Karlsruher Institut für Technologie

Transparente Erweiterung von GPGPU- Speicher durch Hauptspeicher

Bachelorarbeit
von

Jonathan Metter

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Dipl.-Inform. Jens Kehne

Bearbeitungszeit: 14. April 2014 – 13. August 2014

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 7. August 2014

Zusammenfassung

GPGPUs eignen sich zur Beschleunigung einer Vielzahl von Anwendungen und sind daher auch für Cloud Umgebungen interessant, in denen die Grafikkarte virtualisiert und auf mehrere Benutzer aufgeteilt wird. Nicht nur in diesem Szenario ist es schwer zu vermeiden, dass die Anwendungen mehr Speicher anfordern, als auf der Grafikkarte vorhanden ist. Wir halten es für nötig, dass Betriebssysteme diesen Fall behandeln, anstatt die Verwaltung des schnellen, aber kleinen, Videospeichers der Anwendung zu überlassen. Vorhandene Methoden zur Speicherverwaltung benötigen direkte Kontrolle über den Befehlsstrom zwischen Anwendung und Grafikkarte, während sich der Overhead für die Virtualisierung durch Verzicht auf diese direkte Kontrolle reduzieren lässt.

In dieser Bachelorabeit begegnen wir diesem Widerspruch, indem wir eine Methode vorstellen, die auch ohne direkte Kenntnis und Kontrolle über die Anwendung, für die Anwendung transparent, Videospeicher durch Hauptspeicher ersetzt um Kapazitätsüberschreitungen zu behandeln. Neben unserem eigenen Verfahren stellen wir alternative Ansätze zur Speicherverwaltung von GPGPUs vor und beschreiben die Verwaltung von Videospeicher in bestehenden Open- Source Treibern im Zusammenhang mit der Architektur von Grafikkarten der Nvidia- „Fermi“ Generation.

Wir haben unser Verfahren in einem Prototypen implementiert. Mit Hilfe dieses Prototypen untersuchen wir, in welchem Maße verschiedene GPGPU- Anwendungen durch das Swapping ausgebremst werden. Unser Prototyp ist in der Lage Speichermangel vor der Anwendung zu verbergen und den VRAM gleichmäßig auf mehrere Anwendungen aufzuteilen. Unsere Experimente zeigen, dass der Overhead für das Swapping von Buffern, auf die selten zugegriffen wird, gering ausfällt.

Inhaltsverzeichnis

Kapitel 1

Einführung

Wegen ihrer hohen Leistung und Effizienz bei vielen Aufgaben [12, 32, 38, 43, 61] sind GPGPUs aus dem High Performance Computing (HPC) nicht mehr wegzudenken und werden zunehmend für Cloud Umgebungen, in denen die Hardware auf mehrere Kunden aufgeteilt wird (Infrastructure as a Service [44]) interessant.

Die Vorteile von GPUs gegenüber CPUs bestehen in einer auf Parallelität optimierten Architektur und dem schnellen Videospeicher (VRAM), der auf Grafikkarten verbaut wird. Diese Eigenschaften wurden ursprünglich für Grafikanwendungen entwickelt, in welchen typischerweise die Grafikkarte von einer einzigen Anwendung genutzt wird, so dass diese mit dem gegenüber Hauptspeicher (SYSRAM) kleinen VRAM [59] haushalten kann. Computerspiele oder Videoplayer binden die Aufmerksamkeit des Nutzers, weshalb der Nutzer keine weiteren grafischen Anwendungen ausführt.

Anwendungen, die die GPU dagegen für allgemeine Berechnungen (GPGPU) nutzen, greifen auf die GPU zu, wann immer anspruchsvolle Berechnungen anstehen. Die implizite Zuweisung der Grafikkarte durch den Benutzer fällt weg. Folglich tritt häufiger der Fall ein, dass unabhängige Anwendungen die GPU gleichzeitig benötigen und demzufolge unabhängig VRAM reservieren, wodurch die Kapazitäten der Grafikkarte schnell erschöpft werden. Die Hardware- Unterstützung für die Speicherverwaltung ist bei heutigen GPUs gegenüber CPUs eingeschränkt. Insbesondere bieten GPUs bislang keine Möglichkeit nach einem fehlgeschlagenen Speicherzugriff („Pagefault“) die fehlenden Daten nachzuladen, um die Berechnung fortzusetzen. Marktübliche GPGPU- Treiber überlassen die Behandlung von Speicherengpässen der Anwendung, weshalb sie Allokationen, für die nicht genügend VRAM verfügbar ist, fehlschlagen lassen.

Weil der Programmierer einer Anwendung das Verhalten anderer Anwendungen zur Laufzeit nicht abschätzen kann, weiß er nicht wann und wie viel Speicher diese anfordern. Er muss folglich mit fehlschlagenden Speicherallokationen rechnen und entweder eine CPU- Ausweichlösung vorsehen, wodurch sich der Pro-

grammieraufwand erhöht, oder die Anwendung abbrechen.

Wir sehen in VRAM- Engpässen ein Hindernis für die Verbreitung von Cloud-Diensten, in denen die Kunden GPU- Rechenkapazitäten einkaufen können. Der Betreiber eines Cloud- Dienstes kann seine Hardware besser auslasten und folglich seinen Kunden flexiblere und günstigere Angebote unterbreiten, wenn er die GPU, wie den Rest des Systems, ebenfalls virtualisiert und auf mehrere Kunden aufteilt. Ein solches System anzubieten wird durch zwei Faktoren erschwert: zum einen der Overhead für die Virtualisierung der GPU, zum anderen durch die Größe des VRAM. Die Kunden erwarten, dass ihre Berechnungen zuverlässig ausgeführt werden, egal wie das System gegenwärtig durch andere Kunden ausgelastet wird. Für viele Kunden ist jedoch ein statisch zugeteilten Anteil am VRAM zu klein, denn diese Kunden wollen auch große Datensätze effizient verarbeiten.

Als Ausweg bietet sich eine dynamische Speicherverwaltung an, die jeder Anwendung erlaubt den gesamten VRAM auszunutzen, solange dieser von keiner anderen Anwendung benötigt wird. Eine derartige Speicherverwaltung sollte weiterhin die zuverlässige Ausführung der Anwendungen gewährleisten, wenn der Bedarf an VRAM die Kapazitäten übersteigt, indem die Speicherverwaltung Anwendungen, die unverhältnismäßig viel Speicher belegen, Speicher entzieht. Dieser Vorgang sollte für die Anwendung transparent sein und folglich keine Kooperation der Anwendung erfordern. Aus diesem Grund liegt es nahe die Speicherverwaltung als Teil des Betriebssystems zu realisieren. Moderne Betriebssysteme verwalten bereits den SYSRAM in einer Weise, die allen genannten Ansprüchen gerecht wird [55, S. 173ff.], unter anderem, indem sie Inhalte aus dem SYSRAM auf die Festplatte auslagern („Swapping“). Wir schlagen einen Swapping- Mechanismus für den VRAM vor, der Inhalte in den SYSRAM auslagert, denn SYSRAM ist auf den meisten Systemen reichlich vorhanden, günstig erweiterbar und die GPU kann per DMA effizient Daten zwischen VRAM und SYSRAM austauschen.

In [39, 41, 59] wurden bereits Verfahren zum Swapping von VRAM vorgeschlagen. Diese können allerdings den Speicher entweder nicht anwendungsübergreifend verwalten oder sie steuern per Software, welche Anwendung zu einem gegebenen Zeitpunkt Zugriff auf die Hardware hat („Scheduling“). In letzterem Fall kopieren sie bei jedem Wechsel zwischen den Anwendungen den gesamten ausgelagerten Speicher derjenigen Anwendung, die als nächstes die GPU nutzt, in den VRAM zurück und verdrängen gleichzeitig Inhalte anderer Anwendungen aus dem VRAM. Weil alle Daten der aktuellen Anwendung im VRAM liegen, sind Pagefaults auf gültigen Adressen ausgeschlossen. Ein Nachteil dieser Arbeitsweise ist, dass die Phasen, in denen die GPU genutzt werden kann, von Kopieroperationen unterbrochen werden. Außerdem erschwert das Scheduling eine effiziente Virtualisierung: wie mit dem GPGPU- Virtualisierungskonzept LoGV [31] gezeigt wurde, ist es möglich GPUs mit geringem Overhead zu virtualisieren, indem man

Anwendungen im Gast- System ermöglicht Befehle direkt auf der Hardware abzusetzen. Das direkte Absetzen von Befehlen umgeht allerdings einen etwaigen Scheduler und ist daher mit den bekannten Verfahren für GPGPU- Swapping unvereinbar.

Wir schlagen vor, ausgelagerten Speicher im SYSRAM zu belassen und lediglich in die GPU einzublenden. Moderne Grafikkarten unterstützen dies, denn sie verfügen über virtuelle Adressräume [50], in die der Treiber nicht nur VRAM, sondern auch SYSRAM, einblenden kann [3, 11]. Zugriffe der GPU auf Speicherbereiche, in die SYSRAM eingeblendet ist, verhalten sich wie Zugriffe auf den VRAM – sie sind lediglich langsamer. Aus diesem Grund ist es möglich fehlenden VRAM durch SYSRAM zu „simulieren“ und die Kopieroperationen zwischen den Phasen, in denen die GPU rechnet, zu eliminieren. Außerdem wählen wir den zu verdrängenden Speicher mit dem Ziel, den VRAM gleichmäßig unter den Anwendungen aufzuteilen, sodass Anwendungen mit besonders großem Speicherverbrauch andere Anwendungen nicht unverhältnismäßig beeinträchtigen. Ein Vorteil unseres Swapping- Mechanismus ist, dass dieser sich mit LoGV verwenden lässt, weil wir nicht auf Software- Scheduling angewiesen sind. Unser Ansatz zur Speicherverwaltung benötigt kein Scheduling, denn in unserem Entwurf ist der gesamte Speicher aller Anwendungen dauerhaft für die GPU erreichbar. Entweder die Daten sind direkt im VRAM gespeichert oder sie werden aus dem SYSRAM eingeblendet. Um Daten in den SYSRAM auszulagern, müssen wir sie zunächst vom VRAM in den SYSRAM kopieren. Dabei stellt sich die Frage, wie die Daten einer Anwendung zu kopieren sind, wenn nicht ausgeschlossen werden kann, dass die Anwendung sie während des Kopiervorgangs verändert. LoGV ist in der Lage eine GPU- Anwendung während der Ausführung auf einen anderen Rechner zu migrieren, indem es der Anwendung vorübergehend und transparent den Zugriff auf die GPU entzieht. Wir wenden dieses Verfahren an, um das Swapping ohne Gefahr von Inkonsistenzen durchzuführen.

Wir haben einen Prototypen für unseren Ansatz in Form einer Treiber- Modifikation implementiert. Dieser Prototyp belegt, dass unser Ansatz technisch umsetzbar ist und dass der Prototyp Anwendungen mehr Speicher zur Verfügung stellt, als VRAM physisch vorhanden ist. Ferner erzwingt der Prototyp eine gleichmäßige Aufteilung des VRAM. Auf Basis dieses Prototypen haben wir Benchmarks durchgeführt, die zeigen, dass der Overhead für das Swapping von Buffern, auf die selten zugegriffen wird, gering ausfällt. Außerdem haben wir untersucht, wie lange wir Anwendungen den Zugriff auf die GPU entziehen müssen und wie sehr GPGPU- Anwendungen mit verschiedenen Zugriffsmustern durch das auslagern ihrer Daten in den SYSRAM ausgebremst werden.

Der Rest dieser Arbeit ist wie folgt aufgebaut: im folgenden Kapitel 2 beschäftigen wir uns mit den technischen Hintergründen zur Hardware und Grafiktreibern. Im 3. Kapitel stellen wir die Ansätze zur Speicherverwaltung für GPGPUs

aus [39, 41, 59] vor und vergleichen sie mit unserem Entwurf, den wir im Kapitel 4 detailliert vorstellen. Auf einige Details der Implementierung gehen wir im 5. Kapitel ein. Die Ergebnisse unserer Benchmarks und zugehörige Analysen präsentieren wir im 6. Kapitel. Wir schließen mit einer Zusammenfassung und einigen Schlussfolgerungen in Kapitel 7.

Kapitel 2

Hintergrund

Wir beschäftigen uns in diesem Kapitel mit dem Zusammenspiel von Soft- und Hardware in der GPGPU Technologie. Zunächst nehmen wir die Sichtweise des Anwendungsprogrammierers ein. Im zweiten Abschnitt stellen wir dem die Fähigkeiten der von uns gewählten Hardwareplattform gegenüber, um anschließend das Bindeglied – den Treiber – zu beschreiben. An dieser Stelle zeigen wir die Unterschiede zwischen der Speicherverwaltung für Grafik- und für GPGPU- Anwendungen auf.

2.1 GPGPU

CPUs wurden für die sequentielle Ausführung weit verzweigter Programme mit umfangreichen Befehlssätzen und geringer Latenz entwickelt. Dem gegenüber wurden GPUs von Anfang an als spezialisierte, hochgradig parallelverarbeitende Rechenwerke entworfen [42]. Durch die hohe Datenparallelität in der Bildverarbeitung können Latenzen leicht verborgen werden, weshalb Caches von geringerer aber die Speicherbandbreite von hoher Bedeutung ist. Ihre Architektur ist auf einfacher strukurierte Programme optimiert, was den Kontrollfluss vereinfacht. Wegen der kleineren Caches und des einfacheren Kontrollflusses, können die Hersteller einen größeren Anteil der Chipfläche für Rechenwerke aufwenden [21] und folglich eine höhere theoretische Rechenleistung erzielen (vgl Tabelle 2.1).

Mit DirectX 8 wurden Shader eingeführt, die erstmals GPUs programmierbar machten [47]. Beispielsweise ist ein Pixel- Shader ist ein kleines Programm, das die GPU für jeden Pixel unabhängig auswertet. Die Auswertung erfolgt für viele Pixel parallel durch einzelne Prozessoren innerhalb der GPU.

Der Befehlssatz für Shader wurde in den letzten Jahren beträchtlich erweitert und ermöglicht heute die Implementierung einer Vielzahl von Algorithmen aus verschiedenen Aufgabenfeldern, was GPUs „allzwecktauglich“ macht. Wir spre-

	Nvidia GeForce GTX 480 [4]	Intel Core i7- 2600 [27]
Erscheinungsdatum	März 2010	Anfang 2011
Prozessortakt	1400 MHz	3400 MHz
GFLOPS (SP/DP)	1345 / 168	217,6 / 108,8
Speicherbandbreite	177 GB/s	21 GB/s
PCI-E Bandbreite	6 GB/s effektiv [54]	-

Tabelle 2.1: Vergleich der theoretischen Leistungsfähigheit von CPU und GPU

chen daher von *General Purpose GPU (GPGPU)*. Anstelle von Shader sprechen wir in Zusammenhang mit GPGPU von einem *Kernel*. Betrachten wir folgende Schleife:

```
for (int i = 0; i < n; i++) {
    y[i] = a*x[i] + y[i];
}
```

Der Schleifenrumpf hängt nicht von den Ergebnissen vorangegangener Iterationsschritte ab, weshalb die Iterationsschritte in beliebiger Reihenfolge durch je einen eigenen Thread berechnet werden können. Der Kernel ist die Funktion die den Schleifenrumpf ausführt, wobei sie als zusätzliches Argument die Laufvariable i als Thread- Index erhält.

Um den Kernel auszuführen, muss eine Anwendung den Binärcode des Kernels und alle vom Kernel adressierten Daten in den VRAM kopieren oder der GPU anderweitig verfügbar machen. Anschließend führt die Anwendung einen „Kernel Launch“- Befehl aus, dem Sie die Adresse des Kernels, die Anzahl der zu startenden Threads und die Funktionsargumente übergibt. In der Regel wartet die Anwendung anschließend bis der Kernel vollständig ausgeführt wurde und kopiert die Ergebnisse in den SYSRAM zurück. Die Notwendigkeit Daten explizit hin- und her zu kopieren ist ein verbreiteter Kritikpunkt an GPGPU- Anwendungen [33], weil das Kopieren die Programmierung erschwert und den Zeitaufwand für die Kernelausführung leicht übersteigen kann.

Die populärsten Programmierschnittstellen für die Entwicklung von GPGPU Anwendungen sind *OpenCL* [35] und das Nvidia- spezifische *CUDA* [37]. CUDA wiederum umfasst das abstraktere *Runtime API* und das direktere *Driver API*. Das Runtime API enthält eine Erweiterung von C++ („CUDA C“) die die GPGPU Programmierung vereinfachen soll, indem beispielsweise Kernel Launches ähnlich wie gewöhnliche Funktionsaufrufe geschrieben werden können. Der Nvidia Compiler `nvcc` übersetzt den CUDA C Code und generiert für die im CUDA C-Code definierten Kernel den Binärcode der jeweiligen GPU- Generation.

Gdev [41] ist eine Open- Source Laufzeitumgebung für Nvidia GPGPUs. Ihre Kernkomponente übersetzt Anweisungen aus einem eigenen API in Treiberaufru-

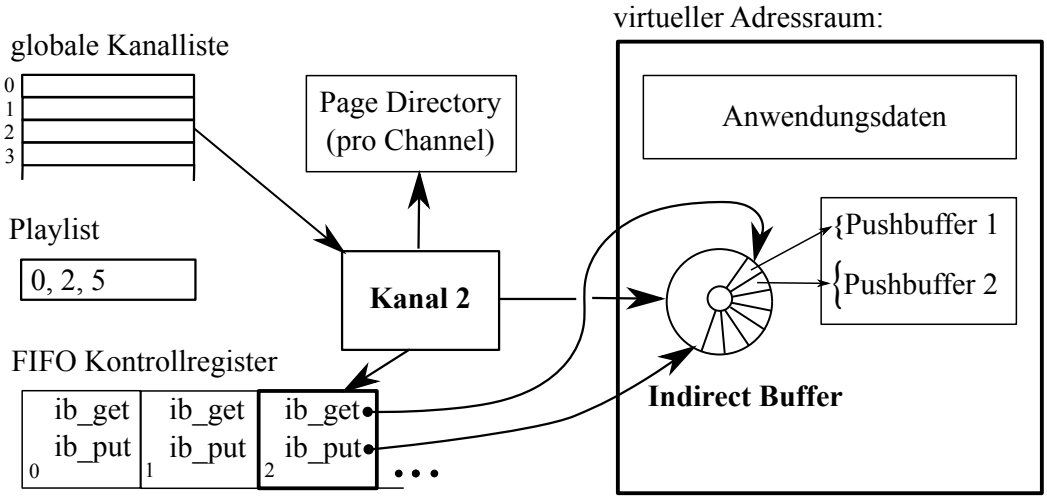


Abbildung 2.1: Übersicht über die wesentlichen Datenstrukturen. Der Anwender kann nur auf Register zugreifen, die seinen eigenen Kontext betreffen.

fe bzw. Befehle an die Grafikkarte. Gdev enthält ferner ein Frontend („uCuda“) das eine Teilmenge des CUDA Driver API auf Basis des nativen Gdev- API reimplementiert [40]. Der Kernel selbst muss aber weiterhin für die jeweilige GPU vom nvcc übersetzt werden, weshalb die Grafikkarte, sobald der Kernel läuft, den exakt gleichen Code ausführt. Im nächsten Kapitel werden wir weitere Komponenten von Gdev betrachten.

2.2 Nvidia Fermi- Architektur und Hardwareschnittstelle

In diesem Abschnitt untersuchen wir Grafikkarten der Nvidia Fermi Generation, speziell die Geforce GTX480 (GF100), im Folgenden meist als NVC0 bezeichnet [51], aus der Sicht eines Treiberentwicklers, soweit dies für das Verständnis der nachfolgenden Kapitel notwendig ist.

Wir legen dieser Beschreibung die nur unvollständige öffentlich verfügbaren Dokumentation [3], die Quelltexte der Open- Source Treiber nouveau [6] und pscnv [7] und eigenes Reverse Enginieering zugrunde.

2.2.1 Kanäle

NVC0 verwaltet eine Liste von Zeigern auf *Kanäle* (vgl. Abb 2.1). Die Position innerhalb dieser Liste bestimmt die *ID* dieses Kanals. Die *Playlist* enthält die IDs der *aktiven* Kanäle.

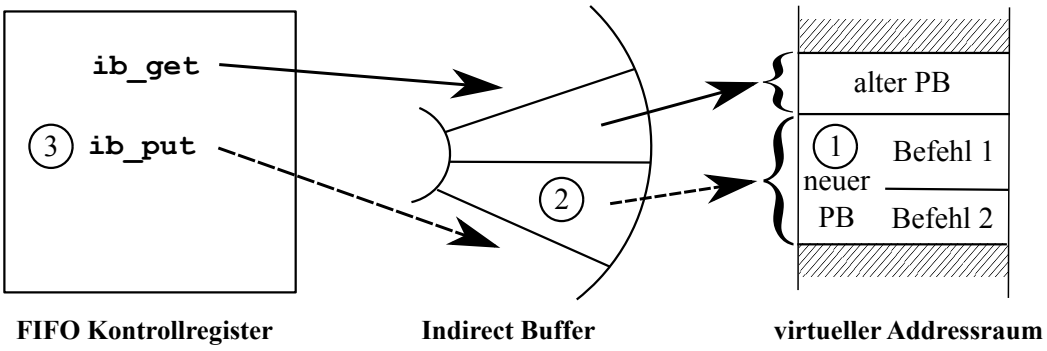


Abbildung 2.2: Übergabe einer Befehlsfolge an die GPU

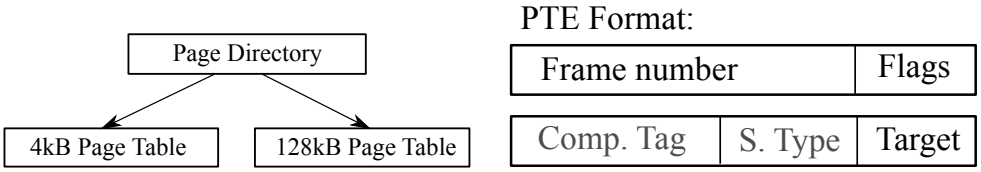
Ein Kanal ist grob mit einem Prozesskontext in Betriebssystemen vergleichbar. Eine Anwendung reserviert einen Kanal und setzt Befehle, beispielsweise Kopieroperationen oder Kernel Launches, in ihrem Kanal ab. Technisch ist ein Kanal eine Struktur von Zeigern auf Datenstrukturen die den Kontext einer Anwendung bilden.

