is available. However, relocating large portions of application data to system RAM has a significant impact on application performance.

GPUswap is under active development, and there are two main issues that we plan to address in the future. First, since GPUswap is still limited to Nvidia Fermi GPUs, we are working on porting our prototype to the Nouveau driver, which will allow us to evaluate GPUswap on Nvidia's latest GPU hardware. Second, it is currently difficult to select appropriate memory for relocation due to the lack of appropriate hardware support – such as reference bits – in current GPUs. We are currently exploring ways to limit the performance impact of GPUswap by detecting particularly poor selections – for example, the GPU's built-in performance counters could allow us to detect an increased number of bus transactions in response to a chunk relocation.

References

[1] 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 5th International Symposium on Workload Characterization*, IISWC '09, pages 44–54, Austin, TX, Oct. 2009. IEEE.

[2] 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 Symposium on Networked Systems Design & Implementation*, NSDI '05, pages 273–286, Boston, MA, USA, May 2005. USENIX Association.

[3] J. Duato, A. Pena, F. Silla, J. Fernandez, R. Mayo, and E. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 18th International Conference on High Performance Computing*, HiPC '11, pages 1–10, Bengaluru, India, Dec. 2011.

[4] G. Giunta, R. Montella, G. Agrillo, and G. Coviello. A GPGPU transparent virtualization component for high-performance computing clouds. In *Proceedings of the 16th International Euro-Par Conference on Parallel processing*, Euro-Par '10, pages 379–391, Naples, Italy, Sept. 2010. Springer.

[5] 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.

[6] 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, Nuremberg, Germany, Apr. 2009. ACM.

[7] 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.

[8] S. Kato. Rodinia for gdev. `https://github.com/shinpei0208/gdev-bench`, Nov. 2014.

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

[10] S. Kato, K. Lakshmanan, R. 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.

[11] S. Kato, M. McThrow, C. Maltzahn, and S. 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, USA, June 2012. USENIX Association.

[12] Nvidia Corporation. CUDA toolkit. `https://developer.nvidia.com/cuda-toolkit`, 2014.

[13] PathScale. Pscnv. `https://github.com/pathscale/pscnv`, Nov. 2014.

[14] 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 23th Symposium on Operating System Principles*, SOSP '11, pages 233–248, Cascais, Portugal, Sept. 2011. ACM.

[15] 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.

[16] 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.

[17] 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.

[18] 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, USA, June 2014. ACM.