Evaluierung der Performanz von GPUDirect 2.0 auf einem Multi-GPU System

Bachelorarbeit

Für die Prüfung zum

Bachelor of Engineering

im Studiengang Informationstechnik

an der Dualen Hochschule Baden-Württemberg Mosbach

von

Michael Drost

September 2012

Bearbeitungszeitraum	12 Wochen
Matrikelnummer	5196114
Ausbildungsfirma	Fraunhofer SCAI
	Schloss Birlinghoven
	53754 Sankt Augustin
Betreuer der Ausbildungsfirma	Dr. Thomas Brandes
Betreuer der DHBW Mosbach	Prof. Dr. Wolfgang Funk

Inhaltsverzeichnis

Abkürzungsverzeichnis

BLAS Basic Linear Algebra Subprograms

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DMA Direct Memory Access

ECC Error Correction Code

GEMM General Matrix Multiply

GPU Graphics Processing Unit

IOH I/O Hub

IPC Inter-Process Communication

LAMA Library for Accelerated Math Applications

MPI Message Passing Interface

NUMA Non Uniform Memory Access

PCIe Peripheral Component Interconnect Express

RAM Random Access Memory

RDMA Remote Direct Memory Access

Glossar

NUMA-Rechner Mehrprozessor Rechner mit physikalisch getrenntem Speicher aber gemeinsamem Speicherbereich

cuBlas Eine BLAS Bibliothek die von NVIDIA für die Verwendung auf NVIDIA Grafikkarten zur Verfügung gestellt wird. cuBlas ist Teil des zum Entwickeln benötigten NVIDIA CUDA Toolkit.

Warp Die kleinste Einheit von Threads auf der Grafikkarte. Alle Threads eines Warps können immer nur die selbe Anweisung zu einem Zeitpunkt ausführen.

Abbildungsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

In modernen Hochleistungsrechnern werden zunehmend auch Grafikkarten für die Berechnungen eingesetzt. Jede Grafikkarte besitzt ihren eigenen Speicher. Wenn ein Problem auf mehrere Grafikkarten aufgeteilt werden soll, so müssen die Grafikkarten untereinander Daten austauschen. Bisher mussten die Daten dazu in den Hauptspeicher geladen und von dort wieder auf die andere Grafikkarte kopiert werden. Ab Compute Unified Device Architecture (CUDA) 4.0(2011) ist es durch GPUDirect möglich, Daten direkt zwischen zwei Grafikkarten zu übertragen, so dass keine zusätzliche Kopie im Hauptspeicher angelegt werden muss.

Im Rahmen dieser Arbeit soll der Einfluss von GPUDirect auf die Performance einer typischen Anwendung aus der numerischen Mathematik untersucht werden. Weiterhin soll untersucht werden, wie GPUDirect in einer Bibliothek zur Lösung von linearen Gleichungssystemen LAMA eingesetzt werden kann.

1.2 Zielsetzung

Ziel dieser Arbeit ist es einen Überblick über die Möglichkeiten zu geben, die GPUDirect bietet. Anhand eines Beispielprogramms soll die reine Bandbreite bzw. die Verbesserung dieser mit GPUDirect und cudaIPC dargestellt werden.

Mit Hilfe eines weiteren Beispielprogramms, der Matrix-Matrix-Multiplikation, soll gezeigt werden, wie ein praktisches Problem von der Beschleunigung durch GPUDirect profitieren kann. Des Weiteren sollen die Ergebnisse dieser Performancemessung genau analysiert und erklärt werden.

Anschließend soll ein Konzept erarbeitet werden, wie GPUDirect bzw. cudaIPC in das LAMA Projekt eingebunden werden kann. Das Konzept soll Aufschluss darüber geben, wie aufwendig es ist, die vorgestellten Techniken in eine Bibliothek zu integrieren.