Zu einem Kanal gehört insbesondere ein eigener virtueller Adressraum, den das Page Directory definiert auf das der Kanal zeigt. Alle Befehle die in einem Kanal ausgeführt werden arbeiten auf virtuellen Adressen in diesem Adressraum. Die Befehle selbst stehen in *Push Buffer (PB)* variabler Größe, die über Zeiger in einer zyklischen Queue, dem *Indirect Buffer (IB)*, adressiert werden. Es ist Aufgabe des Schedulers der GPU („PFIFO“) den Einträgen im IB zu folgen und die Befehle, die in den PB stehen, zur Ausführung zu bringen. Um zu erfahren, welche PB PFIFO noch verarbeiten muss, besucht PFIFO zyklisch die Pages der aktiven Kanäle im *FIFO Kontrollregister*. Jede Page im Kontrollregister enthält je einen ib_get und ib_put- Zeiger. ib_get verweist auf den nächsten Eintrag im IB, den PFIFO verarbeiten soll. PFIFO liest den PB, auf den der Eintrag im IB zeigt und verschiebt ib_get. ib_put steht hinter dem letzten gültigen Eintrag und wird von der Software gesetzt.

Um eine Folge von Befehlen in ihrem Kanal auszuführen (vgl. Abb. 2.2), muss eine Anwendung oder ein Treiber einen PB befüllen (1), die virtuelle Startaddresse und Größe des PB in den Eintrag des IB, auf den ib_put verweist, schreiben (2) und ib_put inkrementieren (3).

2.2.2 Virtueller Speicher

NVC0 arbeitet mit 40bit breiten virtuellen Adressen. Diese können sowohl auf VRAM als auch SYSRAM verweisen. Pages und Frames sind 4 kB groß. Zusätz-



(a) Pagetable Hierarchie (b) Aufbau von Page Table Entries

Abbildung 2.3: Datenstrukturen für den virtuellen Speicher

lich kennt NVC0 Hugepages von 128 kB.

Je nach Sichtweise ist die Pagetable- Hierarchie zwei oder dreistufig. Jeder Eintrag im Pagedirectory (PD) repräsentiert einen Block von 128 MB und enthält die Startadressen von bis zu zwei Pagetables (vgl. Abb. 2.3a). Die eine Pagetable enthält Einträge für 4 kB Pages, die andere Pagetable für 128 kB Pages. Bei der Auflösung von virtuellen Adressen hat letzterer Vorrang.

Der Aufbau der Page Table Entries (PTE) unterscheidet sich nicht zwischen beiden Pagetables (vgl. Abb. 2.3b). Ob sich die Frame Number auf den VRAM oder SYSRAM bezieht, legt der Wert des Feldes Target fest. Die Frame Number ist 28 Bit breit und die Pages 4 kB groß, wodurch sich ein physischer Adressraum von 40 Bit ergibt. Dieser steht auch für SYSRAM zur Verfügung, DMA Operationen können daher auf dem unteren 1 TB Hauptspeicher erfolgen. Als Target steht SYSRAM sowohl in einer Variante ohne als auch mit Cache Kohärenz zur Wahl. Gültige PTE werden durch das present Flag gekennzeichnet. Die Page, für die der PTE steht, kann mit dem readonly Flag schreibgeschützt werden. Die bei CPUs selbstverständlichen access und dirty Bits existieren nicht. Klassische Page Replacement Algorithmen sind auf diese Bits angewiesen und daher nicht auf GPUs anwendbar.

An den verbleibenden Feldern Storage Type und Compression Tag erkennen wir, dass GPU MMUs Funktionen besitzen, die bei CPUs unbekannt sind. Als Storage Type können neben der bekannten Linearen Adressierung auch verschiedene Tiling Modi [3] gewählt und zusätzlich Texturkompression [25] aktiviert werden. Für die von uns verwendeten GPGPU- Anwendungen sind diese Funktionen uninteressant, weshalb wir sie im Folgenden ignorieren.

Pagefaults

Kann die GPU eine virtuelle Adresse nicht auflösen, löst sie einen Interrupt aus und bricht die Ausführung ab. Dem Treiber stehen zur Interruptbehandlung folgende Informationen zur Verfügung:

- die virtuelle Adresse, die nicht aufgelöst werden konnte

- die Art des Zugriffs (lesend/schreibend)
- Kanal und Engine, die den Zugriff ausgelöst haben
- die Art des Fehlers (PTE nicht vorhanden, keine Schreibrechte, …)

Uns ist keine Möglichkeit bekannt, die Berechnung nach der Behandlung eines Pagefaults zuverlässig fortzusetzen um beispielsweise Demand Paging [14, S. 385] zu implementieren. Eine einfache Lösung wäre den Kanal an einem „sicheren“ Punkt neu aufzusetzen. Einen solchen Punkt kennt der Treiber allerdings im Allgemeinen nicht. Beispielsweise darf man Kernel nicht wiederholen, weil sie möglicherweise bereits Änderungen im Speicher vorgenommen haben, die bei einer erneuten Ausführung wieder gelesen werden und das Ergebnis beeinflussen. Außerdem können Kernel langwierige Berechnungen von mehreren Sekunden darstellen, die zu wiederholen zu teuer wäre.

Änderungen an den Pagetables

Die GPU verfügt über einen TLB zur effizienten Implementierung des Pagings. Werden PTE gelöscht oder geändert, muss dieser durch schreiben in ein spezielles Register geflusht werden.

Änderungen an bestehenden PTEs sind nicht möglich, solange eine Berechnung im entsprechenden Adressraum ausgeführt wird. Es ist im Allgemeinen unbekannt, wann die GPU auf eine Adresse zugreift. Folglich kann bei einem Zugriff nicht bestimmt werden, ob dieser bereits auf dem alten Pageframe stattgefunden hat oder erst auf dem neuen Pageframe erfolgen wird.

2.2.3 Zugriff auf Videospeicher

PCI Express Geräte blenden ihren lokalen Speicher als Linearen Adressbereich mittels Base Address Registers (BAR) in den physischen Speicher der CPU ein [48, S. 595f.]. Dies ermöglicht der Software mittels Memory- Mapped- IO aus dem VRAM zu lesen oder in diesen zu schreiben. Die MMIO- Register erlauben direkten Zugriff auf den physischen VRAM. Für die meisten Zugriffe der CPU auf den VRAM dienen jedoch BAR1 und BAR3. Alternativ übernimmt die GPU selbst den Datentransfer zwischen VRAM und SYSRAM.

Zugriff über MMIO- Register in BAR0

Über BAR0 sind die meisten Konfigurationsregister der GPU zuglänglich. Diese werden als MMIO- Register bezeichnet und enthalten unter anderem die globale Kanalliste, die Playlist oder die Anfangsadresse des FIFO Kontrollregisters. Ferner

befindet sich in den MMIO- Registern eine als PRAMIN bezeichnete Schnittstelle, die direkten Zugriff auf den physischen VRAM gewährt. PRAMIN ist insbesondere nützlich für die Initialisierung der Grafikkarte. In einem Register kann ein Offset gesetzt werden. Über weitere Register lässt sich anschließend der Inhalt des VRAMs zwischen Offset und Offset+16kB lesen und schrieben. Auf diese Weise sind beliebige Manipulationen am VRAM möglich. Um die Isolation zu gewährleisten, darf deshalb nur der Kernel Zugriff auf PRAMIN haben.

Zugriff über BAR1 und BAR3

BAR1 und BAR3 sind aus Sicht der GPU „gewöhnliche“ virtuelle Adressräume, die über jeweils ein Page Directory definiert werden. Zugriffe der CPU auf diese BARs leitet die GPU dementsprechend durch ihre MMU. Weil in die virtuellen Adressräume beliebige Teile des VRAM eingeblendet werden können, ist es möglich über BAR1 und BAR3 auf den gesamten VRAM zuzugreifen, obwohl die BARs meist kleiner als der VRAM sind. BAR3 verwendet der Treiber typischerweise für seine eigenen Datenstrukturen, wie Pagetables, während der Treiber über BAR1 Anwendungen Zugriff auf ihre jeweiligen Buffer im VRAM gewährt, indem er die Buffer in BAR1 und den jeweiligen Ausschnitt aus BAR1 in die Anwendung einblendet. BAR1 existiert nur einmal im System, nicht pro Anwendung. Der Treiber muss daher sicherstellen, dass Anwendungen nur den Ausschnitt aus BAR1 sehen, der zur jeweiligen Anwendung gehört.

Zugriff über Kopieroperationen innerhalb eines Channels

Wie in 2.2.1 beschrieben, werden alle Operationen innerhalb des virtuellen Adressraums des jeweiligen Kanals ausgeführt und dieser Adressraum setzt sich sowohl aus VRAM- als auch aus SYSRAM- Pages zusammen. Zugriffe auf den SYSRAM führt die Hardware als lesende bzw. schreibende DMA Operationen aus. Setzt die Anwendung einen Kopierbefehl ab, wobei die physische Adresse von Quelle oder Ziel im SYSRAM liegt, führt die Grafikkarte daher einen DMA- Transfer durch. Weil der Treiber alleinigen Zugriff auf die Pagetables hat, kann der Treiber sicherstellen, dass jede Anwendung nur ihre eigenen Daten in ihrem Adressraum sieht, egal in welchem Speicher sie liegen. Aus diesem Grund darf den Anwendungen erlaubt werden, beliebige Kopierbefehle in ihrem Adressraum abzusetzen.

2.3 Grafiktreiber

Die wesentlichen Aufgaben eines Grafiktreibers bestehen in der Initialisierung der Hardware, Setzen der Displayauflösung („Modesetting“), Übersetzung und Wei-

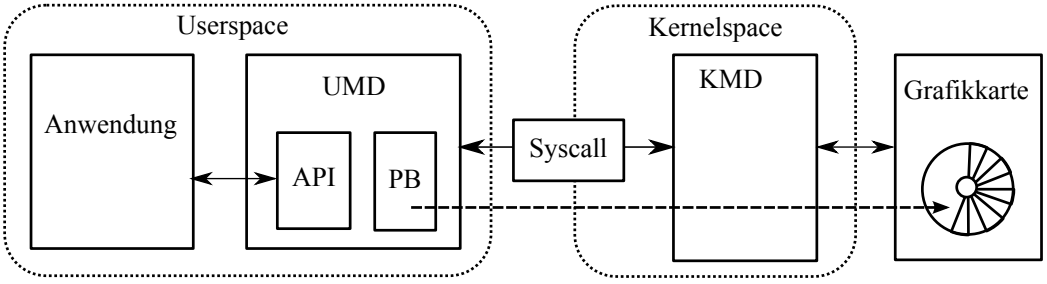


Abbildung 2.4: typischer Direct- Rendering Pfad

tergabe von Anwendungsbefehlen an die Grafikkarte, Speicherverwaltung, Abwicklung von DMA- Transfers, Isolation zwischen Anwendungen und Energiemanagement.

Wir untersuchen in dieser Arbeit neben der Speicherverwaltung auch die Befehlsübergabe, denn wir werden zeigen, dass die Art, wie Anwendung die Grafikkarte ansteuern, die Möglichkeiten des Treibers bestimmt, den Speicher zu verwalten.

2.3.1 Direct- Rendering Pfad

Eine Anwendung betreibt „Direct Rendering“, wenn sie im Rahmen ihrer Anwendungslogik mit der Grafikkarte interagiert. Beispiele für derartige Anwendungen sind 3D- Spiele, CAD- Programme, Videoplayer und GPGPU- Anwendungen. Zeigt eine Anwendung lediglich eine grafische Benutzeroberfläche, ist dies kein Direct Rendering.

In der Regel werden Anwendungen gegen ein hardwareunabhängiges API wie Direct3D, OpenGL, OpenCL oder CUDA programmiert (vgl. Abb. 2.4 und [18,24, 28]). Zumindest ein Großteil der Funktionen dieses API werden durch den User-Mode Driver (UMD) implementiert. Der UMD ist typischerweise eine dynamische gelinkte Bibliothek, die zur Laufzeit passend zur Hardware ausgewählt und in die Anwendung eingebunden wird. Funktionen des API, die von der GPU ausgeführt werden, wie z.B. Zeichenoperationen oder Kernel Launches, übersetzt der UMD in eine Befehlsfolge (Pushbuffer, PB) für die jeweilige Grafikkarte. Außerdem compiliert der UMD Shader in das native Format der Grafikkarte.

API- Funktionen, zu deren Implementierung dem Userspace die notwendigen Rechte fehlen, wie direkte Hardwarezugriffe, übersetzt der UMD in Aufrufe an den Kernel Mode Driver (KMD). Die Syscall- Schnittstelle, über die diese Aufrufe abgewickelt werden, ist typischerweise Teil des Treibermodells des Betriebssystems (WDDM/DirectX bei Windows, DRM/GEM bei Linux [56]). Der KMD

übernimmt die Kommunikation mit der Hardware, die Speicherverwaltung, Zugriffskontrolle und in manchen Fällen Scheduling und Context Switching.

Bei den meisten Treibern stellt der KMD dem UMD einen Syscall bereit, der einen Pushbuffer, welchen der UMD befüllt in einem Channel der Grafikkarte ausführt. Bei der Bearbeitung dieses Syscalls hat der KMD die Möglichkeit die Befehlsfolge zu überprüfen und zurückzuweisen, um beispielsweise ungültige Speicherzugriffe zu verhindern. Wie in Abschnitt 2.2.1 beschrieben, schreibt der KMD von modernen Grafikkarten einen Pointer in den Indirect Buffer, um die Befehle ausführen zu lassen.

2.3.2 Synchronisation

Bisher haben wir Befehlsausführung mit der Eintragung eines Befehls in eine Warteschlange, z.B. dem Indirect Buffer, gleichgesetzt. Die GPU bearbeitet diese Befehle asynchron, daher erhält die Software auf diesem Weg keine Information über den Abschluss einer Operation.

Zu diesem Zweck existieren spezielle *Fence*- Operationen [16]. Eine Fence ist ein Befehl, der mit einem anderen Befehl verknüpft wird und entweder einen bestimmten Wert an eine wählbare Speicheradresse schreibt oder einen Interrupt auslöst. Die GPU führt die Fenceoperation stets nach der mit ihr verknüpften Operation aus.

Die CPU kann aus der gewählten Speicheradresse lesen, um zu entscheiden ob die Operation bereits abgeschlossen ist, beziehungsweise auf der Adresse aktiv warten. Auf einen Interrupt kann in einer Service Routine reagiert werden.

2.3.3 Speicherverwaltung

Der Treiber verwaltet den Speicher der Grafikkarte mit Puffern wählbarer Größe, die wir als *Buffer Object (BO)* bezeichnen. Fordert der UMD vom KMD ein BO an, reserviert der KMD den notwendigen Speicher im VRAM und übergibt dem UMD einen Handle, der das BO repräsentiert. Die Anwendung benötigt ein BO, um darin beispielsweise Geometriedaten oder Texturen zu speichern. Der Speicherbedarf für die Texturen übersteigt häufig den verfügbaren VRAM [25]. Grafikkartentreiber müssen daher BOs, auf die die GPU aktuell nicht zugreift, in den SYSRAM auslagern können. Folglich benötigt der KMD die Information, welche BOs in Verwendung sind.

OpenGL und Direct3D nehmen als Argumente in ihren Funktionen stets die Handles auf diese BO, anstatt Pointer im GPU- Adressraum. Beispielsweise muss in OpenGL vor der Verwendung eines Buffers dieser mit der Funktion $\mathrm{glBind\text{-}Buffer()}$ gebunden werden [34]. Der UMD kennt daher die verwendeten BOs und gibt diese Information in Form eines $\mathrm{validate}$- Aufrufs an den KMD weiter. Ein

validate fordert den KMD auf, ein BO in einem Speicher abzulegen, auf dem die gewünschte Operation ausgeführt werden kann. Der KMD hat daher Gelegenheit, andere BOs zu swappen um Platz für das zu validierende BO zu schaffen. Der KMD fügt ferner eine Fence ein, um das Ende der Operation, das heißt den Zeitpunkt ab dem das BO nicht mehr valide sein muss, zu bestimmen [49].

Nicht nur einfache Zeichenoperationen, sondern auch Shader werden an BOs gebunden und können in ihrem Code nur auf diese zugreifen[1]. Vor dem Start eines Shaders kann der UMD also ebenfalls validate Aufrufe absetzen. Kernel adressieren dagegen, analog zu CPU- Programmen, den Speicher über Pointer. Der UMD hat daher im allgemeinen keine Möglichkeit die Menge an BOs, die bei einem Kernel Launch valide sein müssen, einzuschränken.

APIs für GPGPU überlassen der Anwendung die direkte Kontrolle über den Speicherort ihrer Buffer. CUDA sieht zum Beispiel zwei Funktionen für die Reservierung von Speicher vor: cudaMalloc allokiert VRAM und cudaHostAlloc allokiert einen DMA- Buffer im SYSRAM [21]. Das BO verbleibt an der reservierten Stelle, bis es durch die Anwendung wieder freigegeben wird. Die Speicherverwaltung durch den KMD wird auf diese Weise ausgehebelt.

Wir schließen, dass für GPGPU ein neuer Mechanismus der Speicherverwaltung benötigt wird, der ohne validate- Aufruf auskommt und sich für die Anwendung wie die direkte Kontrolle über den Speicher verhält.

2.3.4 Memory Mapping

Ein bekannter Unix- Systemcall ist mmap, mit dem eine Anwendung den Inhalt einer Datei in ihren virtuellen Adressraum einbinden kann. Treiber implementieren mmap in einer veränderten Weise, um ausgewählte Teile des physischen Adressraums in den virtuellen Adressraum der Anwendung einzublenden [17]. Auf diese Weise gewährt der KMD einer Anwendung direkten Zugriff auf ihre BOs. Dies ist von Vorteil, weil es der Anwendung erlaubt direkt auf das BO zuzugreifen, ohne für jeden Zugriff in den Kernel springen zu müssen, wodurch unnötiger Mehraufwand entsteht.

Nehmen wir an, BO 1 (vgl. Abb 2.5) liegt im VRAM. Der KMD blendet BO 1 in BAR 1 ein. BAR1 wiederum wird in einen Ausschnitt des physischen Adressraums der CPU eingeblendet, der nur für den Kernel erreichbar ist. Die Anwendung, bzw. der UMD, fordert mit dem mmap- Syscall den KMD auf, den BO 1 in den virtuellen Adressraum der Anwendung einzublenden. Der KMD überprüft die Berechtigung und blendet jene Pages aus dem Bereich von BAR1 in die Anwendung ein, die zu BO 1 gehören.

[1] mit OpenGL 5.5 wurden „bindless Buffers“ eingeführt [19], die über virtuelle Adressen im Shader adressiert werden

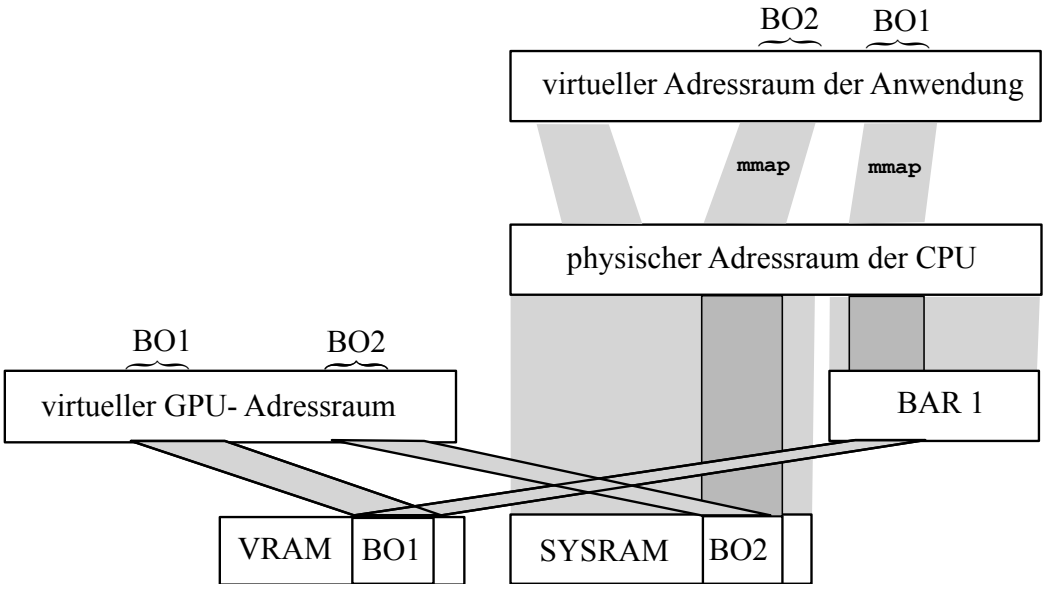


Abbildung 2.5: BO 1 liegt im VRAM, BO 2 im SYSRAM. Beide können in den virtuellen Speicher von CPU und GPU eingeblendet werden

BO 2 liegt im SYSRAM. Weil BO 2 auch in den GPU Adressraum eingeblendet wird, muss BO 2 als DMA- Buffer („Host- pinned memory“ in der CUDA-Terminologie) allokiert werden, so dass der Buffer dauerhaft auf festen physischen Adressen liegt und nicht auf die Festplatte ausgelagert werden kann. DMA- Buffer sind, genau wie BARs, zunächst nur im Kernel adressierbar. Auch in diesem Fall blendet der KMD per mmap den BO in die Anwendung ein.

2.3.5 pscnv

Der pscnv Treiber [7] wurde 2010 von der Firma Pathscale als Fork von Nouveau [6] gegründet. Nouveau ist der am weitesten verbreitete Open- Source Treiber für Nvidia Grafikkarten. Das Design von pscnv ist für GPGPU Anwendungen optimiert und konnte gegenüber Nouveau vereinfacht werden, weil pscnv sich auf neuere GPUs der Tesla- und Fermi- Generation beschränkt, die Funktionen wie Virtuellen Speicher und Hardware Context Switching bieten. Dies erlaubt die Kontrolle weitgehend an den Userspace abzugeben, ohne die Systemsicherheit zu gefährden. Der Funktionsumfang von pscnv beschränkt sich deshalb auf die Initialisierung und Verwaltung von Kanälen und Speicher.

Beispielsweise schreibt der pscnv nicht in Indirect Buffer und bietet konsequenterweise auch keinen Syscall, um Pushbuffer in einem Kanal auszuführen. Stattdessen reserviert der Userspace beim KMD einen Kanal und erhält per mmap

direkten Zugriff auf den Indirect Buffer dieses Kanals. Der Vorteil ist, dass der UMD die Grafikkarte ohne Umweg über den Kernel ansteuern kann. Der Nachteil ist, dass der KMD keine Information hat, wann und welche Operationen die Anwendung ausführt. Wie in 2.2.2 gezeigt, ist pscnv in der Lage durch die Verwaltung des virtuellen Speichers die Kanäle beziehungsweise Anwendungen trotzdem von einander zu isolieren.

Der Userspace erhält die Kontrolle welches BO in welchem Speicher allokiert und in welchen virtuellen Speicher eingeblendet ist. Notwendige DMA- Transfers zwischen diesen Speichern muss der Userspace selbst durchführen, weil pscnv, mangels der Fähigkeit Indirect Buffer zu befüllen, keine DMA- Transfers starten kann.

Die Userspace- Bibliothek von pscnv ist nur ein dünner Wrapper um die Syscalls. Insbesondere fehlen Funktionen, die häufig verwendete Befehlsfolgen, beispielsweise für einen Kernel Launch, absetzen. Gdev [41] bietet diese Funktionalität und lässt sich mit pscnv verwenden, weshalb wir es als unseren UMD betrachten.

Die Entwicklung an pscnv ist vor zwei Jahren eingeschlafen. Die letzte Version unterstützt nur Grafikkarten bis zur Fermi- Generation und erwies sich in unseren Versuchen als unfertig und fehlerhaft.

Kapitel 3

verwandte Arbeiten

In diesem Kapitel behandeln wir zwei Themenkomplexe. Auf der einen Seite geben wir einen Überblick über die Möglichkeiten Grafikkarten zu virtualisieren. Wir beschreiben die Nachteile der Frontend- Virtualisierung und motivieren auf diese Weise den Einsatz von LoGV, das den Entwurf unseres Swapping- Mechanismus geprägt hat. Auf der anderen Seite stellen wir bereits bekannte Verfahren zur Speicherverwaltung vor und diskutieren deren Leistungsfähigkeit.

3.1 Virtualisierung

Betriebssysteme stellen eine Abstraktionsschicht zwischen Hard- und Software bereit. Bei der Virtualisierung wird unterhalb des Betriebssystems eine weitere Abstraktion zur Hardware eingeführt um nicht nur mehrere Programme auf einem Betriebssystem sondern auch mehrere Betriebssysteme auf einem physischen Computersystem betreiben zu können.

Der Hypervisor[1] ist eine Anwendung im Host- Betriebssystem, die virtuelle Hardware- Komponenten simuliert und auf die echte Hardware abbildet. Aus diesen Hardwarekomponenten erzeugt der Hypervisor virtuelle Maschinen (VMs), in denen Gast- Betriebssysteme ausgeführt werden. Alternativ kann der Hypervisor Hardware direkt an den Gast durchreichen (PCI- Passthrough). In diesem Fall steht die Hardware einem Gast exklusiv zur Verfügung. Im Hinblick auf die Speicherverwaltung gibt es keinen Unterschied zum Betrieb der Grafikkarte auf einem System ohne Virtualisierung, weshalb wir diesen Fall nicht näher behandeln.

Wenn das Gast- Betriebssystem auf den Betrieb in einer virtuellen Umgebung vorbereitet ist, muss der Hypervisior keine reale Hardware emulieren. Stattdessen kann der Host dem Gast eine zweckmäßige Hypercall- Schnittstelle anbieten, die

[1] es gibt auch Hypervisor, die direkt auf der Hardware laufen

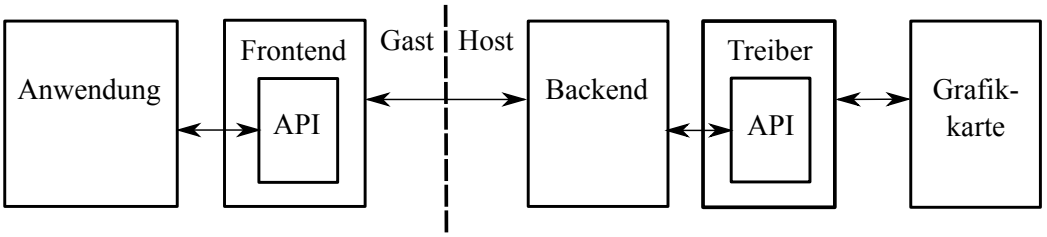


Abbildung 3.1: Bei der Frontend Virtualisierung wird der Treiber im Gast durch das Frontend ersetzt, welches über das Backend mit dem Treiber kommuniziert

der Gast mit einem paravirtualisierten Treiber anspricht [46]. Ein Hypercall ist ein Sprung vom Gast in den Hypervisor, vergleichbar einem Syscall.

3.1.1 Frontend- Virtualisierung

Bei der Frontend- Virtualisierung steuert der Treiber im Host die Grafikkarte an [22]. Im Gast stellt ein Frontend ein API für die Grafikkarte bereit. Um dieses API zu implementieren, kommuniziert das Frontend mit einem Backend im Host (vgl. Abb 3.1). Dieses Backend verhält sich gegenüber dem Host-Treiber wie eine gewöhnliche Anwendung. Die Kommunikation zwischen Frontend und Backend kann entweder über ein Netzwerkprotokoll [23, 60] oder über einen Hypercall-Mechanismus [29, 36, 53] erfolgen.

Je nach Verfahren liegt der Overhead für Frontend- Virtualisierung zwischen 10% bis 100%. Ein Grund für den Overhead sind die Befehlsfolgen, die für einen Kernel Launch abgesetzt werden müssen, denn die Befehle werden aufwendig über Frontend und Backend übermittelt. Einige der zitierten Implementierungen versuchen Befehle zusammen zu fassen, um den Aufwand zu mindern.

Ein weitere Ursache des Overheads sind langsame Kopieroperationen. Die meisten Implementierungen von Frontend- Virtualisierung unterstützen nur expliziten Kopieroperationen zwischen BOs. In der Regel werden Kopien zwischen SYSRAM des Gastes und VRAM über Front- und Backend abgewickelt, was um mindestens eine Größenordnungen langsamer als eine DMA- Operation ist [57]. Als einzige uns bekannte Frontend- Virtualisierung kann GViM [36] zumindest DMA- Buffer in die Gast- Anwendung mittels `mmap` einblenden, was eine Kopieroperation einspart. Eine vollwertige Implementierung sollte sowohl VRAM als auch SYSRAM in GPU und Gast- Anwendung einblenden können. Hierfür ist Unterstützung durch den Treiber nötig, weil nur dieser die relevanten Pagetable modifizieren kann. Den Treiber zu modifizieren widerspricht aber dem Gedanken der Frontend- Virtualisierung, denn diese sitzt zwischen Gastanwendung und UMD im Direct- Rendering Pfad.

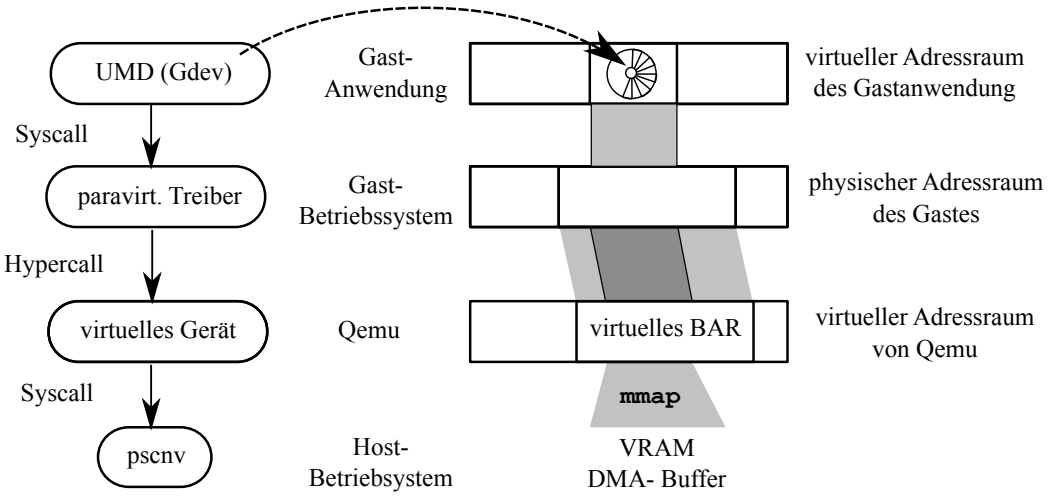


Abbildung 3.2: Architektur von LoGV: Befehle (links) werden zum Hosttreiber durchgereicht, BOs und der Indirect Buffer (rechts) in den Gast eingeblendet

Prinzipiell ist es möglich Policies zur Speicherverwaltung im Backend durchzusetzen, sofern eine Instanz des Backends alle Frontends bedient. Uns ist bislang keine Implementierung bekannt, die von dieser Möglichkeit Gebrauch macht. Normalerweise leitet das Backend Speicheranfragen direkt an den Treiber weiter. Ohne zusätzliche Maßnahmen lässt sich deshalb im Treiber nur eine Speicherverwaltung für alle VMs gemeinsam durchsetzen. Steht stattdessen für jedes Frontend ein eigenes Backend bereit, findet eine etwaige Speicherverwaltung durch den Treiber von selbst Anwendung auf die virtuellen Maschinen.

3.1.2 LoGV

LoGV [30, 31] virtualisiert Grafikkarten mit einem Overhead unter 5% aber ist dennoch in der Lage die Grafikkarte auf mehrere Gäste aufzuteilen. Der geringe Overhead ist möglich, weil LoGV den pscnv als Host- Treiber verwendet und dessen Fähigkeit nutzt BOs und insbesondere den Indirect Buffer direkt in die Anwendung einzublenden.

Die Implementierung von LoGV besteht im wesentlichen aus einem paravirtualisierten Treiber pscnv_virt (vgl Abb 3.2), der das selbe Userspace- API wie der „echte“ pscnv anbietet, und einem erweiterten Qemu. Aufrufe des UMD übersetzt pscnv_virt in Hypercalls an den modifizierten Qemu. Qemu wiederum verhält sich gegenüber pscnv wie eine Anwendung und ruft die entsprechenden Syscalls auf. Bis zu dieser Stelle verhält sich LoGV wie eine Frontend- Virtualisierung, wobei pscnv_virt das Frontend und Qemu das Backend bildet. Der Vor-

teil von LoGV ist, dass nur ein geringer Anteil der Kommunikation zwischen Gast- Anwendung und GPU über diesen Pfad abgewickelt wird. Daten und Befehle tauscht die Gastanwendung direkt mit der GPU aus, indem die Anwendung sich VRAM und DMA- Buffer in ihren Adressraum einblenden lässt. Erhält Qemu die Anweisung Speicher in die Gastanwendung einzublenden, lässt es sich den Speicher in einen Teil seines Adressraums einblenden, den es als BAR eines virtuellen Gerätes verwendet. Der paravirtualiserte Treiber verwaltet dieses virtuelle Gerät im Gast. Analog zum echten Treiber, der Teile des echten BARs in Anwendungen per mmap einblendet, blendet der paravirtualisierte Treiber den Ausschnitt aus dem virtuellen BAR in die Gast- Anwendung ein. Schreibt die Anwendung in den eingeblendeten Speicher, sind die Änderungen für die GPU sofort sichtbar. Der Mehraufwand fürs Kopieren entfällt. Lässt sich die Anwendung auf diesem Weg ihren Indirect Buffer einblenden, kann sie direkt Befehle auf der GPU absetzen, wodurch der Mehraufwand für Sys- und Hypercalls eingespart wird.

Die Aufteilung auf mehrere Gäste ergibt sich automatisch, denn Gast- Anwendungen werden durch Qemu beim pscnv in der gleichen Art wie Host- Anwendungen registiert. Folglich unterliegen Gast- Anwendungen dem normalen Hardware Context- Switching.

Außerdem unterstützt LoGV Live Migration [15]. Das bedeutet LoGV ist in der Lage eine VM während des Betriebs von einem physischen Computersystem auf ein anderes zu übertragen. Bei diesem Vorgang übernimmt die GPU auf dem Zielsystem die Aufgaben der GPU auf dem Quellsystem. Dieser Wechsel erfolgt für die Anwendung transparent. Das ist nur möglich, weil LoGV Gast-Anwendungen den Zugriff auf ihren Indirect Buffer entziehen kann, ohne das diese es bemerken. Sobald der Zugriff entzogen und alle ausstehenden Berechnungen abgeschlossen sind, kann der Zustand von der Grafikkarte gesichert und auf dem Zielsystem wiederhergestellt werden.

3.2 Speicherverwaltung

Wie wir in 2.3.3 beschrieben haben, bietet die Speicherverwaltung aktueller Grafiktreiber kein Swapping und auch Nvidias CUDA Implementierung für Windows kennt kein Swapping[2].

Swapping für GPGPU Anwendungen ist demgegenüber Gegenstand aktueller Forschung und es gibt schon mehrere Vorschläge, die wir an dieser Stelle vorstellen. Allen Varianten ist gemein, dass sie Buffer, die im VRAM keinen Platz mehr haben und aktuell nicht benötigt werden, in den SYSRAM zurückschreiben. Das nächste mal, wenn diese benötigt werden, wird der Buffer wieder in den VRAM

[2] die scheinbar anders lautende Aussage in [41] ist missverständlich formuliert.

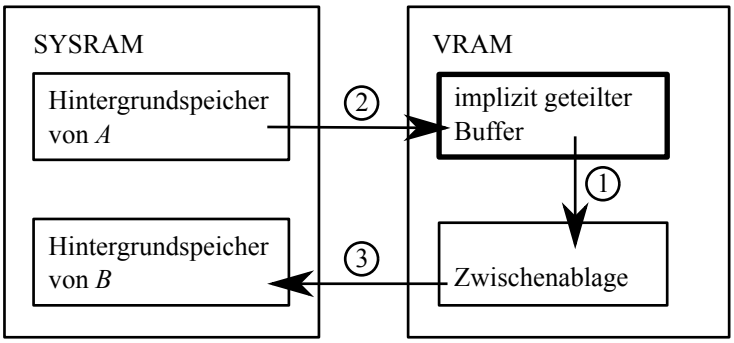


Abbildung 3.3: Austausch des Inhalts eines geteilten Buffers durch Gdev beim Wechsel von B nach A

verschoben und verdrängt mit hoher Wahrscheinlichkeit einen anderen.

Uns ist kein Swapping- Mechanismus bekannt, der direkt in den Grafiktreiber integriert ist. Die hier vorgestellten Mechanismen Gdev und GDM arbeiten mit dem Nouveau Treiber zusammen, implementieren ihr Swapping aber in einer eigenen Zwischenschicht.

Auf das mit CUDA 6 eingeführte Unified Memory [37] und GMAC [26] gehen wir nicht weiter ein, weil diese Speicherverwaltungssysteme lediglich die Konsistenz zwischen VRAM und SYSRAM sicherstellen, aber kein Swapping bieten.

3.2.1 Gdev

Wir benutzen Gdev [41] als User-Mode-Driver (UMD), aber Gdev kann auch in den Kernel integriert werden. In diesem Fall steht das Gdev-API als Syscall Schnittstelle zur Verfügung. Als UMD wird nur noch eine einfache Wrapper- Bibliothek benötigt, denn das Gdev API kennt bereits High- Level Operationen wie Kopierbefehle und Kernel Launch.

Ist Gdev in den Kernel integriert, bietet es einige weitergehende Funktionen, unter anderem einen Software- Scheduler und Shared Memory[3]. Der Software-Scheduler legt die Reihenfolge von Kernel- Ausführungen fest. Shared Memory erlaubt Anwendungen BOs mit anderen Anwendungen im System zu teilen. In diesem Fall blendet Gdev einen Buffer in die GPU Adressräume mehrerer Anwendungen ein.

Swapping ist in Gdev durch „impliziten“ Shared Memory implementiert. Kann eine VRAM- Anforderung einer Anwendung A nicht erfüllt werden, sucht Gdev nach einem „Opfer“-Buffer einer anderen Anwendung B, der mindestens die an-

[3] nicht zu verwechseln mit CUDA- Begriff „Shared Memory“.

gefordete Größe hat. Das Opfer wird dann mit A implizit geteilt. Implizites Teilen bedeutet, dass es für die beteiligten Anwendungen aussieht, als ob ihnen der Buffer exklusiv gehört. Insbesondere dürfen sie nicht die Inhalte sehen, die andere Anwendungen in den implizit geteilten Buffer schreiben.

Gdev stellt sicher, dass A und B jeweils ihre eigene Version des geteilten Buffers sehen, indem es mit seinem Software- Scheduler die Reihenfolge von Kernel-Ausführungen kontrolliert. Der Scheduler legt eine Ausführungsfolge für die Kernel von A und B fest, sodass keine Kernel der beiden Anwendungen parallel ausgeführt werden. Ist ein Kernel von A an der Reihe, nachdem zuvor ein Kernel von B ausgeführt wurde, kopiert Gdev (vgl. Abb 3.3) die implizit geteilten Buffer in eine Zwischenablage im VRAM (1), danach kopiert Gdev den Bufferinhalt von A aus dem Hintergrundspeicher im SYSRAM in die geteilten Buffer (2). Erst zu diesem Zeitpunkt führt Gdev den Kernel Launch von A aus, die Rechenwerke der GPU sind deshalb während der Kopiervorgänge (1) und (2) untätig. Asynchron zur Kernelausführung kopiert Gdev den Inhalt der Zwischenablage in den SYSRAM Hintergrundspeicher von B (3). Durch die Zwischenablage kann, sofern der Kernel von A lange genug läuft, die Latenz für den DMA- Transfer in den SYSRAM verborgen werden. Man beachte, dass Gdev bei jedem Wechsel zwischen A und B alle geteilten Buffer austauschen muss, unabhängig davon, ob sie von dem Kernel, der als nächstes ausgeführt wird, benötigt werden.

Die Art in der Gdev Swapping implementiert, geht mit einer Reihe von Einschränkungen einher: die Zwischenablage muss mindestens so groß wie der zu swappende Buffer sein und stellt eine Verschwendung von Speicherplatz dar, wenn sie nicht benötigt wird. Im Quellcode sind 128 MB vorgesehen. Bei Speicherknappheit ist Swapping nicht immer möglich, denn es muss eine andere Anwendung im System vorhanden sein, die bereits über einen Buffer verfügt, der mindestens so groß ist wie der Buffer, der allokiert werden soll. Das bedeutet insbesondere, dass Gdev nicht mit einzelnen Anwendungen, die für sich genommen mehr Speicher benötigen als verfügbar ist, umgehen kann. Des Weiteren werden zwei Anwendungen durch das implizite Teilen unnötig miteinander gekoppelt und können ab dem Zeitpunkt des Swappings nur noch wechselweise die GPU nutzen. Ein weiterer Schwachpunkt ist, dass Gdev nur ganze, möglicherweise mehrere hundert Megabyte große, Buffer auf einmal auslagern kann.

Die Leistungsfähigkeit haben die Gdev- Autoren anhand von mehreren 128 MB großen Index- Baum- Suchen getestet. Der verfügbare Speicher genügt, um bis zu drei Instanzen gleichzeitig im VRAM zu halten. Ist kein Swapping erforderlich, beträgt die Ausführungszeit pro Instanz beträgt im Mittel ca. 360 ms. Die Gdev-Autoren haben die Anzahl der Instanzen bis auf 8 erhöht, sodass ab der vierten Swapping notwendig wird, wodurch die mittlere Ausführungszeit auf ca. 500 ms steigt. Dies deutet auf einen Overhead von 39% für den Austausch der Bufferinhalte hin.

3.2.2 GDM

GDM [59] erweitert Gdev um einen überarbeiteten Swapping Mechanismus, der einige der beschriebenen Schwächen behebt. Wir beschränken uns auf die Funktionen von GDM, die für Grafikkarten ohne Unterstützung für Page Fault Handling vorgesehen sind. In diesem Fall ist die grundlegende Arbeitsweise von GDM die selbe wie von Gdev: vor jedem Kernel Launch werden alle verdrängten Buffer in den VRAM zurück kopiert, die zu dem Kontext des jeweiligen Kernels gehören, selbst wenn dieser die Buffer nicht benötigt.

Die GDM Autoren haben beobachtet, dass einige GPGPU Anwendungen ein BO („Region“ in der Sprechweise von GDM) abschnittsweise für unterschiedliche Zwecke verwenden. Weil in diesem Fall auch die Zugriffsmuster verschieden sein können, unterteilt GDM ein BO in mehrere Objekte, wenn Kopieroperationen nur einen Abschnitt des BOs überschreiben. Um die Speicherverwaltung feingranularar durchzuführen, unterteilt GDM Objekte wiederum in 4 MB große Blöcke.

GDM verzichtet auf das implizite Teilen von Buffern und die Zwischenablage aus Gdev. Stattdessen unterhält jede Anwendung eine „Staging Area“. Vor einer Kopie von SYSRAM nach VRAM legt GDM eine Copy-On-Write Kopie der Daten in der Staging Area ab. Wenn GDM Blöcke aus dem VRAM verdrängt, werden diese ebenfalls in die Staging Area zurückgeschrieben. Anders ausgedrückt nutzt GDM den VRAM als eine Art Write- Back Cache für die Staging Areas der Anwendungen.

In ihren Versuchen haben die GDM Autoren festgestellt, dass 61% der Daten von der GPU nur gelesen wurden. In diesem Fall liegt bereits eine aktuelle Kopie des Bufferinhalts in der Staging Area vor und es ist überflüssig die unveränderten Blöcke nach der Kernelausführung zurückzuschreiben. Mangels dirty- Bit in den Pagetables von GPUs berechnet GDM nach der Kernelausführung Hashwerte der Buffer auf der GPU und vergleicht diese mit älteren Hashes, um unveränderte Buffer zu erkennen.

Die Autoren von GDM geben nur relative Laufzeiten gegenüber dem Swapping Mechanismus von Gdev an. Im Mittel ist GDM um 20% schneller als Gdev. Den Extremfall stellt der nn (Nächste Nachbarn) Benchmark dar. nn führt eine kurze Berechnung auf einem großen Datensatz aus. Den großen Datensatz in den VRAM und die Ergebnis zurück in den SYSRAM zu kopieren macht 80% der Gesamtlaufzeit aus, wobei zwei Drittel der übertragenen Daten nur gelesen werden. Wir erklären uns die gegenüber Gdev um 43% verbesserte Laufzeit damit, dass GDM diese konstanten Daten nicht in den SYSRAM zurück kopieren muss.