2 Problembeschreibung

Die Programmierung von Grafikkarten zur Lösung von massiv parallelen Problemen setzt sich in letzter Zeit immer stärker durch. Grafikkarten zeichnen sich vor allem dadurch aus, dass sie mit sehr vielen Threads die gleiche Aufgabe lösen können. Gewöhnliche CPUs hingegen lösen alle Aufgaben nacheinander. Dafür besitzen CPUs eine deutlich höhere Geschwindigkeit. Deshalb ist die GPU vor allem bei parallelen Problemen schneller als eine CPU. Ein paralleles Problem ist ein Problem, bei dem der selbe Programmcode mehrmals unabhängig voneinander ausgeführt werden muss. Die vielen unabhängigen Vorgänge können somit, z.B. auf einer Grafikkarte, gleichzeitig ausgeführt werden.

Die größte Schwäche der Grafikkarten bei der Beschleunigung von Programmen liegt darin, dass die Datenübertragung zwischen Grafikkarte und Hostrechner sehr langsam ist. So beansprucht der Datentransfer bei einer gewöhnlichen Berechnung den Hauptteil der Laufzeit.

Zudem kann der Speicher von der Grafikkarte meist nur in einen speziellen Bereich auf den Hostrechner übertragen werden. Dieser Speicherbereich ist speziell gepinnt, so dass er immer an der selben Stelle im Random Access Memory (RAM) steht. So wird verhindert, dass dieser auf die Festplatte ausgelagert wird, oder sich der Speicher im RAM verschiebt bzw. sich die Adresse ändert. Durch diesen gepinnten Speicher kann sichergestellt werden, dass die Grafikkarte die Daten immer an der selben Stelle findet und keine falschen Daten liest oder schreibt.

Daten, die auf eine andere Grafikkarte kopiert werden sollen, müssen deshalb erst auf den Host kopiert werden, damit sie anschließend auf die andere Grafikkarte kopiert werden können. Das selbe Problem besteht beim Senden von Daten über das Netzwerk.

Viele Netzwerkkarten, die im High Performance Bereich eingesetzt werden, können die Daten ähnlich wie bei der Grafikkarte auch nur verwenden, wenn diese in einem speziellen gepinnten Speicherbereich liegen. Der Speicherbereich, der von CUDA verwendet wird, ist im Normalfall nicht kompatibel mit dem Speicher, der von der Netzwerkhardware verwendet wird. So müssen, wie schon beschrieben, Daten, die mit CUDA bereits in einem gepinnten Bereich angelegt wurden nochmals in den Speicherbereich, der von der Netzwerkkarte verwendet werden kann kopiert werden. In diesem Fall muss sogar vier Mal kopiert werden. Zu erst auf den von CUDA gepinnten Speicher und

anschließend in den Speicher der für die Netzwerkkarte zugreifbar ist. Von dort können die Daten an einen anderen Rechner versendet werden. Der andere Rechner empfängt die versendeten Daten in einem speziellen Speicherbereich. Von dort aus müssen die Daten anschließend in einen von der Compute Unified Device Architecture (CUDA) gepinnten Speicher kopiert werden.

Seit der Version 4.0 von CUDA ist es möglich, Daten direkt von einer Grafikkarte auf eine andere zu kopieren. Diese Technik wird GPUDirect genannt. In den weiteren Versionen von CUDA wurde GPUDirect weiterentwickelt und erweitert. [1][2][5]

3 Stand der Technik

3.1 GPUDirect

GPUDirect wurde im Juni 2010 vorgestellt und seit dem mehrfach verbessert und erweitert. So ist es mit GPUDirect möglich, dass die Netzwerkkarte auch auf den selben Speicherbereich zugreifen kann wie die Grafikkarte(Abschnitt 3.1.1). Eine weitere Möglichkeit ist die Verwendung von Direct Memory Access (DMA) zwischen den Grafikkarten(Abschnitt 3.1.2). Die neuste Verbesserung besteht darin, dass auch ein RDMA Zugriff auf den Grafikspeicher möglich sein soll. [5]

3.1.1 Kompatibilität des gepinnten Speichers

Das erste von GPUDirect eingeführte Feature war die Möglichkeit gemeinsamen Speicher mit dem Netzwerktreiber zu nutzen. Die Möglichkeit gemeinsamen Speicher zu benutzen ist bisher auf zwei Hersteller(Mellanox und QLogic) von Infinibandkarten beschränkt. Infinibandkarten sind im High Performance Bereich eingesetzte Netzwerkkarten, die sich vor allem durch ihre geringe Latenz aber auch durch ihre hohe Bandbreite auszeichnen.