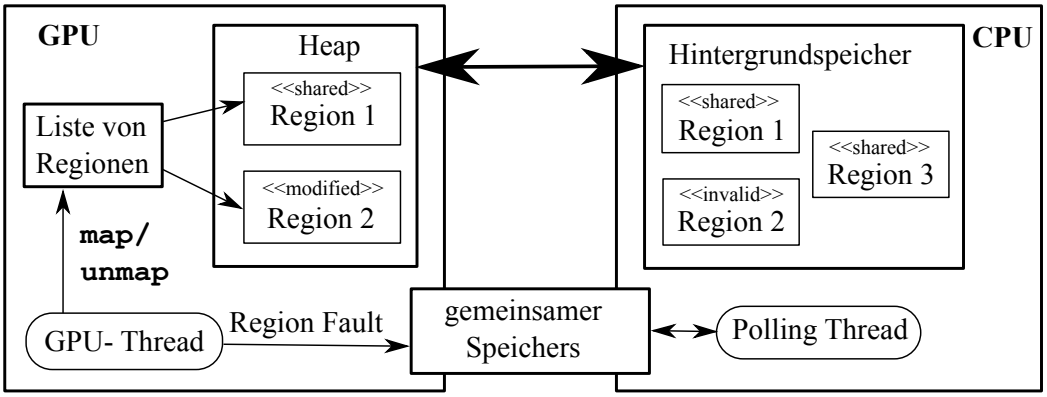


Abbildung 3.4: Arbeitsweise von RSVM: schlägt ein map() auf einer Region fehl, meldet der GPU- Thread einen Region Fault über den RPC- Mechanismus, worauf der Polling- Thread die Region in den VRAM kopiert.

3.2.3 RSVM

Als reine Userspace- Lösung unterscheidet sich RSVM [39] grundlegend von Gdev, GDM und unserem Entwurf. RSVM implementiert einen virtuellen Adressraum in Software, der sich über CPU und GPU erstreckt. Es wird als Bibliothek, die sowohl GPU als auch CPU- Code enthält, in eine Anwendung eingebunden. Wir beschränken uns in dieser Beschreibung auf die Funktionen von RSVM, die für den Swapping Mechanismus relevant sind.

RSVM realisiert die Speicherverwaltung auf ähnliche Weise wie klassische Grafiktreiber (siehe 2.3.3), jedoch können die Buffer während der Kernelausführung auf der GPU selbst mit dem map() Befehl gebunden werden. RSVM validiert daraufhin den Buffer, das heißt RSVM lädt den Buffer nötigenfalls aus dem SYS-RAM nach.

Die Anwendung muss speziell an RSVM angepasst werden. Unter anderem müssen alle Speicherallokationen durch Definition von ein oder mehreren Regionen ersetzt werden. Eine Region ähnelt einem BO, unterliegt jedoch einem Modified-Shared-Invalid Kohärenzprotokoll. Die Größe von Regionen muss vom Anwendungs- Programmierer selbst auf maximale Performance optimiert werden. Anstatt einer virtuellen Adresse erhält die Anwendung nur einen Handle („Region ID“) für die Region.

Um im Kernel auf den Inhalt einer Region zuzugreifen, muss zunächst ein map() auf der jeweiligen Region ID erfolgen, um diese für lesenden oder schreibenden Zugriff zu binden. map() ist eine von RSVM bereitgestellte GPU- Funktion, die in der Liste von Regionen (vgl Abb 3.4) nach der gewünschten ID sucht. Ist diese bereits im VRAM, genauer in dem von RSVM verwalteten Heap, gibt map()

einen Zeiger auf die Region zurück. Falls die Region noch nicht im Heap liegt, schreibt map() einen Region Fault in den gemeinsamen Speicher. Wenn Schreibzugriff angefordert wurde setzt map(), unabhängig von der Frage, ob die Region schon im VRAM liegt, den Zustand der Region auf «modified».

Der gemeinsame Speicher ist sowohl in GPU als auch CPU- Adressraum eingeblendet. Ein Polling Thread auf der CPU liest regelmäßig aus dem gemeinsamen Speicher. Im Falle eines Region Faults stößt der Thread eine Kopieroperation an, die die Region in den Heap kopiert. Der GPU- Thread, der map() aufgerufen hat, muss in diesem Fall auf den Abschluss der Kopieroperation warten.

Die Regionen werden per Reference Counting verwaltet, daher muss jeder GPU- Thread durch eine unmap() Operation anzeigen, wann die Region wieder freigegeben werden kann. Bei Speicherknappheit gibt RSVM Regionen im «shared» Zustand und Reference Count 0 wieder frei. Sind alle ungenutzten Regionen «modified», wählt der Polling- Thread mit einem Not- Frequently- Used Verfahren eine unbenutzte Region aus und kopiert sie in den Hintergrundspeicher zurück. Anschließend wird der Zustand der Region auf «shared» zurückgesetzt, so dass sie freigegeben werden kann.

RSVM verursacht Overhead selbst wenn die Probleminstanz vollständig in den Speicher passt. Im Falle einer Matrixmultiplikation beträgt der Overhead 21% und steigt um weitere 10%, wenn der Swapping Mechanismus aktiv werden muss. Die RSVM- Autoren begründen den Overhead hauptsächlich mit dem zusätzlichen Register- Bedarf für den GPU- Code. Register Spilling kann auf GPUs teuer sein [45]. Außerdem reserviert RSVM einen Teil des wertvollen Shared Memory für seinen „Software TLB“, der die map() Operation beschleunigen soll.

Wir vermuten, dass RSVM ungünstiges Verhalten aufweist, wenn mehrere Anwendungen parallel auf der GPU laufen. Zum einen startet RSVM zunächst den Kernel und kopiert ausgelagerte Regionen beim ersten map() zurück. Die betroffenen Kernel- Threads müssen warten bis eine komplette Region in den VRAM kopiert wurde und können währenddessen mangels Preemption die GPU nicht abgeben. Gdev und GDM kopieren dagegen alle Daten im Voraus, sodass die Kernel in einem Rutsch durchlaufen. Zum anderen allokiert RSVM bei der Initialisierung einen großen Buffer um darin seinen Heap anzulegen. Die Größe dieses Heaps bestimmt, ab welchem Zeitpunkt Swapping beginnt. Um mehrere RSVM- basierte Anwendungen parallel zu betreiben, darf die Summe der Heaps die Gesamtkapazität nicht übersteigen. Swapping- Mechanismen, die im Kernel implementiert sind, können dagegen den Speicher über Anwendungsgrenzen hinweg verwalten.

Kapitel 4

Entwurf

GPGPU- Anwendungen reservieren explizit Speicher im VRAM, das heißt die Speicherallokation schlägt fehl, wenn im VRAM nicht mehr hinreichend Platz ist. Deshalb müssen GPGPU- Anwendungen auf Fehlschläge bei der Speicherallokation vorbereitet sein. Ob eine Speicherallokation fehlschlägt hängt in Systemen mit mehreren parallel laufenden GPGPU- Anwendungen und insbesondere in Cloud-Umgebungen, auf denen sich mehrere Anwender die Grafikkarte teilen, vom Speicherbedarf aller auf dem System aktiven Anwendungen ab und ist daher außerhalb der Kontrolle eines einzelnen Entwicklers. Mittels einer systemweiten Verwaltung für den VRAM wollen wir Fehlschläge bei der Speicherverwaltung vermieden, sodass lediglich die Laufzeit und nicht das Ergebnis einer Anwendungen von der Lastsituation abhängt. Teilen sich mehrere Anwender die Grafikkarte, sollte der Speicher fair zwischen den Anwendern aufgeteilt werden. Außerdem möchten wir den Swapping- Mechanismus in Kombination mit LoGV [31] einsetzen können, um die für das Cloud Szenario wichtige Virtualisierung mit geringem Overhead zu ermöglichen. Bestehende Ansätze zur Speicherverwaltung eignen sich hierfür nicht, denn sie sind darauf angewiesen das Scheduling der GPU kontrollieren zu können.

Wir stellen in diesem Kapitel einen Swapping- Mechanismus für VRAM vor, der bei Speicherknappheit Daten auf eine Weise in den SYSRAM auslagert, die es der GPU erlaubt, weiterhin auf diese Daten zuzugreifen, und die keine Ansprüche an des Scheduling stellt.

Im Abschnitt 4.1 präzisieren wir zunächst die Anforderungen an unseren Entwurf. In Abschnitt 4.2 geben wir einen Überblick über das Zusammenspiel der einzelnen Komponenten. Auf diese Komponenten und wie sie geeignet sind unsere Anforderungen zu erfüllen, gehen wir in den nachfolgenden Abschnitten näher ein. Wir erörtern in Abschnitt 4.3 das Für und Wider unseres Konzepts GPU-Swapping durch Einblenden von SYSRAM zu realisieren. In Abschnitt 4.4 stellen wir den Mechanismus und schließlich in Abschnitt 4.5 die Policy unseres Verfah-

rens vor.

4.1 Anforderungen

Wir analysieren unsere Anforderungen im Rahmen eines Cloud- Dienstes. Grundsätzlich kann man die selben Folgerungen auch für sonstige Mehrbenutzerumgebungen herleiten. In diesem Fall ist der im Folgenden verwendete Begriff „Kunde“ durch „Benutzer“ beziehungsweise „Anwendung“ zu ersetzen. Deshalb möchten wir uns nicht künstlich auf virtuelle Maschinen beschränken, sondern vielmehr das Swapping für jede Anwendung auf dem System, egal, ob sie direkt im Host oder in einem Gast ausgeführt wird, bereitstellen. Wir definieren unsere Anforderungen im Hinblick auf Fairness, Leistung und Transparenz.

Fairness

Jeder Kunde soll durch die Aufteilung des VRAM möglichst wenig beeinflusst und bei Engpässen fair behandelt werden. Zumindest soll dem Kunden stets sein fairer Anteil am VRAM zur Verfügung stehen. Mittels dynamischer Speicherzuteilung soll er einen größeren Anteil am VRAM erhalten können, wenn dieser nicht durch andere Kunden ausgenutzt wird. Dies geht für den Betreiber eines Cloud- Dienstes mit minimalen Mehrkosten einher, denn er verteilt lediglich überschüssige Ressourcen. Die dynamische Speicherzuteilung macht das Angebot jedoch attraktiver für Kunden, denn diese können große Datensätze schneller verarbeiten, wenn es ihnen möglich ist, vorübergehend mehr VRAM zu beanspruchen.

Wir beziehen Fairness lediglich auf den Anteil am VRAM, den ein Kunde erhalten kann, nicht auf den Verlust an Leistung, der mit dem Swapping einhergeht. Je nach dem, ob die Leistung einer Anwendung durch den Prozessor oder den Speicher beschränkt ist, ist zu erwarten, dass verschiedene Anwendungen unterschiedlich stark vom Swapping ausgebremst werden.

Leistung

Der Einfluss unseres Swapping- Menchanismus auf die Leistung der GPU soll so gering wie möglich sein. Ein Kunde eines HPC- Dienstleisters, der seine gemietete Grafikkarte dauerhaft auslastet, hat keinen Grund sich diese mit anderen Kunden zu teilen. GPU- Virtualisierung ist daher nur wirtschaftlich sinnvoll, wenn die Kunden ihren Anteil an Rechenkapazität und Speicher über einen Großteil der Zeit nicht ausnutzen. Andernfalls ist entweder die Hardware unterdimensioniert oder die Kunden gehen verschwenderisch mit dem Speicher um. In beiden Fällen ist

die Systemleistung ohnehin eingeschränkt. Wir nehmen daher an, dass Speicherknappheit selten ist. Das Swapping soll deshalb ohne Einfluss auf die Systemleistung bleiben, solange es nicht benötigt wird. Wenn Swapping unvermeidbar ist, soll der Verlust an Durchsatz und die Zunahme an Latenz in einem gesunden Verhältnis zur Differenz von angefordertem und verfügbarem Speicher stehen.

Transparenz

Unser Swapping- Mechanismus soll sich möglichst unaufdringlich in ein bestehendes System integrieren. Einen Anteil an einer virtualisierten Grafikkarte zu mieten halten wir nur dann für attraktiv, wenn sich vorhandene Anwendungen, die für echte Grafikkarten entwickelt wurden, unverändert auf virtualisierten Grafikkarten einsetzen lassen. Der Swapping- Mechanismus darf folglich keine Änderungen am Programm erzwingen und muss deshalb so transparent wie möglich sein. Idealerweise kann ein Rechenzentrumsbetreiber seinen Kunden alle marktüblichen Schnittstellen für die Programmierung von Grafikkarten anbieten. Der Swapping- Mechanismus soll dem nicht im Wege stehen und deshalb nicht an bestimmte Funktionen von CUDA, OpenCL oder vergleichbaren Schnittstellen gebunden sein.

4.2 Überblick

Die Idee hinter unserem Swapping- Mechanismus ist, Speicher, für den nicht länger Platz im VRAM ist, in den SYSRAM zu verschieben und die virtuelle Speicherverwaltung der GPU zu nutzen, um die ausgelagerten Daten bei Bedarf von der GPU selbst aus dem SYSRAM nachladen beziehungsweise in diesen schreiben zu lassen. Zu diesem Zweck benötigen wir erstens eine Routine, die Daten zwischen VRAM und SYSRAM kopiert und gleichzeitig sicherstellt, dass die Daten in der Zwischenzeit nicht verändert werden. Zweitens müssen wir die Pagetable der GPU geeignet modifizieren, sodass Zugriffe auf den ausgelagerten Speicherbereich in Zukunft auf dem SYSRAM erfolgen. Drittens sind die Speicherbereiche, die wir aus- oder einlagern, im Sinne unseres Fairness- Kriteriums auszuwählen.

Das Swapping kann entweder auf ganzen Buffer Objects (BO) oder feingranularer auf Teilen eines BOs erfolgen. Wir führen daher den Begriff der *Swapping Option (SO)* ein. Eine SO ist ein im virtuellen Adressraum der GPU zusammenhängender Speicherblock, der für das Swapping zur Verfügung steht. Als SO kommt nur Speicher in Frage, den eine Anwendung im VRAM reserviert hat. Datenstrukturen des Treibers sind in der Regel nicht als SO geeignet. Entweder ein SO ist identisch mit einem BO, oder es ist ein Teil eines BOs. Jedes SO ist entweder vollständig im VRAM oder SYSRAM allokiert.

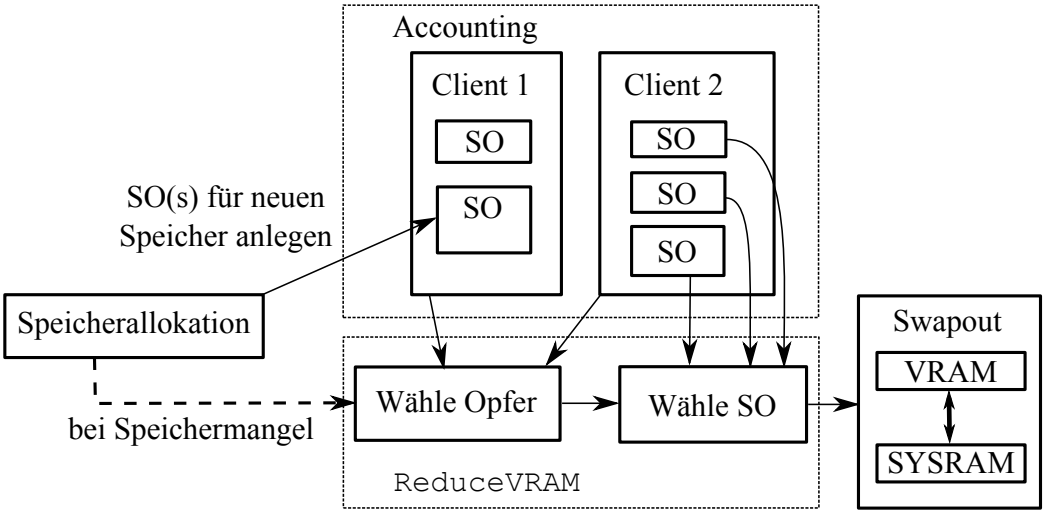


Abbildung 4.1: Arbeitsweise unseres Swapping- Mechanismus: Bei der Speicherallokation fügen wir zuerst SO(s) für den neuen Speicher an und rufen bei Bedarf ReduceVRAM auf und führen anschließend das Swapping durch

In Abbildung 4.1 ist das Zusammenspiel der drei Komponenten unseres Entwurfs dargestellt. Die erste Komponente ist das *Accounting*, welches für jeden *Client*, der die GPU verwendet, den Speicherverbrauch und die SOs dieses Clients verwaltet. Die Entscheidung welche SOs in den SYSRAM zu verschieben sind, trifft die ReduceVRAM Funktion. SwapOut führt diese Entscheidung aus.

Die Funkionen zur Speicherallokation und -freigabe verknüpfen unseren Swapping- Mechanismus mit dem Rest des Treibers. Fordert ein Client Speicher an, legt die Speicheralloktion zunächst ein oder mehrere SO für den neu zu reservierenden Speicher in der Liste für diesen Client an. Sofern noch genügend VRAM verfügbar ist, werden die SO direkt im VRAM allokiert. Andernfalls ruft die Speicherallokation die ReduceVRAM Funktion auf und aktiviert auf diese Weise den Swapping-Mechanismus. ReduceVRAM wählt unter den Clients ein oder mehrere Opfer aus und wählt unter diesen wiederum SOs für das Swapping. Anschließend lagert *SwapOut* die gewählten SOs in den SYSRAM aus.

Analog existiert eine Funktion *IncreaseVRAM*, die bei der Speicherfreigabe aufgerufen wird. *IncreaseVRAM* wählt unter den Clients, deren SOs teilweise in den SYSRAM ausgelagert wurden, ein oder mehrere *Gewinner* aus. Aus den ausgelagerten SOs der Gewinner wählt *IncreaseVRAM* wiederum einige SOs aus, die per *SwapIn* zurück in den VRAM verschoben werden.

Wir zielen mit unserem Entwurf auf eine Implementierung im Kernel auf der Basis des pscnv [7] Treibers ab. Mit einer Implementierung im Userspace können

wir unsere Anforderungen an die Transparenz nicht erfüllen, denn unser Swapping soll auf alle Anwendungen gleichermaßen zutreffen und sich deshalb nicht aushebeln lassen. Wenn wir das Swapping in den User- Mode- Driver (UMD) integrierten, ließe es sich leicht durch Linken gegen eine andere Variante des UMD umgehen.

4.3 Erweiterung von GPU- Adressräumen

Wir erweitern GPU- Adressräume mit SYSRAM, um unabhängig von der Reihenfolge, in der Kernels gestartet werden, zu sein. Anwendungen starten einen Kernel, indem sie Befehle direkt in den Indirect Buffer ihres Kanals schreiben, weshalb der Kontrollfluss nicht durch den pscnv- Treiber führt. Die Anwendung springt daher nicht in eine Kernel- Launch- Funktion, die unter der Kontrolle von pscnv steht. Die Ansätze von Gdev und GDM setzen jedoch die Existenz einer derartigen Funktion voraus, um über diese die Weiterleitung des Kernel Launches an die GPU verzögern zu können, bis sie alle BOs, auf die der Kernel zugreifen könnte, in den VRAM zurückkopiert haben. Weil wir diese Kontrolle nicht haben, müssen wir stattdessen annehmen, dass jede Anwendung die Zugriff auf den Indirect Buffer hat, zu jedem Zeitpunkt einen Kernel Launch ausführen kann und folglich bleibt uns nur sicherzustellen, dass alle BOs aller Anwendungen, zu jedem Zeitpunkt, in die GPU- Adressräume der jeweiligen Anwendungen eingeblendet sind. Das ist selbst dann möglich, wenn die Nachfrage die VRAM- Kapazität übersteigt, da moderne Grafikkarten sind in der Lage VRAM und SYSRAM gleichberechtigt in einen Adressraum einzublenden. Praktisch addiert sich die SYSRAM Kapazität zur VRAM Kapazität. Unter der Voraussetzung, dass genügend SYSRAM vorhanden ist, steht uns deshalb hinreichend Speicherkapazität zur Verfügung, um alle BOs vorzuhalten.

Die GPU adressiert den SYSRAM physisch, weshalb uns als Hintergrundspeicher nur fest im physischen Speicher allokierte Pages, das heißt DMA- Buffer, zur Verfügung stehen. Beispielsweise GDM legt dagegen den Hintergrundspeicher im virtuellen CPU- Speicher der Anwendung an [59]. Beide Varianten haben Vor- und Nachteile: für den virtuellen Speicher führen die GDM- Autoren an, dass der Speicher der einheitlichen Speicherverwaltung des Betriebssystems unterliegt. Das Betriebssystem kann den Speicherverbrauch im Hinblick auf die Gesamtsystemleistung optimieren und gegebenenfalls auf die Festplatte auslagern. Auf der anderen Seite können DMA- Operationen nicht direkt auf dem virtuellen Speicher, sondern nur über den langsameren Umweg eines Bounce Buffers erfolgen. Ein Bounce- Buffer ist ein DMA- Buffer, in den die CPU Daten aus dem virtuellen Adressraum der Anwendung kopiert, um sie für die DMA- Einheit der GPU erreichbar zu machen.

Es ist im allgemeinen eine schlechte Praxis in einem Treiber große DMA-Buffer für einen längeren Zeitraum zu reservieren [17, S. 230]. DMA- Buffer für moderne Grafikkarten zu reservieren ist allerdings vergleichen mit anderen Hardwarekomponenten unproblematisch. Die SYSRAM- Adressen in den Pagetables der GPU sind frei wählbar, im virtuellen Adressraum zusammenhängende Buffer müssen deshalb nicht physisch zusammenhängend sein. Man spricht auch von Scatter- Gather- DMA [17, S. 450]. Ferner können moderne Grafikkarten auch über 32Bit hinaus den gesamten physischen Speicher adressieren. Ein DMA- Buffer kann folglich unter den gleichen Bedingungen wie sonstiger Speicher reserviert werden.

Ob von unserem Ansatz bessere oder schlechtere Leistung als von Gdev und GDM zu erwarten sind, hängt von diversen Faktoren ab. Der grundlegende Unterschied liegt im Zeitpunkt, zu dem die ausgelagerten Daten über den verhältnismäßig langsamen PCI- Express Bus übertragen werden. Weil Zugriffe auf den SYSRAM um mindestens eine Größenordnung langsamer als Zugriffe auf den VRAM sind, betrachten wir im Folgenden nur die SYSRAM- Zugriffe.

GDM und Gdev übertragen alle ausgelagerten Daten genau einmal pro *Timeslice* der Anwendung in den VRAM. Ein Timeslice besteht in diesem Zusammenhang aus einem oder mehreren Kernels, die ausgeführt werden, bevor die Kernels einer anderen Anwendung zur Ausführung kommen. Am Ende des Timeslice kopieren GDM/Gdev die ausgelagerten Buffer wieder zurück, es sei denn GDM erkennt, dass der Buffer nicht modifiziert wurde. Bei unserem Ansatz müssen die Daten für jeden Zugriff, der nicht durch einen Cache beantwortet werden kann, erneut aus dem SYSRAM gelesen werden.

Für den häufigen Fall, dass ein Kernel einen Buffer genau einmal geordnet ließt oder schreibt, ist unser Entwurf GDM/Gdev theoretisch überlegen und bei einem geordneten lesenden und einem geordneten schreibenden Zugriff theoretisch gleichwertig zu GDM/Gdev, sofern dieser Kernel der einzige im Time Slice ist, der auf den Buffer zugreift. Ob dies praktisch gilt, hängt von der Frage ab, ob die Hardware lineare Zugriffe von Kernels auf SYSRAM genauso effizient ausführen kann wie per Kopierbefehl. Im gegebenen Fall ist es sogar möglich, dass Zugriffe von den Kernels schneller sind, als die DMA- Transfers, die GDM/Gdev durchführen, weil diese über Bounce Buffer erfolgen. Definitiv unterlegen ist unser Ansatz, wenn der Kernel auf einen Speicherbereich mehr als zwei mal zugreift. Außerdem sind wir bei ungeordneten Zugriffe im Nachteil, denn in diesem Fall muss die GPU viele kleine PCI- Express Transfers anstoßen. Außerdem ist das Verhältnis der Latenz zwischen VRAM und PCI-Express noch schlechter als das der Bandbreite [58]. GDM und Gdev übertragen auch Daten in den VRAM, die von den auszuführenden Kernels nicht gelesen werden. Neben Buffern, die für den gegebenen Kernel nicht relevant sind, trifft dies auch auf Buffer zu, in die der Kernel nur schreibt. Hier ist unser Entwurf im Vorteil, denn alle Lese- und

Schreibzugriffe erfolgen nach Bedarf.

4.4 Swapping- Mechanismus

Der Swapping- Mechanismus ist in den SwapIn und SwapOut- Prozeduren realisiert, die die Entscheidungen von *IncreaseVRAM* beziehungsweise *ReduceVRAM* umsetzen. SwapIn und SwapOut ersetzen ein SO im SYSRAM durch eines im VRAM oder umgekehrt und stellen gleichzeitig sicher, dass während des Austauschs der Inhalt des SOs nicht verändert wird.

Die Konsistenz eines SO während des Kopiervorgangs zu garantieren, ist nicht trivial, denn eine Anwendung ist in der Lage jederzeit einen Kernel zu starten, der unter Umständen Daten während des SwapIn- oder SwapOut verändert. Aus diesem Grund ist es nicht möglich, während des normalen Betriebs konsistente Kopien von Bufferinhalten anzufertigen. Dieser Sachverhalt erschwert das Verschieben eines Buffers in den Hintergrundspeicher. Nachdem beispielsweise der Inhalt des SO in den SYSRAM kopiert und die Pagetable Einträge angepasst wurden, sodass sie von nun an auf den SYSRAM verweisen, könnten ältere Daten zum Vorschein kommen, die die Anwendung beziehungsweise ein Kernel in der Zwischenzeit überschrieben haben. Die Konsequenz ist, dass die Implementierung eine Möglichkeit vorsehen muss Kanäle zu „pausieren“, das heißt für einen gewissen Zeitraum zu verhindern, dass bestehende Kernel der Anwendung laufen oder die Anwendung neue Kernel startet.

Ein weiteres Problem ist, dass pscnv der Anwendung erlaubt VRAM, den sie allokiert hat, in ihren virtuellen Adressraum einzublenden, weshalb nicht nur die GPU, sondern auch die CPU während des Swappings Daten modifizieren könnte. Die Implementierung sollte dies mit den Speicherschutzmechanismen der CPU beziehungsweise des Betriebssystems verhindern.

Bei einer normalen Allokation durch die Anwendung stößt der Treiber die folgende SwapOut Prozedur an, sobald ein SO für das Swapping ausgewählt wurde:

1. Pausiere alle Kanäle des Clients, die auf das SO zugreifen könnten
2. Reserviere einen DMA- Buffer in der Größe des SOs
3. Kopiere den Inhalt des SOs in den DMA- Buffer
4. Gebe den VRAM für das SO frei
5. Überschreibe die Page Table Einträge des SOs mit den Adressen des DMA-Buffers
6. Verschiebe das SO in die Liste für geswappte SOs des jeweiligen Clients

7. Setze die Kanäle fort

Die SwapIn Prozedur arbeitet in entgegengesetzter Weise: nachdem ein Client VRAM freigegeben hat, verschiebt SwapIn SOs zurück in den VRAM, um die GPU nicht unnötig durch langsame SYSRAM- Zugriffe auszubremesen. Die Arbeitsweise der SwapIn- Prozedur ist analog zu SwapOut, nur dass sie statt dem DMA- Buffer einen Buffer im VRAM allokiert, die DMA- Operation in die andere Richtung ausführt und anschließend den DMA- Buffer freigibt.

Durch eine einfache Optimierungen können Channel- Pausiervorgänge eingespart werden: wenn ein Kanal pausiert wurde, swappt der Treiber direkt alle gewählten SOs für diesen Kanal, anstatt den Kanal fortzusetzen, nur um ihn unmittelbar danach wieder zu pausieren.

Ein Sonderfall tritt ein, wenn das gewählte SO zu jenem Client gehört, der durch seine Speicheranfrage den Swapping- Mechanismus aktiviert hat. Wir unterscheiden zwei Unterfälle: zum einen ist es möglich, dass das gewählte SO selbst zum neu zu reservierenden Speicher gehört, weil bei der Speicherallokation die SOs für den zu reservierenden Speicher dem Client hinzugefügt werden, bevor die Swapping- Prozedur aktiviert wird. In diesem Fall kommt der Treiber ohne das Pausieren des Kanals und Reservieren eines VRAM- Buffers aus, sondern legt stattdessen direkt den DMA- Buffer an. Schwieriger zu behandeln ist der Fall, dass das gewählte SO zwar dem Client gehört, der die Speicheranfrage gestellt hat, aber bereits reserviert ist: in diesem Fall ist es notwendig den Kanal dennoch zu pausieren. Weil ein Kernel Launch eine asynchrone Operation ist, ist es prinzipiell möglich, dass noch ein Kernel des Clients läuft, während die Anwendung bereits weiteren Speicher anfordert, wodurch Speicher, auf den der Kernel potentiell zugreifen kann, verdrängt wird.

4.5 Speicherzuteilung

Die Aufgabe der Speicherzuteilung ist, unter dem Aspekt der Fairness, zu entscheiden, welche SO in den SYSRAM zu verschieben oder umgekehrt in den VRAM zurück zu kopieren sind. Die Speicherzuteilung besteht aus auf der einen Seite aus dem *Accounting* und auf der anderen aus der $\mathrm{ReduceVRAM}$ beziehungsweise $\mathrm{IncreaseVRAM}$ Funktion.

Aus Sicht des Treibers ist jede Anwendung ein *Client* und das Accounting erfolgt auf diesen Clients. Speicherallokationen und -freigaben ordnet der Treiber aufgrund der jeweiligen Process- ID einem Client zu. Die Process- ID ist als Domäne besser geeignet, als der Kanal oder GPU- Adressraum, denn prinzipiell kann eine Anwendung mehrere Kanäle und Adressräume beanspruchen. Ein Beispiel dafür ist Qemu selbst. Weil es alle Anfragen von Anwendungen innerhalb seiner VM an den Treiber weitergibt, kann es mehrere Kanäle öffnen. Diese

Kanäle werden alle unter der Process- ID von Qemu geöffnet, sodass sich alle Anwendungen innerhalb der VM ihren Anteil am VRAM teilen und Fairness zwischen verschieden VMs hergestellt wird, weil verschiedene VMs in unabhängigen Qemu- Prozessen laufen. Fairness innerhalb einer VM setzen wir nicht um. Wie der Speicher innerhalb der VM einzuteilen ist, kann ohnehin nur der Betreiber der VM entscheiden.

Zu jedem Client erfassen wir mindestens den aktuellen VRAM- Speicherverbrauch. Des weiteren verfügt jeder Client über eine Liste aller SOs, die er allokiert hat, getrennt nach den SOs die normal im VRAM und den SOs, die im SYSRAM liegen. Ferner ordnen wir dem Client sämtliche Kanäle und Adressräume zu, die dieser anlegt.

Die Funktion zur Speicherallokation ruft $\mathrm{ReduceVRAM}$ auf, wenn eine Speicheranfrage nicht mehr erfüllt werden kann und übergibt $\mathrm{ReduceVRAM}$ die Differenz zwischen benötigtem und verfügbarem Speicher. $\mathrm{ReduceVRAM}$ muss daher mindestens die Anzahl Bytes, die es als Argument erhalten hat, freigeben. Zu diesem Zweck wählt $\mathrm{ReduceVRAM}$ zunächst ein oder mehrere Clients als *Opfer* aus und wählt anschließend einige SOs der Opfer, die es für den SwapOut markiert. Das Gegenstück ist die $\mathrm{IncreaseVRAM}$ Funktion, die Anzahl Bytes, welche im VRAM noch frei sind, als Argument erhält, einige Clients, von denen SOs ausgelagert wurden, als *Gewinner* auswählt und versucht auf diese den verfügbaren Restspeicher zu verteilen, indem sie einige SOs dieser Clients für den SwapIn markiert.

Die Auswahl von Opfern und Gewinnern erfolgt mit dem Ziel den Speicher im Sinne unseres Fairness- Kriteriums aufzuteilen. Praktisch bedeutet dies, den Client, der seinen Anteil am weitesten überzogen hat, als Opfer zu wählen und den Client, der den geringsten Speicheranteil hält, als Gewinner zu wählen.

Der zweite Arbeitsschritt von $\mathrm{ReduceVRAM}$ ist die Auswahl der SOs eines Opfers. Diese erfolgt mit dem Ziel, die Leistung des Opfers möglichst wenig zu beeinträchtigen. Auszuwählen sind daher SOs, die das Opfer in Zukunft voraussichtlich wenig verwendet, beispielsweise durch einen Page Replacement Algorithmus wie *Least- Recently- Used*, sofern die Hardware dies zulässt. SOs, auf die der Kernel ungeordnet zugreift, bremsen das System stärker aus, wenn sie in den SYSRAM ausgelagert werden, als SOs mit regulärem Zugriffsmuster (siehe Abschnitt 4.4). Deshalb ist es auch denkbar, bevorzugt SOs mit regulärem Zugriffsmuster auszuwählen. Für den Fall, dass das Opfer gerade jener Client ist, der die letzte Speicheranfrage gestellt hat, besteht die Möglichkeit, entweder immer oder niemals das SO, welches mit der Speicheranfrage erzeugt wurde, auszuwählen. Dieses SO immer zu wählen, hat den Vorteil, dass der Kanal nicht pausiert werden muss, weshalb die Speicherallokation sofort ausgeführt werden kann. Der Nachteil ist, dass Speicher, nachdem er reserviert wurde, wahrscheinlich in naher Zukunft verwendet wird. Deshalb kann es auch sinnvoll sein, grundsätzlich SOs

auszulagern, die früher allokiert wurden, um zumindest das zuletzt reservierte SO im VRAM vorhalten zu können. Wir wählen unter allen SOs, einschließlich dem zuletzt reservierten, zufällig eines aus.

Kapitel 5

Implementierung

Im Kapitel 4 haben wir unseren Ansatz zum Swapping von VRAM vorgestellt. Um zu demonstrieren, dass unser Entwurf technisch umsetzbar ist und um dessen Leistungsfähigkeit bewerten zu können, haben wir diesen als Prototypen implementiert. Wir gehen in diesem Kapitel auf einige Aspekte der Implementierung ein und zeigen die Beschränkungen unseres Prototypen auf. Sofern nicht ausdrücklich anders gekennzeichnet, bezeichnet „pscnv“ im Folgenden stets den von uns erweiterten Treiber.

Im Abschnitt 5.1 beschreiben wir, auf welche Weise wir den Swapping- Mechanismus in den Treiber integriert haben. Anschließend gehen in Abschnitt 5.2 näher auf die DMA- Transfers und in Abschnitt 5.3 auf unsere Implementierung der $\mathrm{ReduceVRAM}$ Funktion ein. Ausführlich behandeln wir anschließend in Abschnitt 5.4 das Pausieren von Kanälen. Zum Abschluss gehen wir noch auf einige Beschränkungen des Prototypen ein.

5.1 Integration des Swappings

Wir haben den Swapping- Mechanismus direkt in den pscnv Treiber integriert. Eine Alternative wäre gewesen, ein eigenes Modul als Zwischenschicht zu verwenden, wie dies bei Gdev der Fall ist. Durch die Implementierung direkt im Treiber stehen dessen Datenstrukturen zur Verfügung, weshalb wir Informationen nicht redundant in eigenen Datenstrukturen ablegen müssen. Außerdem vermeiden wir eine weitere Abstraktionsschicht, die sich möglicherweise negativ auf die Leistung auswirkt. Insbesondere das Pausieren von Kanälen ist eine sehr hardwarenahe Funktion und in dieser Form eng mit der Arbeitsweise von pscnv verknüpft. Es ist natürlicher, diese Funktionalität direkt in den Treiber zu integrieren.

Unsere Implementierung muss mit mehreren Anwendungen, die gleichzeitig Speicheranfragen stellen, welche Swapping erfordern, umgehen. Das bedeutet,

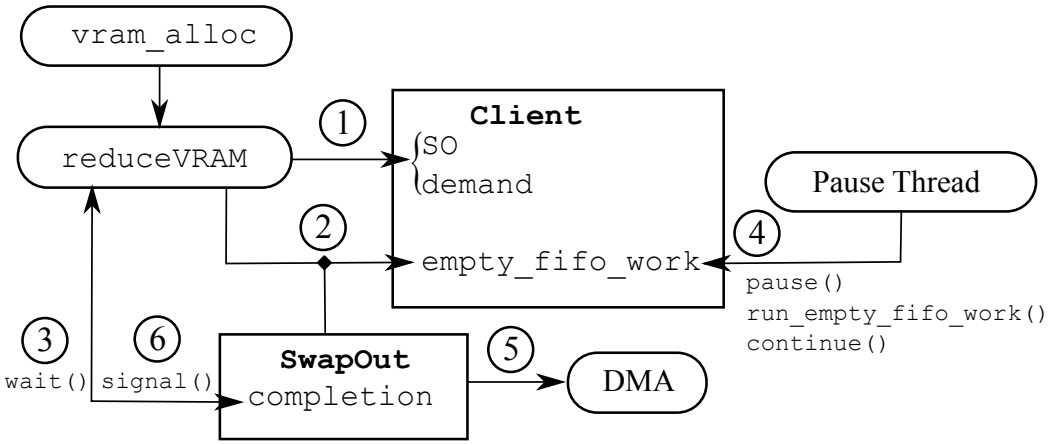


Abbildung 5.1: Zusammenspiel der Komponenten unseres Swapping- Mechanismus

dass die Liste der SOs durch einen Lock geschützt werden muss. Andernfalls besteht die Gefahr, dass ein SO zweifach ausgelagert wird. Das Problem ist, dass das Pausieren der Kanäle und die DMA- Operationen langwierige Operationen sind, während deren Ausführung der Lock nicht gehalten werden sollte. Außerdem ist es möglich, dass mehrere Anwendungen Swapping- Operationen für denselben Client anstoßen, bevor dieser Client pausiert und wieder fortgesetzt wird. In diesem Fall sollten alle Swapping- Operationen für den Client in einem Pausiervorgang abgearbeitet werden, weil jeder Pausiervorgang Overhead ist. Beide Probleme behandeln wir, indem wir die Ausführungsfäden für die Auswahl der auszulagernden SOs von der Ausführung des Swappings durch eine Workqueue entkoppeln und in unterschiedlichen Threads ausführen.

Wir haben bereits in Abschnitt 4.2 die ReduceVRAM- und IncreaseVRAM-Funktionen von SwapIn und SwapOut getrennt. Im Prototypen werden ReduceVRAM- und IncreaseVRAM von dem Kernel- Thread, der zu der Anwendung gehört, die die Speicheranfrage gestellt hat, ausgeführt. Auf der anderen Seite existiert ein eigener „Pause- Thread“, der das Pausieren übernimmt und die SwapIn und SwapOut- Funktionen ausführt. Welche Funktionen genau aufzurufen sind, liest der Pause- Thread aus der empty_fifo_work Workqueue des jeweiligen Clients. Den Ablauf haben wir in Abbildung 5.1 dargestellt: das Swapping wird durch die Funktion vram_alloc aktiviert, die normalerweise ein neues BO im VRAM reserviert. Bei Speicherknappheit betritt reduceVRAM den Lock und wählt SOs für das Swapping aus (1). Anschließend schreibt ReduceVRAM die auszuführende SwapOut Operation in die empty_fifo_work des jeweiligen Clients (2) und verlässt den Lock. Analog schreibt IncreaseVRAM (nicht dargestellt) die SwapIn

Operation in die die empty_fifo_work. ReduceVRAM wartet anschließend auf den Abschluss aller Operationen die es in die empty_fifo_work eingetragen hat (3). IncreaseVRAM überspringt diesen Schritt, denn es gibt für die Anwendung, die Speicher freigibt keinen Grund auf den Abschluss des SwapIn zu warten.

Durch das Einfügen einer Operation in empty_fifo_work wird der Pause-Thread geweckt. Dieser stellt fest, dass die empty_fifo_work eines bestimmten Clients nicht leer ist und ruft pause() auf allen Kanälen eines Clients auf (4). Nacheinander arbeitet der Pause- Thread alle Operationen in der empty_fifo_work ab (run_empty_fifo_work()). Anschließend ruft der Pause- Thread continue() auf allen Kanälen dieses Clients auf.

In empty_fifo_work sind die einzelnen SwapIn und SwapOut- Operationen eingetragen. SwapIn reserviert einen Zielbuffer im VRAM, SwapOut reserviert einen Zielbuffer im SYSRAM. Anschließend führen beide Operationen den DMA-Transfer von dem SO, dass sie zu bearbeiten haben (Quellbuffer), in den Zielbuffer aus (5). Danach passen SwapIn/ SwapOut mithilfe der in pscnv hierfür vorgesehenen Funktionen die Pagetable der GPU an. Bei diesem Arbeitsschritt erfolgt auch der notwendige TLB- Flush in der GPU. Abschließend geben sie den Quellbuffer frei. SwapOut signaliesiert zusätzlich den Abschluss der Operation an reduceVRAM (6).

Reserviert der Treiber Speicher für eigene Datenstrukturen, wird der Swapping Mechanismus nie angestoßen. Andernfalls können Deadlocks auftreten, beispielsweise ist unter Umständen eine weitere Pagetable nötig, um sowohl den Buffer im VRAM als auch den DMA- Buffer im SYSRAM gleichzeitig einblenden zu können. Nur wenn beide Buffer im Adressraum vorhanden sind, kann der Kopiervorgang angestoßen werden, mit dem der Inhalt des SOs zwischen VRAM und SYSRAM kopiert wird. Ist das Kopieren aber nicht möglich, weil vorher Platz für den Pagetable geschaffen werden müsste, liegt ein Deadlock vor.

5.2 DMA- Befehle

In Abschnitt 2.3.5 haben wir erwähnt, dass der originale pscnv keinen Code enthält, um Befehle in einem Kanal abzusetzen. Dies ist jedoch notwendig, um DMA-Transfers ausführen zu können. Diese benötigen wir wiederum, um den Inhalt einer *Swapping Option (SO)* zwischen VRAM und SYSRAM per DMA kopieren zu können. Andernfalls müssten wir die Daten mit der CPU kopieren. Das ist ab 8 kB sehr viel langsamer als DMA [41]. Die Länge einer Kanalpause ist nach unten durch die Dauer der Kopieroperationen, die während der Pause ausgeführt werden, beschränkt. Weil GPGPU- Anwendungen häufig mehrere 100 MB große Buffer reservieren, sind schnelle DMA- Operationen notwendig, um inakzeptabel langes Pausieren von Kanälen zu vermeiden.

Wir haben deshalb Funktionen aus der Userspace- Bibliothek von pscnv und aus Gdev in den Treiber integriert, um die fehlende Funktionalität zu ergänzen. Für DMA- Operation legt pscnv bei der Initialisierung einen eigenen Kanal und Adressraum an. Die Funktion, die einen DMA- Transfer ausführt, arbeitet vollständig synchron, weshalb keine parallelen Transfers möglich sind und die GPU einen Transfer zum nächsten Zeitpunkt, an dem der DMA- Kanal im FIFO- Scheduling an der Reihe ist, ausführt. Asynchrone DMA- Transfers ermöglichen vermutlich kürzere Kanalpausen, weil der DMA- Transfer sofort beginnen kann.

5.3 Speicherzuteilung

Das Konzept zur Speicherzuteilung mittels der $\mathrm{ReduceVRAM}$ Funktion aus Abschnitt 4.5 setzen wir auf einfache Weise um: es muss stets der Client Speicher abgeben, der am meisten VRAM beansprucht und innerhalb dieses Clients wählen wir das SO per Zufall. Der zufälligen Auswahl eines SO liegt die Annahme zu Grunde, diese seien ungefähr gleich groß. Das ist jedoch nicht gegeben, da bislang das Zerlegen von BOs in mehrere SOs nicht implementiert ist. Die SOs sind daher mit den BOs identisch und diese können sich, je nach Anwendung, erheblich in der Größe unterscheiden.