Um die direkte Kommunikation von Hardwareelementen am Peripheral Component Interconnect Express (PCIe) Bus und dem RAM zu ermöglichen muss ein spezieller Speicherbereich angelegt werden. Dieser Speicherbereich ist gepinnt, d.h. er befindet sich immer an der selben Stelle im RAM und kann nicht ausgelagert werden. So wird sichergestellt, dass die Hardware immer mit den selben Adressen Zugriff auf den Speicher besitzt.

Der gepinnte Speicherbereich wird in der Regel vom Treiber der Hardwarekomponente angelegt und verwaltet. Da jeder Treiber seinen eigenen Speicherbereich verwaltet, sind diese Speicherbereiche nicht untereinander kompatibel. Daten, die von einem Gerät auf das andere Gerät kopiert werden sollen müssen also zuerst in den RAM des einen Treibers und anschließend in den RAM des anderen Treibers kopiert werden.

Für eine Netzwerkkommunikation von der Grafikkarte eines Knotens zu der Grafikkarte eines

anderen Knotens sind also, wie in Grafik 3.1 zu sehen, insgesamt sieben Transfers notwendig. Der rot markierte Transfer stellt den Transfer dar, der von der CPU durchgeführt wird, um die Daten vom Speicherbereich des Grafikkartentreibers in den Speicher des Netzwerktreibers zu kopieren.

Dieser Transfer kann mithilfe des von GPUDirect gespart werden, indem der Treiber der Grafikkarte und der Treiber der Netzwerkkarte den gleichen gepinnten Speicher verwenden. Wie in Grafik 3.2 dargestellt sind in diesem Fall nur fünf Transfers notwendig.

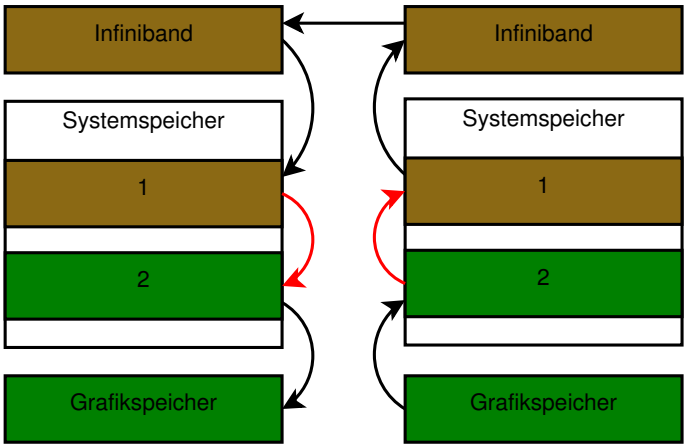

Abbildung 3.1: Schema der Kommunikation zweier Grafikkarten in verschiedenen Knoten ohne GPUDirect

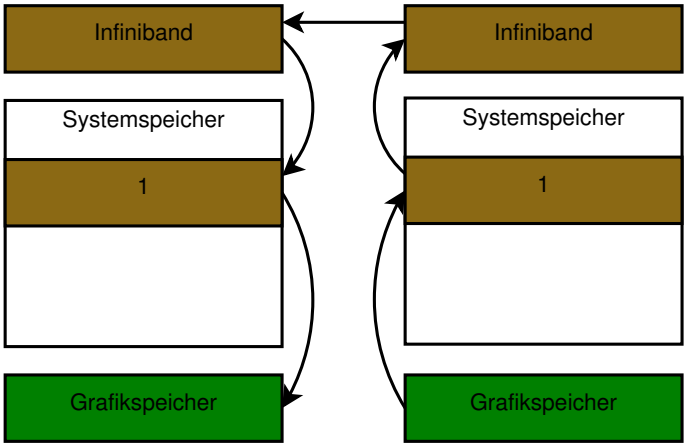

Abbildung 3.2: Schema der Kommunikation zweier Grafikkarten in verschiedenen Knoten mit GPUDirect

[1][2][5]

3.1.2 Direkttransfer zwischen Grafikkarten

Ohne GPUDirect müssen Daten, die von einer auf die andere Grafikkarte kopiert werden solle zuvor auf den Host geladen werden. Von dort aus können die Daten anschließend zu der anderen Grafikkarte kopiert werden.

Mit GPUDirect ist es möglich, Daten direkt über den PCIe Bus zu kopieren. So kann eine Grafikkarte direkt in den Speicher einer anderen Grafikkarte kopieren. Durch das direkte Kopieren über den PCIe Bus kann der Umweg über den Hostspeicher gespart und damit die Bandbreite vergrößert werden.

Weiterhin ermöglicht GPUDirekt den direkten Zugriff auf den Speicher der anderen Grafikkarten. So kann eine Grafikkarte direkt auf Daten zugreifen, die auf einer anderen Grafikkarte gespeichert sind.

Diese beiden Techniken benötigen aber mindestens NVIDIA Grafikkarten der Fermi Generation. Zusätzlich müssen alle Grafikkarten am selben PCIe Bus angeschlossen sein. Besonders bei Non Uniform Memory Access (NUMA) Rechnern ist dies oft nicht der Fall, da diese mehrere CPUs besitzen. Oft besitzt jede CPU wie in Grafik 3.3 einen eigenen I/O Hub (IOH) Controller der mit einem PCIe Bus verbunden ist. Die Grafikkarten `GPU0` und `GPU1` können also direkt miteinander kommunizieren. Die Grafikkarte `GPU0` und die Grafikkarte `GPU2` können nicht direkt miteinander kommunizieren, da das beim PCIe Bus verwendete Protokoll einen Umweg über den IOH Controller nicht vorsieht. Von GPUDirect wird dies aber für den Nutzer transparent behandelt, so dass dieser die Daten von einem auf das andere Device kopiert. Und GPUDirect für den Fall, dass die Grafikkarten nicht direkt untereinander kommunizieren können, Speicher auf dem Host anlegt und dort zwischenspeichert. [1][2][5][6][7][8]

3.1.3 Remote Direct Memory Access

Eine weitere Funktion von GPUDirect soll RDMA Zugriff auf die Grafikkarte sein. Mit RDMA Zugriff ist es möglich, dass entfernte Knoten über spezielle Netzwerkhardware (wie z.B. Infiniband) auf den Speicher der Grafikkarte zugreifen. Zum einen Ermöglicht dies einen einseitigen Transfer auch für den Grafikkartenspeicher. Zum anderen ermöglicht es diese Funktion im Vergleich zur gemeinsamen Verwendung des gepinnten Speichers(siehe Grafik 3.2) eine noch effektivere Variante.

Wie in Grafik 3.4 zu sehen kann bei dieser Variante des Transfers der Hauptspeicher des Computer komplett umgangen werden. Im Idealfall muss dazu die CPU gar nicht belastet werden und kann so simultan für Berechnungen eingesetzt werden. Weiterhin wird im Vergleich zur Kommunikation

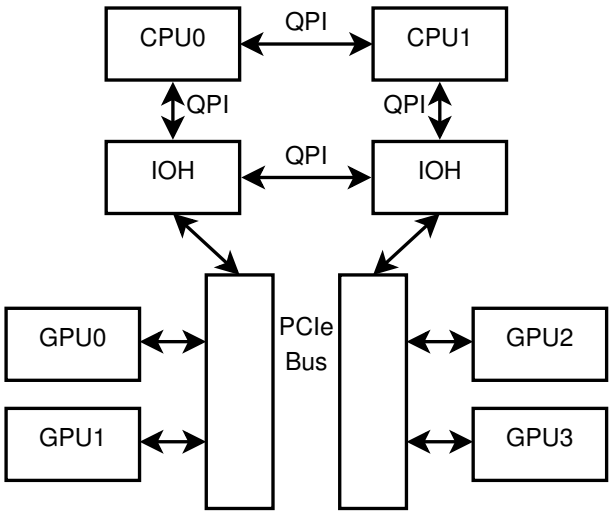

Abbildung 3.3: Schema der Hardware Topologie eines NUMA Rechners mit 2 CPUs und 4 GPUs