Die SOs sollten, anstatt zufällig, besser derart gewählt werden, dass durch das Swapping dieser SOs die Leistung möglichst wenig beeinflusst wird. Mangels $\mathrm{accessed}$ und dirty Bits stehen jedoch keine verlässlichen Informationen über das Speicherzugriffsverhalten der GPGPU- Anwendungen zur Verfügung. Eine Optimierung kann daher nur auf Basis von Annahmen über typisches Programmverhalten erfolgen. Man kann beispielsweise annehmen, dass die GPU auf den zuletzt allokierten Buffer mit hoher Wahrscheinlichkeit zugreifen wird. Derlei Optimierungen müssen praktisch überprüft werden, weshalb wir aus Zeitgründen auf diese verzichten.

$\mathrm{ReduceVRAM}$ haben wir ähnlich wie in Algorithmus 1 implementiert. Als Argument req erhält $\mathrm{ReduceVRAM}$ die Differenz zwischen dem angeforderten und dem noch verfügbaren VRAM. Die Opferauswahl erfolgt auf Grundlage des demand eines jeden Clients. Dieser gibt an, wie viel Byte dieser Client nach dem Abschluss aller bereits geplanten Swapping- Operationen noch reserviert haben wird. Bietet kein einziger Client eine weitere Swapping Option, ist die Speicheranforderung unerfüllbar und wir brechen ab. Andernfalls wählen wir den Client mit dem höchsten demand, der noch eine SO anzubieten hat, als Opfer. Wir wählen ein zufälliges SO dieses Opfers, aktualisieren den demand des Opfers und fügen den Auftrag, dieses SO auszulagern, in die $\mathrm{empty\_fifo\_work}$ Workqueue des Opfers ein. Haben wir hinreichend Speicher zur Freigabe ausgewählt, warten wir, bis alle ausgewählten SOs durch den Pause- Thread ausgelagert wurden.

```
Procedure reduceVRAM(req)
    will_free ← 0;
    selectedSOs ← ∅;
    while req < will_free do
        victim ← Client with highest demand that has SO ≠ ∅;
        if victim = NULL then fail;
        so ← RandomPop(victim.SO);
        will_free ← will_free + so.size;
        victim.demand ← victim.demand - so.size;
        AddJob(victim.on_empty_fifo, SwapOut, so);
        selectedSOs ← selectedSOs ∪ so;
    end
    foreach so in selectedSOs do
        Wait (so.completion);
    end
end
```

Algorithmus 1 : wähle Opfer, so dass mindestens req Bytes freigegeben werden und warte bis alle SOs ausgelagert wurden

5.4 Pausieren von Kanälen

Wie wir in Abschnitt 4.4 begründet haben, benötigen wir ein Möglichkeit, den Kanal einer Anwendung zu Pausieren, das heißt die Anwendung für eine wählbare Zeit am Absetzen von weiteren Befehlen an die GPU zu hindern und sicherzustellen, dass die GPU alle ausstehenden Befehle dieses Kanals abgearbeitet hat (*Fencing*, 2.3.2). Den Kanal aus der Playlist (siehe Abschnitt2.2.1) zu entfernen genügt nicht, da dies lediglich verhindert, dass die GPU weitere Befehle der Anwendung liest. Es erlaubt aber kein Fencing, weil die Fence Operation von pscnv selbst in dem Kanal ausgeführt werden muss, den wir pausieren wollen.

Wie wir in Abschnitt 2.2.1 erläutert haben, erfolgt das Absetzen von Befehlen in drei Schritten: zunächst ist der Befehl in einen Pushbuffer (PB) zu schreiben, anschließend ein Zeiger auf diesen PB in den Indirect Buffer (IB) zu schreiben und schließlich der ib_put Zeiger im FIFO- Kontrollregister zu inkrementieren. Einzig den PB der Anwendung müssen wir nicht verändern, denn wir legen mit jedem neuen Kanal einen separaten PB für den Treiber an, in dem dieser seine eigenen Kommandos absetzen kann. Um die Fence Operation abzusetzen, muss pscnv folglich sowohl den IB, als auch die Page des jeweiligen Kanals im FIFO-Kontrollregister (CTRL) manipulieren, ohne dass dies für die Anwendung zu irgend einem Zeitpunkt sichtbar ist.

Das Verfahren, mit dem wir den Kanal pausieren, wurde bereits im Rahmen

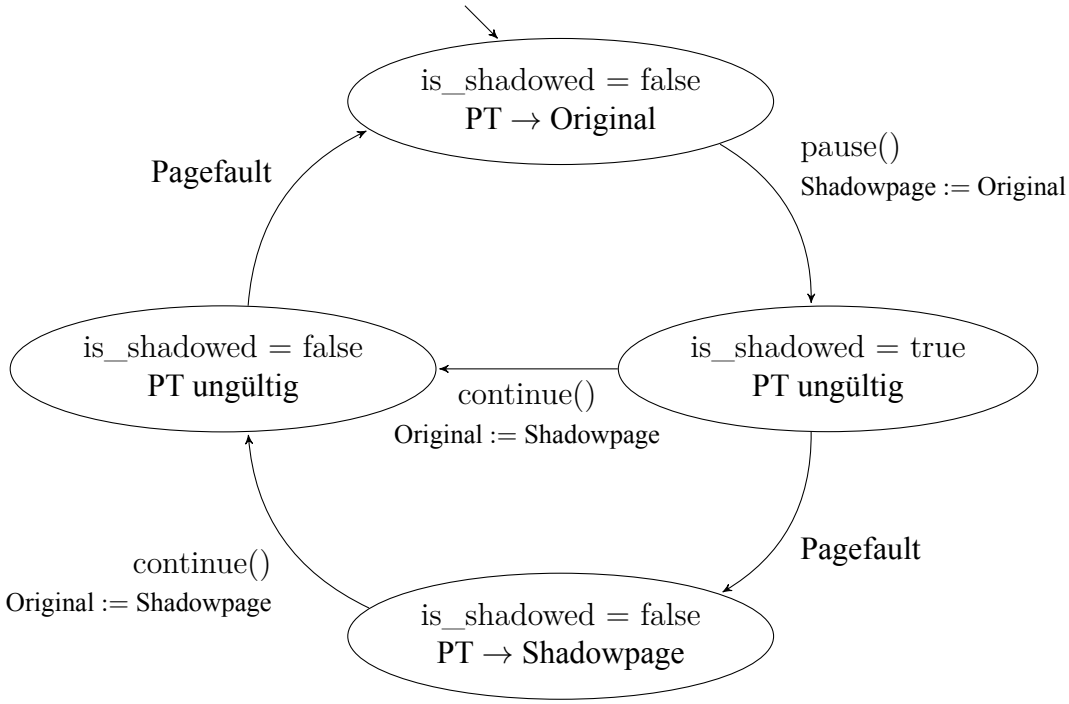


Abbildung 5.2: Shadow Paging als Zustandsautomat

von LoGV [30] erprobt. LoGV migriert eine virtuelle Maschine (VM), indem es auf der Ursprungsmaschine sowohl CTRL, als auch IB durch je eine „Shadowpage“ im SYSRAM ersetzt. Die VM schreibt die Zeiger auf alle weiteren PB in die Shadowpage des IB. In den „echten“ IB der Hardware schreibt LoGV anschließend den Pointer auf den eigenen PB mit der Fence- Operation und inkrementiert den ib_put Zeiger in CTRL. LoGV wartet auf den Abschluss der Fence- Operation, das heißt den Zeitpunkt an dem der Kanal vollständig pausiert ist, bevor es die VM anhält und und auf der Zielmaschine neu startet. In den IB auf der Zielmaschine fügt LoGV solange No-Ops ein, bis ib_put wieder an der Stelle steht, an der die VM diesen erwartet, und kopiert danach die IB-Shadowpage in den IB auf der Zielhardware zurück. Anschließend setzt LoGV den ib_put auf den Wert aus der CTRL-Shadowpage und gewährt von nun an der Anwendung wieder den direkten Zugriff auf die Hardware.

Die Arbeitsschritte sind im pscnv fast die selben, aber, anstatt die VM anzuhalten, führen wir die empty_fifo_work, also die DMA- Operationen für den SwapIn- beziehungsweise SwapOut, aus. Die Implementierung dieses Verfahrens im pscnv erfordert im Gegensatz zu LoGV eine aufwändige Synchronisation. Das *Shadowpaging*, das heißt den Austausch einer *Original* Page der Hardware durch die jeweilige *Shadowpage*, führt LoGV im Hypervisor- Code von Qemu aus. Par-

allel zum Hypervisor führt Qemu keine Anwendung in der VM aus [13], weshalb während des Shadowpaging keine Zugriffe der Anwendung auf CTRL oder IB zu erwarten sind. Im pscnv müssen wir jedoch annehmen, dass die Anwendung parallel zu unserem Swapping Code läuft.

Das Shadowpaging erfolgt in den zwei Schritten $\mathrm{pause}()$ und $\mathrm{continue}()$ (vgl. Abb. 5.2). Durch $\mathrm{pause}()$ entziehen wir der Anwendung den Zugriff auf das Original, nach einem $\mathrm{continue}()$ erhält die Anwendung den Zugriff zurück. Beide Operationen sind zur Hälfte im normalen Treibercode und zur anderen Hälfte in einem Pagefault- Handler implementiert. Um Inkonsistenzen zu vermeiden führen wir die Kopieroperationen zwischen Original und Shadowpage stets mit invalidiertem CPU- Pagetable- Entry (PTE) durch. Das heißt, sollte die Anwendung in der Zwischenzeit auf die entsprechende Page zugreifen, springt sie in den Pagefault-Handler, wartet dort per Spinlock auf den Abschluss des Kopiervorgangs zwischen Original und Shadowpage und aktualisiert danach den PTE entsprechend dem Wert der gemeinsamen Variable $\mathrm{is\_shadowed}$. Wir verzichten der Einfachheit halber auf die Möglichkeit, nach dem Kopiervorgang den PTE bereits im normalen Treibercode zu aktualisieren. Wir haben Shadowpaging einmal für den IB und einmal für CTRL implementiert, wobei die Implementierungen insofern mit einander verschränkt sind, als dass die Anwendung erst nachdem der Inhalt des IB vollständig zurückkopiert wurde, wieder Zugriff auf CTRL erhalten kann.

5.5 Einschränkungen

In diesem Abschnitt zählen wir einige Unzulänglichkeiten unseres bisherigen Prototypen auf.

Wir zerlegen bislang keine BOs in mehrere SOs. Fordern die Clients überwiegend große BOs an, ist die Folge zum einen eine ungleiche Verteilungen des VRAMs auf die Clients, weil der Client, der als Opfer ausgewählt wird, einen großen Buffer abgeben muss und daraufhin gegenüber den anderen erheblich schlechter gestellt ist, und zum anderen eine schlechte Ausnutzung des VRAM, weil deutlich mehr Kapazität ausgelagert wird, als notwendig. Unser Design ist zwar auf SOs ausgelegt, aber bislang haben wir den pscnv nicht an diese angepasst.

Die Anwendung kann sich vom pscnv nicht nur den Indirect Buffer und das FIFO- Kontrollregister, sondern auch andere Buffer mittels mmap aus dem VRAM in ihren CPU- Adressraum einblenden lassen. Wir behandeln bislang nicht den Fall, dass ein per mmap eingebundener Buffer in den SYSRAM ausgelagert wird. Pscnv sollte in diesem Fall, anstatt des nicht mehr vorhandenen VRAM- Buffers, den ausgelagerten Buffer im SYSRAM in die Anwendung einblenden. Außerdem müssen Zugriffe der Anwendung während des Kopiervorgangs zwischen dem VRAM- Buffer und dem SYSRAM behandelt werden. Die einfache Lösung be-

steht darin, die Anwendung im Page Fault Handler zu blockieren, bis die Kopie abgeschlossen ist.

Einer effektiven Verwendung mit Qemu steht bislang entgegen, dass wir SOs nur den jeweiligen Clients zuordnen und nicht präziser den virtuellen Adressräumen der GPU, in die diese eingeblendet sind. Das bedeutet, dass wir unter Umständen Kanäle unnötig pausieren. Im Fall von Qemu pausieren wir stets die gesamte VM und nicht nur die eine Anwendung, deren SO wir auslagern möchten. Wir ordnen bislang die SOs nur den Clients zu, weil eine korrekte Zuordnung aufgrund der vielen Freiheiten, die pscnv gewährt, nicht einfach ist: es ist durchaus zulässig ein BO beziehungsweise SO in mehrere virtuelle GPU- Adressräume einzublenden. Ferner ist es in pscnv möglich mehrere Kanäle in einem einzelnen virtuellen Adressraum zu verwenden. Wir müssten deshalb für jedes SO ermitteln, in welchen virtuellen Adressräumen es liegt und welche Kanäle wiederum mit diesen Adressräumen verknüpft sind.

BOs, die kleiner als 4 MB sind, werden von unserem Swapping im Moment ignoriert, um einigen Randfälle zu vermeiden. Beispielsweise verhindern wir auf diese Weise, dass Pushbuffer geswappt werden, was möglichersweise zu einem unerwarteten Leistungseinbruch führt. Kleine Buffer zu ignorieren ermöglicht einen Denial- of- Service- Angriff, der im Allokieren vieler kleine Buffer besteht. Wir sind zuversichtlich, dass durch experimentieren hierfür eine robuste Lösung gefunden werden kann.

Das Zurückschreiben von ausgelagerten SO in den VRAM ist bislang sehr unzuverlässig.

Kapitel 6

Evaluation

In diesem Kapitel zeigen wir die Möglichkeiten und Grenzen von Swapping durch Erweiterung des GPU- Adressraums mittels Hauptspeicher auf. Als Grundlage unserer Versuche dient das Swapping- Verfahren, das wir in Kapitel 4 vorgestellt haben, in der Form unseres Prototypen aus Kapitel 5. Wir überprüfen zunächst welches Potential der Einsatz von SYSRAM als Speicher für GPGPU- Anwendungen hat, das heißt welche Auswirkungen auf die Systemleistung wir erwarten müssen und stellen dem unseren Prototypen gegenüber.

Zunächst geben wir in Abschnitt 6.1 Details zum verwendeten System an und stellen anschließend in Abschnitt6.2 die von uns verwendeten Benchmarks und Methodik dar. Zunächst untersuchen wir diese Benchmarks in Abschnitt 6.3 unabhängig vom Einfluss unseres Swapping- Mechanismus auf ihre Empfindlichkeit gegenüber der Auslagerung in den Hauptspeicher und wie sich Kernel mit unterschiedlichen Zugriffsmustern verhalten, wenn sie auf den SYSRAM zugreifen. Ab Abschnitt 6.4 wenden wir uns unserem Prototypen zu. Wir demonstrieren zunächst, dass dieser in der Lage ist, mit der Überbeanspruchung des VRAM durch mehrere Anwendungen umzugehen und den Anteil an VRAM, den einzelne Anwendungen erhalten an einander anzugleichen. Anschließend untersuchen wir in Abschnitt 6.5 den Einfluss des Swappings auf die Benchmarkergebnisse im Bezug auf den Durchsatz und den Zeitaufwand für das Pausieren von Kanälen. Wir schließen dieses Kapitel in Abschnitt 6.6 mit einer Diskussion zu den Möglichkeiten GPGPU- Swapping zu optimieren, ab.

6.1 Systemkonfiguration

Wir haben unsere Benchmarks auf einem System mit Intel Core i7 – 4770 Prozessor (3,4 GHz, 4 Kerne) und 8 GB RAM durchgeführt. Wir haben keine Kerne deaktiviert. Als Betriebssystem verwenden wir Arch Linux mit dem Linux Ker-

nel 3.5.7. Diese Kernelversion haben wir wegen der Kompatibiltät mit dem pscnv Treiber gewählt. Als GPU verwenden wir die Nvidia GeForce GTX 480, denn sie ist eine der leistungsfähigsten Grafikkarten, die von pscnv unterstützt wird und Gdev ist in Kombination mit dieser Karte gut getestet. Vor den Tests setzen wir mit Hilfe des proprietären Nvidia Treibers die Taktraten auf die höchsten, die vom Hersteller vorgesehen sind: 1512 MHz Shadertakt und 1900 MHz Speichertakt. Auf der Grafikkarte sind 1536 MB VRAM installiert.

6.2 Benchmarks und Methodik

In diesem Abschnitt stellen wir die von uns verwendeten Benchmarks vor. Außerdem machen wir allgemeine Angaben zur Durchführung unserer Tests.

Um die Auswirkungen des Swappings auf die Systemleistung realistisch bewerten zu können, verwenden wir deshalb Benchmarks, die zum einen Instanzen eines gängigen Problems lösen und zum zweiten effizient implementiert sind. Letzteres ist wichtig, weil eine ineffiziente Implementierung entweder unnötig aufwendig rechnet oder vermeidbare Speicherzugriffe macht. Durch ersteres wird die Speicherbandbreite unter-, durch zweiteres überbewertet. Außerdem mussten wir uns auf Benchmarks, die mit Gdev und pscnv kompatibel sind, beschränken.

Wir geben bewusst die Laufzeit der Kernel und nicht die Zeit für einen kompletten Arbeitsgang aus Kopie in den VRAM, Kernelausführung und Rückkopie an. Auf diese Weise vermeiden wir, dass der Aufwand für die zusätzlichen Kopien von den Effekten unseres Swappings ablenkt.

Der Ausdruck „ein Benchmark kopiert Daten in den VRAM“ ist aus Sicht der Anwendung zu verstehen, das heißt die Anwendung reserviert VRAM beim Treiber und kopiert die Daten in diesen Speicher. Ob die Anwendung tatsächlich VRAM erhält und ob dieser Speicher weiterhin im VRAM verbleibt, entscheidet unser Swapping- Verfahren.

Die Laufzeiten von Kernels beziehungsweise der Zeitpunkt, an dem ein Kernel abgeschlossen ist, ermitteln wir, indem wir nach jedem Kernel Launch cuCtxSynchronize (auch cudaDeviceSynchronize) aufrufen und danach die Zeit bestimmen. cuCtxSynchronize führt eine Fence- Operation (vgl. Abschnitt 2.3.2) aus und wartet, bis alle vorangegangenen Befehle abgeschlossen sind. Das bedeutet, die Anwendung blockiert auf der CPU bis der Befehlsstrom auf der GPU abgearbeitet ist, also mindestens bis der Indirect Buffer des jeweiligen Kanals geleert ist. Nvidia empfiehlt cuCtxSynchronize nur selten zu verwenden, um Stalls in der GPU- Pipeline zu vermeiden [9]. Für die Zeitmessung von Kernels steht mit den „Events“ ein besseres Mittel zur Wahl, doch diese sind mit pscnv und Gdev nicht nutzbar.

Matrix Multiplikation (mmul)

Der **mmul** Benchmark initialisiert zwei 2048x2048 Single- Precision Floating Point Matrizen A und B mit konstanten Werten, kopiert A und B in den VRAM und startet anschließend einen Kernel, der das Produkt $C = A * B$ berechnet. Wir verwenden die Implementierung der Matrixmultiplikation, die in [21] als „Matrix Multiplication with Shared Memory“ vorgestellt wird. Im Shared Memory speichert je eine Gruppe von GPU- Threads eine Submatrix von A und B zwischen, sodass nicht jeder Thread eine ganze Zeile aus A, beziehungsweise Spalte aus B aus dem Speicher lesen muss. Die Verwendung von Shared Memory ist eine der wichtigsten Optimierungen für GPGPU- Anwendungen [9, 52]. Der Speicherbedarf für alle drei Matrizen beträgt zusammen ca. 50 MB. Wir verwenden die Matrixmultiplikation, denn sie ist einfach zu verstehen, sehr rechenintensiv und gut vergleichbar.

Laut Nvidia ist die von uns verwendete Implementierung der Matrixmultiplikation für Lehrzwecke gedacht und nicht auf maximale Leistung optimiert. Ohne Swapping erreichen wir mit dieser Implementierung 260 GFlops. Mit der CuBLAS [1] Bibliothek unter Windows erreichen wir dagegen 960 GFlops und auf der CPU mit Intels MKL [5] immerhin 300 GFlops. Allerdings ist die Leistung unserer Implementierung mit den Benchmarks aus verwandten Arbeiten vergleichbar: mit ihrer „native“ Implementierung haben die RSVM- Autoren 220 GFlops auf einer GTX 480 erzielt [39]. Der Unterschied ist möglicherweise auf eine andere Taktrate der GPU zurückzuführen. Mit der Implementierung, die bei Gdev mitgeliefert wird und die kein Shared Memory verwendet, konnten wir dagegen lediglich 68 GFlops erzielen.

Breitensuche (bfs)

Der **bfs** Benchmark reagiert aus allen Benchmarks der Rodinia Suite [52] am empfindlichsten auf die Speicherbandbreite. Wir gehen daher davon aus, dass bfs eine der realen Anwendungen ist, die am deutlichsten die Nachteile unseres Swappings aufzeigen und folglich als Worst- Case- Szenario geeignet ist. Wir verwenden eine gegenüber Rodinia abgewandelte Version des Benchmarks, die neben der Breitensuche selbst, auch die Initialisierung des Graphen weitgehend im VRAM durchführt. Dadurch belasten wir zum einen unseren Benchmark nicht mit dem zeitaufwendigen Einlesen eines vorbereiteten Graphen und können zum anderen Kernels mit verschiedenen Zugriffsmustern vergleichen. Eine Instanz von bfs enthält 1000000 Knoten und belegt etwa 276 MB Speicher. Der bfs Benchmark besteht aus vier Kernels: edge_fwd, edge_bwd und bfs_init initialisieren die Datenstruktur. Anschließend ruft der Benchmark bfs_main mehrfach auf, bis alle Knoten besucht sind. Im einzelnen zeigen die Kernels folgendes Zugriffsverhalten:

edge_fwd liest linear, schreibt mit regelmäßigen Abständen, aber nicht zusammenhängend.

edge_bwd führt willkürliche Speicherzugriffe aus.

bfs_init führt lineare Schreibzugriffe aus.

bfs_main führt daher willkürliche Speicherzugriffe aus. Dieser Kernel ähnelt dem bfs aus Rodinia, verwendet allerdings einfachere Datenstrukturen.

6.3 Voruntersuchungen

Bevor wir die Leistungsfähigkeit des Swapping- Verfahrens evaluieren, untersuchen wir zunächst, in welcher Größenordnung sich der zu erwartende Leistungsverlust bewegt. Zu diesem Zweck überprüfen wir, wie stark unsere Benchmarks ausgebremst werden, wenn sie vollständig im SYSRAM ausgeführt werden, was einem 100%-igen Swapping entspricht. Wir vergleichen mit dem Verhältnis der Bandbreite zwischen VRAM und PCI- Express Bus.

Die Bandbreite des PCI- Express Bus und des VRAM haben wir mit dem Nvidia Bandwidth Test [2] unter Windows ermittelt: Über den PCI- Express Bus werden 5,93 GB/s in Richtung VRAM und 6,13 GB/s in Richtung SYSRAM übertragen. Die effektive VRAM- Bandbreite haben wir mit 152,45 GB/s bestimmt. Das Verhältnis zwischen VRAM und PCI- Express- Bus beträgt daher ungefähr 25.

In Tabelle 6.1 ist das Verhältnis, um das die in unseren Benchmarks verwendeten Kernel durch die vollständige Auslagerung in den SYSRAM ausgebremst werden. Wir geben für die Laufzeiten jeweils Durchschnittswerte und Standardabweichung über 25 Testdurchgänge an. Selbst die Matrixmultiplikation, die normalerweise durch die Rechenleistung beschränkt ist, verlangsamt sich um Faktor 4,9. Während der Berechnung tauschen GPU und Speicher ungefähr 2 GB an Daten aus[1]. Läuft der mmul im SYSRAM, beträgt die Bandbreite daher ungefähr 6,35 GB. Dies ist, vermutlich wegen Caching- Effekten, ein wenig mehr als die PCI- Express Bandbreite, deutet dennoch klar auf diese als beschränkende Ursache hin. Die 2 GB Daten, die zwischen Speicher und GPU ausgetauscht werden, sind das 40- fache der Instanzgröße, weshalb mmul ein gutes Beispiel für einen Kernel ist, bei dem die ausgelagerten Daten besser vor der Ausführung in die GPU zurück kopiert werden.

Die Kernel edge_fwd, edge_bwd und bfs_init werden alle in ähnlichem Maße ausgebremst. Überraschend ist, dass das Zugriffsmuster der jeweiligen Kernel sich nicht in dem Verhältnis, um welches sie verlangsamt werden, widerspiegelt: bfs_init

[1] 4096 Blöcke à 1024 Threads. Jeder Block ließt 32 Zeilen und 32 Spalten.

Kernel	Laufzeit (VRAM)	Laufzeit (SYSRAM)	Verhältnis
mmul	65,914 ms ± 0,642 ms	323,165 ms ± 0,408 ms	4,90
edge_fwd	2,283 ms ± 0,175 ms	24,786 ms ± 0,380 ms	10,86
edge_bwd	6,614 ms ± 0,006 ms	72,645 ms ± 0,019 ms	10,98
bfs_init	0,046 ms ± 0,001 ms	0,610 ms ± 0,001 ms	13,26
bfs_main	0,726 ms ± 0,981 ms	14,622 ms ± 26,513 ms	20,14

Tabelle 6.1: Ausführungszeit der Kernels aus unseren Benchmarks in VRAM und SYSRAM

hat das günstigste, weil lineares, Zugriffsverhalten und wird dennoch am stärksten ausgebremst. Wir vermuten, dass wegen der kurzen Ausführungszeit des Kernels, die Hardware nicht ihre volle Effizienz erreichen kann. Außerdem scheint wahlfreier Zugriff, wie ihn edge_bwd ausführt, gleich schnell wie Zugriffe mit festen Abständen zu sein. Bei bfs_main führen die wahlfreien Zugriffe auf einem 256 MB großen Buffer dagegen zu einem erheblichen Leistungseinbruch. Die große Standardabweichung kommt zustande, weil bfs_main pro Durchlauf mehrfach gestartet wird, wobei sich die Laufzeit unterscheidet.

Wir haben ferner beobachtet, dass die Verwendung von 128 kB Pages für den VRAM gegenüber 4 kB Pages entscheidend für die Geschwindigkeit ist. In Abschnitt 2.2.2 haben wir ausgeführt, dass die GPU sowohl 4 kB als auch 128 kB Pages kennt. Mit Abstand am drastischsten ist der Unterschied bei edge_bwd ausgefallen, denn wenn wir den VRAM mit 4 kB Pages einbinden, benötigt dieser Kernel 65,1 ms, wird also um Faktor 9,8 verlangsamt. Auf edge_fwd mit Faktor 1,3 ist der Einfluss geringer und die Laufzeit von bfs_init und bfs_main wird durch die Pagesize nur minimal beeinflusst. Details zur Arbeitsweise des TLB sind uns keine bekannt, aber wir vermuten, dass edge_bwd noch genug Lokalität im Zugriffsverhalten zeigt, um von größeren Pages und der folglich geringeren TLB-Miss-Rate profitieren zu können. Auf der anderen Seite erfolgen Speicherzugriffe von bfs_main derart ungeordnet, dass sie, unabhängig von der Pagesize, fast immer zu einen TLB-Miss führen.

6.4 Aufteilung des Speichers

In diesem Abschnitt demonstrieren wir, dass unser Prototyp in der Lage ist einer Anwendung mehr Speicher zur Verfügung zu stellen, als auf der Grafikkarte vorhanden ist, und dass der Prototyp eine faire Aufteilung des Speichers erzwingen kann. Wir verwenden hierfür ein künstliches Programm „alloc“ welches kontinuierlich Speicher in 32 MB Schritten reserviert, bis alloc einen vorgegebenen

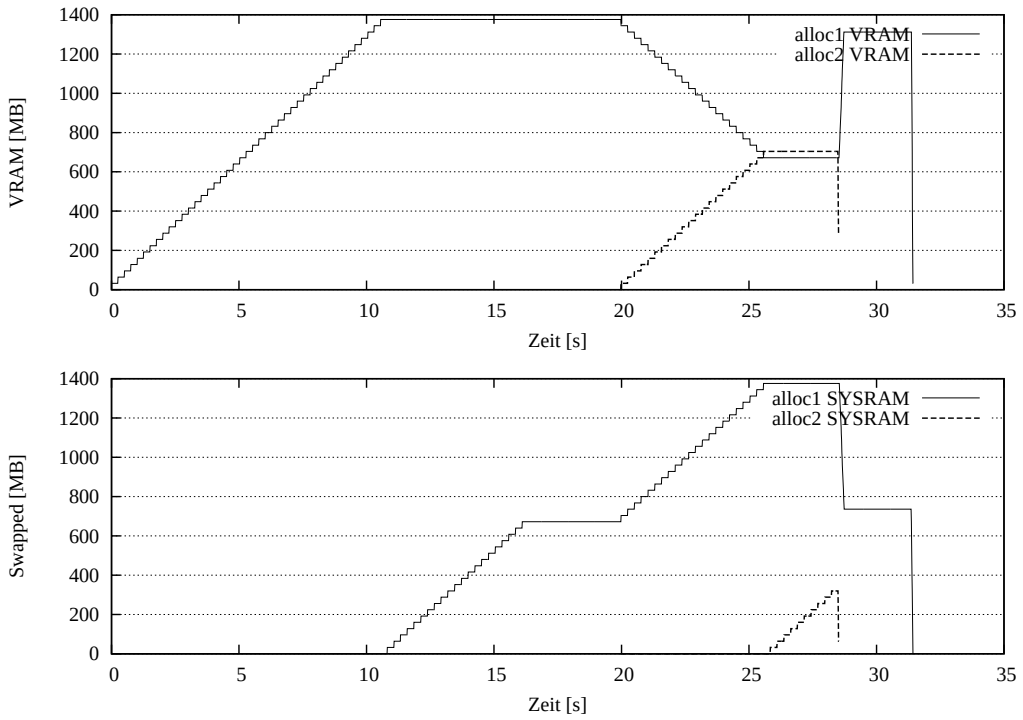


Abbildung 6.1: Aufteilung des Speichers auf zwei Prozesse

Grenzwert erreicht. Eine definierte Zeit später gibt alloc seinen gesamten Speicher auf einmal wieder frei und beendet sich. Alloc verhält sich also ähnlich einem Programm mit einem Speicherleck. Das bedeutet, solange unser Prototyp die Speicheranfragen von alloc beantworten kann, würde ein echtes Programm, trotz Speicherleck, nicht abstürzen.

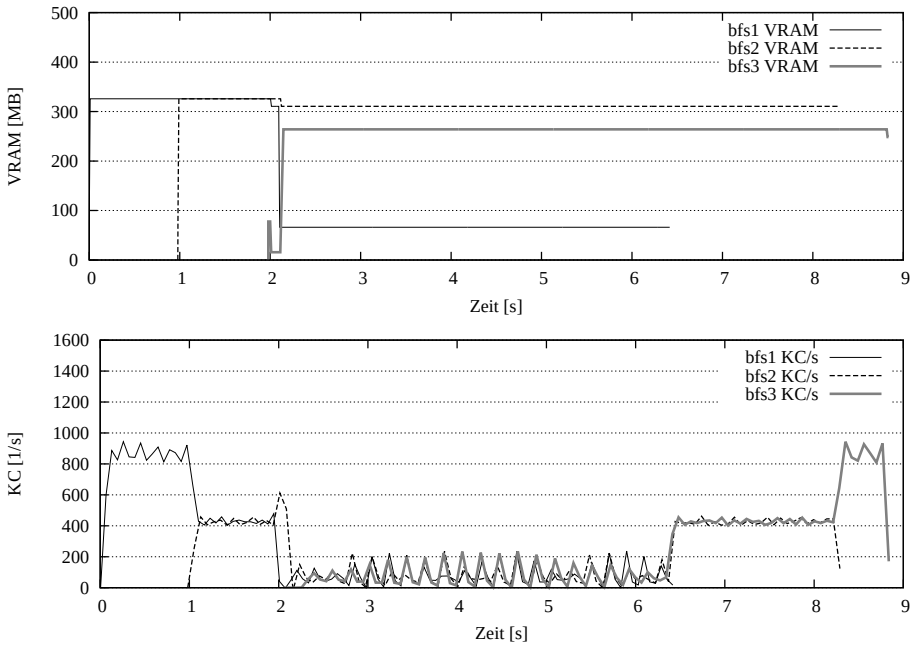
In Abbildung 6.1 ist im oberen Graph der VRAM, den zwei Instanzen von alloc erhalten haben, dargestellt. Der untere Graph gibt an, wie viel Speicherkapazität für die jeweilige Anwendung von unserem Prototypen durch SYSRAM ergänzt wird. Zunächst ist nur alloc1 aktiv und allokiert in den ersten 10 s stetig VRAM, bis alloc1 bei 1400 MB an die Kapazitätsgrenze der Grafikkarte stößt. Insgesamt enthält die Grafikkarte rund 1,5 GB Speicher, doch die verbleibenden ca. 150 MB behalten wir dem Treiber vor. Das bedeutet, dass ab diesem Zeitpunkt unser Swapping- Mechanismus aktiv werden muss. Dieser verschiebt ab jetzt zufällig ausgewählte Buffer von alloc1 in den SYSRAM, um Platz für den neu angeforderten Speicher zu schaffen. Nach Sekunde 16 fordert alloc1 keinen weiteren Speicher mehr an. Es hält zu diesem Zeitpunkt bereits rund 2 GB. Ab der 20. Sekunde startet alloc2 und beginnt ebenfalls schrittweise Speicher zu reservieren, welchen unser Prototyp bereitstellt, indem es diesen Speicher alloc1 entzieht, das heißt in

den SYSRAM verschiebt. Auf diese Weise stellt der Swapping- Mechanismus die Fairness her, denn bislang hält alloc1 weit mehr als die Hälfte des VRAM. Dieser Vorgang dauert 5 s an, dann erreichen beide alloc- Instanzen Parität. Ab diesem Zeitpunkt beginnt der Prototyp Buffer von alloc2 in den SYSRAM zu verschieben, damit alloc2 nicht mehr als die Hälfte des VRAM erhalten kann. Nachdem alloc2 in der 28. Sekunde ungefähr 1 GB Speicher reserviert hat, beendet sich alloc2 und Gibt seinen Speicher wieder frei. An dieser Stelle verteilt die IncreaseVRAM-Funktion den VRAM, den alloc2 freigegeben hat, auf die Buffer von alloc1. Deshalb werden einige Buffer von alloc1 wieder zurück in den VRAM verschoben, so dass der VRAM weiterhin gut ausgelastet bleibt. Nach Sekunde 32 beendet sich auch alloc1 und der gesamte verbleibendes Speicher wird wieder freigegeben. In der Darstellung erreichen die Linien für den Speicherverbrauch am Ende der Ausführungszeit von alloc1/ alloc2 nicht mehr die x- Achse. Die Ursache ist lediglich messtechnisch bedingt, weil die Aufzeichnung beendet wurde, bevor der Treiber die Freigabe des Speichers abschließen konnte.

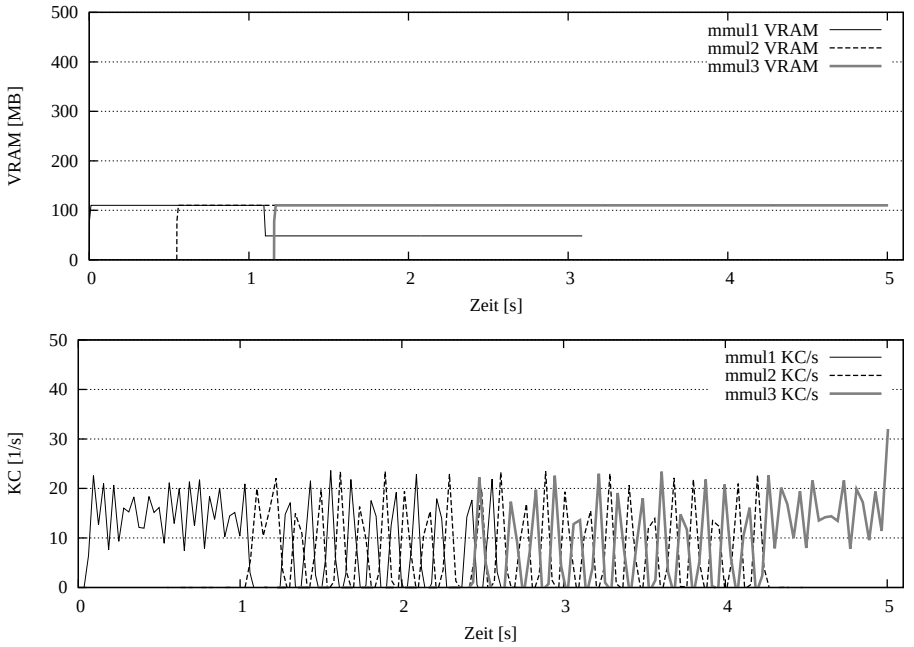
6.5 Swapping Overhead

Um das dynamische Verhalten von unserem Swapping- Mechanismus beurteilen zu können, haben wir mehrere Instanzen unserer Benchmarks parallel gestartet und im Hintergrund die Anzahl der von diesen abgeschlossenen Kernel (KC) protokoliert. Mit einer Konfigurationsvariable setzen wir ein künstliches Limit für den verfügbaren Speicher, um einen Speicherengpass zu simulieren und eine Reaktion des Swapping- Mechanismus zu provozieren. Wir verwenden ein künstliches Limit, weil wir andernfalls viele oder große Instanzen benötigen, um die GPU auszureizen. Aufgrund von Stabilitätsproblemen mit dem pscnv- Treiber ist es unter diesen Bedingungen schwierig Benchmarks durchzuführen. Die Instanzen sind in diesen Darstellungen jeweils ca. 60 MB größer als in 6.2 angegeben, weil wir an dieser Stelle den VRAM Bedarf aus Sicht des Treibers ermittelt haben, weshalb sich zum Anwendungsspeicher noch der Speicher für die Laufzeitumgebung, insbesondere Stacks, addiert.

In Abbildung 6.2a ist oben der Anteil am VRAM und unten die Anzahl an Kernel Completions (KC, geglättet) für drei Instanzen von bfs abgetragen, wobei jede Instanz 100 mal den Benchmark durchläuft. Der verfügbare Speicher ist auf 800 MB limitiert. Die Instanzen haben wir im Abstand von 1 s gestartet, um die Zuteilung des Speichers besser sichtbar zu machen. Zwischen Sekunde 0 und 1 läuft nur die Instanz bfs1, aber mit voller Leistung. Für bfs2, welche ab Sekunde 1 hinzukommt, ist ebenfalls noch genügend VRAM verfügbar und beide Prozesse teilen die Rechenleistung fair untereinander auf. Die Darstellung erweckt wegen der Glättung den Eindruck, dass bfs1 und bfs2 parallel auf der GPU laufen, doch



(a) drei bfs- Instanzen, die im Abstand von 1 s gestartet und auf 800 MB begrenzt wurden



(b) drei mmul- Instanzen, die im Abstand von 0,5 s gestartet und auf 300 MB begrenzt wurden

Abbildung 6.2: jeweils Anteil am VRAM (oberer Graph) und geglättete Anzahl abgeschossener Kernels pro Sekunde (KC/s) als Maß für die Leistung (unterer Graph)

es handelt sich lediglich um schnelle Kontextwechsel zwischen den Instanzen. Ab Sekunde 2 kommt bfs3 hinzu, welches zunächst ca. 80 MB an Buffern allokiert, bis es zum Speicherengpass kommt, weil bfs3 weitere Buffer anfordert. An dieser Stelle wählt ReduceVRAM zwei Buffer mit einer Gesamtgröße von 259 MB von bfs1, einen 15 MB großen Buffer von bfs2 und einen 61 MB großen Buffer von bfs3 aus. Wie in Abschnitt 5.5 erläutert, zerlegt unser Prototyp Buffer nicht in kleinere Einheiten, weshalb an dieser Stelle keine faire Aufteilung des Speichers gelingt. Den Buffer von bfs3 lagert SwapOut sofort aus, was sich in der Grafik als „Sprung“ erkennbar ist. Zuerst pausiert der Pause- Thread bfs1, dem zunächst der kleinere und dann der größere Buffer entzogen wird, wie man an der Treppenform im oberen Graphen erkennen kann. Für den Zeitraum, in dem bfs1 pausiert ist, ist die Leistung von bfs2 erhöht, denn bfs2 steht für einen Moment die GPU exklusiv zur Verfügung. Nachdem der Pause- Thread die Arbeit für bfs1 abgeschlossen hat, pausiert der Pause- Thread bfs2 und SwapOut entzieht diesem den 15 MB großen Buffer. Danach ist das Swapping abgeschlossen. Die Systemleistung ist ab diesem Zeitpunkt wegen bfs1, welches seit dem Swapping den Großteil seiner Daten im SYSRAM vorhält, erheblich verringert. Dieser Zustand dauert bis zum Zeitpunkt 6,4 s an. Zu diesem Zeitpunkt ist bfs1 fertig und belastet deshalb nicht länger das System. Wir haben diesen Test ohne SwapIn durchgeführt, deshalb erhalten bfs2 und bfs3 den von bfs1 freigegebenen Speicher nicht wieder zurück. Weil bfs2 und bfs3 nur geringfügig Speicher abgeben mussten, und deshalb kaum ausgebremst werden, können die bfs- Instanzen mit voller Leistung abgeschlossen werden. Das bedeutet, sie teilen sich die GPU- Leistung zu gleichen Teilen auf, bis bfs2 fertig ist, danach läuft bfs3 allein auf der GPU.

Nach dem selben Muster wie bei bfs haben wir auch den mmul- Benchmark untersucht. Das Ergebnis ist in Abbildung 6.2b dargestellt. Der Ablauf des Benchmarks unterscheidet sich wie folgt von bfs: der verfügbare Speicher ist auf 300 MB beschränkt. Jede der drei mmul Instanzen führt 25 Matrixmultiplikationen nacheinander aus, wobei die Instanzen im Abstand von 0,5 s gestartet wurden. Die meisten Unterschiede im Ergebnis gegenüber bfs lassen sich auf die längere Kernellaufzeit von mmul zurückführen. So ist im unteren Graph ersichtlich, wie sich die Kernel der verschiedenen Anwendungen abwechseln. Obwohl mmul2 bereits nach 0,5 s startet, fängt mmul2 erst bei Sekunde 1 zu rechnen an. Die Ursache ist die Zeit von 458 ms, die mmul2 benötigt, um seine 50 MB Eingabedaten in den VRAM zu kopieren. Diese 50 MB werden in drei Schritten kopiert und zwischen jedem dieser Schritte führt mmul1 eine Matrixmultiplikation aus, die ungefähr 150 ms dauert. Bei Sekunde 1 beginnt mmul2 zu rechnen und kann die GPU sogar exklusiv nutzen, weil der Pause-Thread mmul1 pausiert, um Platz für die Buffer des frisch gestarteten mmul3 zu schaffen. Die Wahl der Buffer, die mmul1 an dieser Stelle entzogen werden, soll sich als sehr glücklich erweisen: die Leistung von mmul1 ist kaum beeinträchtigt. Der Start von mmul3 wird erheblich verzögert,

Benchmark	**VRAM**			**Laufzeit**		
	total	Limit	Verhältnis	optimal	mit Swapping	Overhead
bfs	977 MB	800 MB	81,8%	5,65 s	8,84 s	56,4%
mmul	330 MB	300 MB	90,9%	4,99 s	5,00 s	0,2%
mmul	330 MB	250 MB	75,8%	4,99 s	7,99 s	60,1%

Tabelle 6.2: Vergleich zwischen der Ausführung des Benchmarks mit und ohne Begrenzung des VRAM

denn zunächst blockiert die Speicheralloktionsfunktion bis mmul1 seinen Speicher freigegeben hat und danach muss mmul3 analog zu mmul2 985 ms warten, bis dessen Buffer vollständig in den VRAM kopiert sind. Im Folgenden führen die Instanzen reihum ihre Kernel aus.

Wie groß der Overhead für das Swapping in den beiden beschriebenen Szenerien und einem weiteren Szenario ausgefallen ist, haben wir in Tabelle 6.2 zusammengefasst. Wir haben die selben Benchmark- Szenarien einmal mit Beschränkung des Videospeichers und einmal ohne Beschränkung ausgeführt und die Laufzeiten verglichen. Bereits geringfügiges Swapping geht im Falle von bfs mit einem erheblichen Overhead von rund 56% einher. Dieser Overhead ist nicht überraschend, in Abschnitt 6.3 haben wir angegeben, dass der bfs_main Kernel durch das Auslagern aller von bfs_main benötigten Buffer um Faktor 20 verlangsamt wird. Dagegen ist der Overhead im ersten mmul- Szenario vernachlässigbar. In diesem Fall hat $\mathrm{ReduceVRAM}$ den lokalen Speicher für die Stacks, welche vom mmul- Kernel nicht benutzt werden, und die Ergebnismatrix C für das Swapping ausgewählt. Dieser Fall demonstriert, dass Swapping durch Einblenden von SYSRAM bei der richtigen Wahl der Buffer sehr effektiv sein kann. Der Overhead für das Swapping von Buffern, auf die selten zugegriffen wird, ist gering. Lagert man dagegen einen Buffer aus, auf den der Kernel häufig zugreift, bricht die Leistung sehr schnell ein, wie das zweite mmul- Szenario demonstriert. In diesem Szenario haben wir lediglich den Speicher mit 250 MB gegenüber 300 MB etwas stärker begrenzt und der Overhead beträgt bereits 60%.

Neben der Verlangsamung der Berechnungen ist auch der Zeitaufwand für das Pausieren ein wesentlicher Bestandteil des Overheads. Wir unterscheiden drei Größen: *Fence- Zeit*, *DMA- Zeit* und *Pause- Zeit*. Die *Fence- Zeit* ist die Dauer zwischen dem Beginn des Pausiervorgangs eines Kanals und dem Ende der Fence-Operation, das heißt dem Zeitpunkt an dem die GPU die gesamte in diesem Kanal ausstehende Arbeit abgeschlossen hat. Die Fence- Zeit hängt deshalb hauptsächlich von den Befehlen ab, die zum Beginn der Pause in dem jeweiligen Kanal noch ausstehen, sowie von der Auslastung der GPU durch andere Kanäle. Nachdem die

Instanz	Fence- Zeit	1. DMA- Zeit	2. DMA- Zeit	Pause- Zeit
bfs1	15,28 ms	4,56 ms	62,12 ms	124,44 ms
bfs2	6,97 ms	2,90 ms	–	37,21 ms
mmul1	52,53 ms	75,56 ms	–	197,78 ms

Tabelle 6.3: Fence-, DMA-, und Pause- Zeit für das mmul- und bfs- Szenario

Fence- Operation abgeschlossen wurde, führen die SwapIn und SwapOut Prozeduren (vgl. Abschnitt 4.4) ein oder mehrere DMA- Operationen aus. Die Zeit für die Durchführung einer DMA- Operation ist die *DMA- Zeit*. Diese hängt nicht nur von der Größe des zu kopierenden Buffers, sondern ebenfalls von der Auslastung der GPU, ab. Ein Grund ist, dass die GPU eine DMA- Operation, genau wie ein Kernel Launch, nicht sofort ausführt, sondern, analog zum mmul Szenario, erst wenn der betreffende Kanal an der Reihe ist. Die *Pause- Zeit* ist die Zeitdifferenz zwischen dem Beginn der Pause und dem Zeitpunkt, an dem die Anwendung wieder auf ihren Kanal zugreifen darf.

Für das mmul- und bfs- Szenario aus Abbildung 6.2 haben wir die Fence-, DMA- und Pause- Zeit im Treiber ausgemessen. Die Ergebnisse sind in Tabelle 6.3 angegeben. Auffällig ist, dass die Pause- Zeit in jedem Fall erheblich größer als die Summe der anderen Zeiten ist, obwohl unser Treiber den Kanal unmittelbar nach dem letzten DMA- Transfer fortsetzt. Hierfür haben wir keine Erklärung. Die lange DMA- Zeit bei mmul1 kommt durch die Auslastung der GPU mit langlaufenden Kernels zustande.

6.6 Diskussion und weitere Forschung

In diesem Abschnitt diskutieren wir Schwachpunkte am GPGPU- Swapping, die in diesem Kapitel deutlich geworden sind und zeigen Verbesserungspotential auf.

Wir betrachten das bfs- Szenario, das in Abbildung 6.2a dargestellt ist, ist ab Sekunde 2, ab der alle drei bfs Instanzen parallel ausgeführt werden. Wir erkennen, dass bfs2 und bfs3, die beide nur wenig Speicher abgeben mussten, kein eigenes Leistungsniveau oberhalb von bfs1 bilden. Stattdessen scheint bfs1, welches einen Großteil seines Speichers abgeben musste, die anderen beiden Instanzen auf sein Niveau „herabzuziehen“. Die Beobachtung ist, dass die Kernel der einzelnen Anwendungen reihum ausgeführt werden. Die Kernels von bfs1 sind wesentlich langsamer als die von bfs2 und bfs3, weil ein Großteil des Speichers von bfs1 ausgelagert wurde. Dennoch führt die GPU im Schnitt für jeden Kernel von bfs1 nur jeweils einen Kernel von bfs2 und bfs3 aus. Die Anwendung mit den langsamsten Kernels, in unserem Fall bfs1, erhält folglich den größten Anteil an der

GPU- Zeit. Für unser Swapping ist dies insofern relevant, als dass wir durch unsere Swapping- Entscheidungen die Laufzeit von Kernels verlängern. Wie wir in Abschnitt 4.1 angegeben haben, ist unser Ziel im Rahmen dieser Arbeit lediglich die faire Aufteilung von Speicher. Der Kunde eines Cloud- Dienstes mag jedoch wenig Wert darin erkennen, dass seine Anwendung einen fairen Anteil am VRAM erhält, wenn die Anwendung trotzdem von anderen Anwendungen mit besonders hohem Speicherverbrauch ausgebremst wird. Anwendungen mit hohem Speicherverbrauch entzieht der Swapping- Mechanismus zwar VRAM, doch dadurch verlängert sich Kernelausführung, weshalb die Anwendungen mit hohem Speicherverbrauch effektiv mehr GPU- Zeit erhalten und alle Anwendungen im System gleichermaßen ausgebremst werden. Erst die gleichmäßige Aufteilung der GPU-Zeit zwischen Anwendungen erzwingt, dass diejenige Anwendung, die das Swapping verursacht hat, auch durch dieses ausgebremst wird.

Der Hardware- Scheduler gewährt jeder Anwendung einen Timeslice [20]. Die GPU sollte daher bis zu einem gewissen Grad in der Lage sein, GPU- Zeit gerecht zwischen Anwendungen mit verschiedenen Kernellaufzeiten zu verteilen, indem sie zwei oder mehr Kernels mit kurzer Laufzeit ausführt, bevor sie den Kontext wechselt. Wir gehen davon aus, dass die GPU eine Form von Round- Robin Scheduling betreibt. Wie in Abschnitt 6.2 angegeben, rufen wir in unseren Benchmarks nach jedem Kernel Lauch cuCtxSynchronize auf, weshalb wir den nächsten Kernel Launch erst ausführen, nachdem der vorangegangene Kernel abgeschlossen wurde. Es befindet sich daher immer nur höchstens ein Kernel Launch im Indirect Buffer einer jeden Anwendung. Nachdem die GPU diesen Kernel abgearbeitet hat, muss sie deshalb den Kontext wechseln, selbst wenn der Timeslice noch nicht abgelaufen ist. Inwiefern der Verzicht auf cuCtxSynchronize der Hardware ermöglicht, die GPU- Zeit fairer aufzuteilen, werden weitere Experimente zeigen müssen. Ansonsten bieten sich Eingriffe durch die Software in das Scheduling an, um die GPU- Zeit aufzuteilen, was auch ermöglicht Anwendungen zu bestrafen, deren Kernels den Timeslice überziehen.

Der erhebliche Overhead von GPGPU- Swapping bleibt ein offenes Problem. Sowohl das vorauseilende Kopieren von Daten in den VRAM, als auch das dauerhafte Speichern im SYSRAM haben ihre Vor- und Nachteile. Wir denken, dass zukünftige Varianten des GPGPU- Swappings beide Möglichkeiten vereinen und je nach ausgelagertem Speicherbereich entscheiden werden, ob es sich lohnt, diesen in den VRAM zurück zu schreiben oder ob sie ihn lediglich in den GPU-Adressraum einblenden. Wir haben im vorangegangenen Abschnitt 6.5 am Beispiel von mmul gezeigt, dass Kernel beim Zugriff auf den SYSRAM die PCI- Express Bandbreite ausreizen. Es lohnt sich daher, wie wir in Abschnitt 4.3 dargelegt haben, nur dann, die Daten vorauseilend zu kopieren, wenn der Kernel wenigstens zwei oder drei mal auf die Daten zugreift. Für die Leistung der GPU hat es erhebliche Auswirkungen einen großen Datensatz in den VRAM zu kopieren, nur um

anschließend einen schnell- laufenden Kernel, der nur einen kleinen Teil der Daten anfasst, auszuführen. In diesem Fall wird die Laufzeit von der Kopieroperation dominiert. Es ist aber ebenso schädlich die Daten im Hauptspeicher zu belassen, wenn sich die Ausführungszeit eines lang laufenden Kernels um ein Vielfaches der Zeit, die für das Kopieren notwendig gewesen wäre, verlängert.

Mit gegenwärtiger Hardware ist es allerdings aus zwei Gründen schwer, das vorauseilende Kopieren mit dem einblenden von SYSRAM zu kombinieren. Zum einen erhält der Treiber keine Rückmeldung, beispielsweise durch accessed und dirty- Bits, über das Speichernutzungsverhalten der Anwendung. Zum anderen ist es dem Treiber, mangels Preemption, nach dem Kernel Launch nicht mehr möglich seine Entscheidung über die Platzierung eines Buffers zu ändern. Insofern sind zusätzliche Informationen über die Speicherzugriffe eines Kernels zur Laufzeit bestenfalls geeignet, eine Swapping- Entscheidung im Nachhinein zu bewerten und gegebenenfalls vor dem nächsten Start des selben Kernels eine andere Entscheidung zu treffen.

In [59] findet sich der Vorschlag, die Parameter zu betrachten, die eine Anwendung vor dem Launch an einen Kernel übergibt. Häufig enthalten diese die Anfangsadressen denjenigen Buffern, die der Kernel verwendet. Eine mögliche Strategie besteht darin, diese Buffer, sofern sie nicht zu groß sind, vor dem Kernel Launch in den VRAM zu kopieren. Denkbar ist auch eine statische Analyse des Kernel- Codes um mehr über das Zugriffsverhalten eines Kernels vorhersagen zu können.

Mehr über das Laufzeitverhalten eines Kernels lässt sich möglicherweise durch die IOMMU [62] in Erfahrung bringen. Diese übersetzt zwischen den Adressen mit denen die GPU auf den SYSRAM zugreift und den physischen Adressen im Hauptspeicher. Zumindest die IOMMU, die in aktuellen Intel- Server- Systemen vorhanden ist, kennt accessed und dirty- Bits [8]. Der Treiber kann daher auslesen, auf welche Buffer im SYSRAM die GPU zugreift. Der Treiber kann allerdings das Verhältnis von VRAM zu SYSRAM- Zugriffen nicht ermitteln, welches als Metrik für die Swapping- Entscheidung dienen könnte. Für diese Aufgabe eignen sich möglicherweise die auf der GPU verbauten Hardware- Counter [10].

Kapitel 7

Fazit

In dieser Arbeit haben wir ein Verfahren für das Swapping von VRAM- Buffern, die von GPGPU- Anwendungen angelegt werden, vorgestellt. Durch das Swapping lassen sich – für die Anwendung transparent – Situationen behandeln, in denen der benötigte VRAM die verfügbare Kapazität übersteigt. Fordert eine Anwendung VRAM an, obwohl nicht mehr hinreichend freier VRAM vorhanden ist, wählen wir Buffer im VRAM aus und verschieben diese in den SYSRAM. Während wir einen Buffer kopieren, verhindern wir, dass die Anwendung, zu der der Buffer gehört, in den VRAM schreibt. Das Swapping ist für die Anwendung transparent, weil wir die SYSRAM- Kopie des Buffers an der Stelle in den GPU-Adressraum einblenden, an der sich der ursprüngliche Buffer befand.

Gegenüber anderen Ansätzen für das GPGPU- Swapping bleiben bei unserem Verfahren alle Buffer dauerhaft erreichbar. Der Vorteil ist, dass unser Verfahren deshalb keine direkte Kontrolle über die Kernel Launches voraussetzt. Dadurch ist es möglich virtuellen Maschinen direkten Zugriff auf die GPU zu gewähren, wodurch effiziente Virtualisierung ermöglicht wird.

Wir haben unser Verfahren in einem Prototypen implementiert und haben mit Benchmarks gezeigt, dass der Overhead für das Swapping von Buffern, auf die selten zugegriffen wird, gering ausfällt. Ferner demonstrieren wir, dass unser Prototyp den VRAM fair auf konkurrierende Anwendungen aufteilt. Außerdem analysieren wir, wie sehr Kernel mit unterschiedlichen Zugriffsmustern durch das Swapping ausgebremst werden und welchen Overhead das Pausieren von Kanälen verursacht.

7.1 Zukünftige Arbeiten

Bislang führen wir unser Swapping mit den Buffern durch, die die Anwendung allokiert hat. Der Nachteil ist, dass wenn die Anwendungen große Buffer allokiert, die gleichmäßige Verteilung des VRAM auf mehrere Anwendungen nur eingeschränkt möglich ist. Wir werden daher unseren Prototypen erweitern, sodass Buffer der Anwendung in kleinere Verwaltungseinheiten unterteilt und unabhängig von einander ausgelagert werden können.

Außerdem werden wir die Kompatibilität des Prototypen mit verschiedenen Grafikkarten verbessern und die Probleme lösen, die bisher das Zurückkopieren vom SYSRAM in den VRAM verhindern.

Literaturverzeichnis

[1] cuBLAS. https://developer.nvidia.com/cublas.

[2] CUDA samples. http://docs.nvidia.com/cuda/cuda-samples.

[3] Envytools git documentation. http://envytools.readthedocs.org/en/latest/.

[4] GPU accelerated scientific computing: Evaluation of the NVIDIA Fermi architecture; elementary kernels and linear solvers. In Proceedings of the second International Workshop on New Frontiers in High-performance and Hardware-aware Computing, HipHaC'11.

[5] Intel® Math Kernel Library. https://software.intel.com/en-us/intel-mkl.

[6] Nouveau: Accelerated open source driver for nvidia cards. http://nouveau.freedesktop.org/wiki/.

[7] pscnv - PathScale NVIDIA graphics driver. https://github.com/pathscale/pscnv.

[8] Intel® virtualization technology for directed I/O, 2013. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf.

[9] CUDA C best practices guide, 2014. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[10] CUPTI Metric API, 2014. http://docs.nvidia.com/cuda/cupti/r_main.html#r_event_api.

[11] AMD. White paper | AMD graphics cores next (GCN) architecture, 2012. https://github.com/AMD-FirePro/SDK/blob/master/documentation/GCN_Architecture_whitepaper.pdf?raw=true.

[12] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, page 94–103. ACM, 2010. http://doi.acm.org/10.1145/1735688.1735706.

[13] Paolo Bonzini. Effictive multi-threading in qemu, 2013. http://www.linux-kvm.org/wiki/images/1/17/Kvm-forum-2013-Effective-multithreading-in-QEMU.pdf.

[14] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel. O'Reilly Media, Inc., third edition, 2005.

[15] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation, volume 2 of NSDI'05, page 273–286, Berkeley, CA, USA, 2005. USENIX Association. http://dl.acm.org/citation.cfm?id=1251203.1251223.

[16] Jonathan Corbet. Memory management for graphics processors, 2007. http://lwn.net/Articles/257417/.

[17] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers, 3rd Edition. O'Reilly Media, third edition, 2005.

[18] Microsoft Corporation. Windows display driver model (WDDM) architecture. http://msdn.microsoft.com/en-us/library/windows/hardware/ff570589(v=vs.85).aspx.

[19] NVIDIA Corporation. Bindless graphics tutorial. http://www.nvidia.com/object/bindless_graphics.html.

[20] Nvidia Corporation. Nvidia's next generation cudatm compute architecture: Fermi. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[21] Nvidia Corporation. Programming guide :: CUDA toolkit documentation, 2014. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[22] Micah Dowty and Jeremy Sugerman. Gpu virtualization on VMware's hosted I/O architecture. SIGOPS Oper. Syst. Rev., 43(3):73–82, 2009. http://doi.acm.org/10.1145/1618525.1618534.

[23] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In Proceedings of International Conference on High Performance Computing and Simulation, HPCS, page 224–231, 2010.

[24] Stéphane Marchesin et al. Linux graphics drivers: an introduction, 2012. http://cgit.freedesktop.org/~marcheu/lgd/.

[25] Simon Fenney. Texture compression using low-frequency signal modulation. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, HWWS '03, page 84–91. Eurographics Association, 2003. http://dl.acm.org/citation.cfm?id=844174.844187.

[26] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. 45(3):347–358, 2010. https://adsm.googlecode.com/files/adsm-asplos10.pdf.

[27] Pawel Gepner, David L Fraser, and Victor Gamayunov. Evaluation of the 3rd generation Intel Core processor focusing on HPC applications. http://www.worldcomp-proceedings.com/proc/p2012/PDP2833.pdf.

[28] Fabian Giesen. A trip through the graphics pipeline, 2011. http://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/.

[29] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU transparent virtualization component for high performance computing clouds. In Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I, EuroPar'10, page 379–391, Berlin, Heidelberg, 2010. Springer-Verlag. http://dl.acm.org/citation.cfm?id=1887695.1887738.

[30] Mathias Gottschlag. Virtualization and migration with GPGPUs. Bachelor thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), 2013. http://os.itec.kit.edu/downloads/ba_2013_gottschlag-mathias_GPGPU.pdf.

[31] Mathias Gottschlag, Marius Hillenbrand, Jens Kehne, Jan Stoess, and Frank Bellosa. LoGV: Low-overhead GPGPU virtualization. In Proceedings of the 4th International Workshop on Frontiers of Heterogeneous Computing, FHC '13, page 1721–1726, Zhangjiajie, China, 2013. IEEE.

[32] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete fourier transforms on graphics processors. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE, 2008. http://research.microsoft.com/apps/pubs/default.aspx?id=131400.

[33] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In International Symposium on Performance Analysis of Systems and Software, ISPASS, page 134–144, 2011.

[34] Khronos Group. GLAPI/glBindBuffer. http://www.opengl.org/wiki/GLAPI/glBindBuffer.

[35] Khronos Group. Khronos OpenCL registry, 2014. https://www.khronos.org/registry/cl/.

[36] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing, HPCVirt '09, page 17–24, New York, NY, USA, 2009. ACM. http://doi.acm.org/10.1145/1519138.1519141.

[37] Mark Harris. Unified memory in CUDA 6, 2013. http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[38] Keon Jang, Sangjin Han, Seungyeop Han, Sue B Moon, and KyoungSoo Park. SSLShader: Cheap SSL acceleration with commodity processors. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation, NSDI'11, 2011. http://www.usenix.org/events/nsdi11/tech/full_papers/Jang.pdf.

[39] Feng Ji, Heshan Lin, and Xiaosong Ma. Rsvm: A region-based software virtual memory for gpu. In 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT, page 269–278, 2013.

[40] Shinpei Kato. Implementing open-source CUDA runtime. In Proceedings of the 54th Programming Symposium, 2013. http://www.ertl.jp/~shinpei/papers/pro13.pdf.

[41] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class gpu resource management in the operating system. In Proceedings

of the 2012 USENIX Annual Technical Conference, USENIX ATC'12, Berkeley, CA, USA, 2012. USENIX Association. http://dl.acm.org/citation.cfm?id=2342821.2342858.

[42] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News, 38(3):451–460, 2010. http://doi.acm.org/10.1145/1816038.1816021.

[43] Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinformatics, 9, 2008. http://www.biomedcentral.com/1471-2105/9/S2/S10.

[44] Viktor Mauch, Marcel Kunze, and Marius Hillenbrand. High performance cloud computing. Future Generation Computer Systems, 29(6):1408–1416, 2013.

[45] Paulius Micikevicius. Local memory and register spilling, 2011. http://on-demand.gputechconf.com/gtc-express/2011/presentations/register_spilling.pdf.

[46] Lucas Nussbaum, Fabienne Anhalt, Olivier Mornard, and Jean-Patrick Gelas. Linux-based virtualization for HPC clusters. In Ottawa Linux Symposium, page 221–234, 2009. https://www.kernel.org/doc/ols/2009/ols2009-pages-221-234.pdf.

[47] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU computing. Proceedings of the IEEE, 96(5):879--899, 2008.

[48] PCI-SIG. PCI Express base specification revision 3.0, 2010.

[49] Martin Peres. a deeper look into GPUs and the Linux graphics stack, 2012. http://phd.mupuf.org/files/toulibre2012_deeper_look.pdf.

[50] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 address translation for 100s of gpu lanes. In Proceedings of 20th International Symposium on High Performance Computer Architecture, HPCA, page 568–578, 2014.

[51] Nouveau Project. Codenames. http://nouveau.freedesktop.org/wiki/CodeNames/.

[52] M. Boyer L. G. Szafaryn L. Wang S. Che, J. W. Sheaffer and K. Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In Proceedings of the IEEE International Symposium on Workload Characterization, 2010.

[53] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In Proceedings of the IEEE International Symposium on Parallel Distributed Processing, IPDPS, page 1–11, 2009.

[54] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: integrating file systems with gpus. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13. ACM, 2013.

[55] Andrew S. Tanenbaum. Modern Operating Systems (3rd Edition). Prentice Hall, 3 edition, 2007.

[56] Daniel Vetter. GEM overview, 2011. http://blog.ffwll.ch/2011/05/gem-overview.html.

[57] M.S. Vinaya, N. Vydyanathan, and M. Gajjar. An evaluation of CUDA-enabled virtualization solutions. In Proceedings of the 2nd IEEE International Conference on Parallel Distributed and Grid Computing, PDGC, page 621–626, 2012.

[58] Vasily Volkov and James Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, 2008. http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html.

[59] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. GDM: Device memory management for GPGPU computing. System, 2:2, 2014.

[60] S. Xiao, Pavan Balaji, Q. Zhu, Rajeev Thakur, Susan M. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In Proceedings of the 1st Innovative Parallel Computing, InPar, San Jose, CA, 2012. IEEE, IEEE. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.233.2051.

[61] Zhiyi Yang, Yating Zhu, and Yong Pu. Parallel image processing based on CUDA. In Proceedings of the International Conference on Computer Science and Software Engineering, volume 3, page 198–201, 2008.

[62] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical report, IBM Research, 2008.