mit gemeinsamen gepinnten Speicher (siehe Grafik 3.2) eine weitere Zwischenspeicherungsschicht abgebaut, so dass eine höhere Transferrate erreicht werden kann.

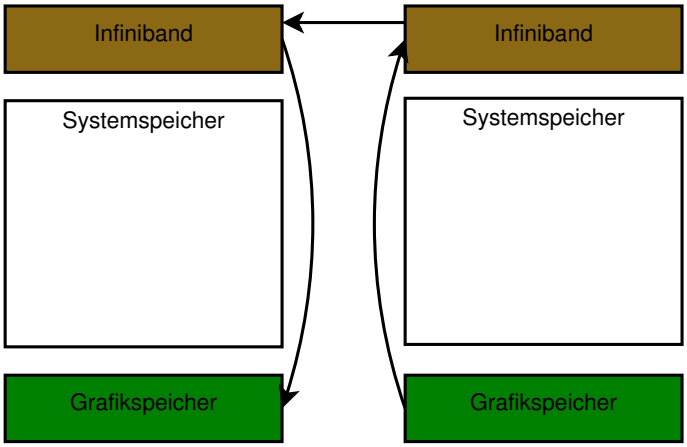

Abbildung 3.4: Schema der Kommunikation zweier Grafikkarten in verschiedenen Knoten mit RDMA

Die RDMA Funktion von GPUDirect soll mit allen Kepler K20 GPU der Tesla Serie funktionieren. Da die Grafikkarten mit Kepler K20 Chipsatz erst im vierten Quartal 2012 ausgeliefert werden, kann die RDMA Funktion in dieser Arbeit nicht betrachtet werden. [5][7][9][10]

3.2 cudaIPC

Mit cuda Inter-Process Communication (IPC) bietet NVIDIA eine weitere Möglichkeit parallele Programme, die mehrere GPUs verwenden, zu beschleunigen. So ist es möglich, aus einem Pointer auf Speicher, der auf der Grafikkarte liegt, einen Memory Handle zu erzeugen. Dieser Memory Handle besitzt alle nötigen Informationen um die Position des Speichers genau zu identifizieren. Der genaue Aufbau der Memory Handles wird in den Headern der CUDA Runtime leider nicht ersichtlich. Dort wird nur ein Array der Größe 64 Byte angelegt.

Mit einem Memory Handle ist es für ein anderes Programm, das auf dem selben Rechner abläuft, möglich, wieder einen Pointer zu erzeugen. Mithilfe dieses Pointers kann der zweite Prozess nun direkt auf den Grafikspeicher, den der erste Prozess angelegt hat, zugreifen. Mit Hilfe dieses Mechanismus können die zuvor beschriebenen Möglichkeiten von GPUDirekt besser ausgenutzt werden. So werden z.B. Direkttransfers von Grafikkarten möglich, die zwar auf dem selben Host liegen aber von unterschiedlichen Prozessen genutzt werden. Mit diesen Direkttransfers kann die Zwischenspeicherung im RAM, wie sie bisher bei der Kommunikation zwischen Prozessen(sie Abschnitt 3.1.1) immer nötig war, komplett umgangen werden. [1][2][7]

3.3 Was ist LAMA?

Die Library for Accelerated Math Applications (LAMA) ist ein Projekt des Fraunhofer SCAI. LAMA ist eine Bibliothek, die numerische Operationen zur Verfügung stellt. LAMA stellt ein Interface bereit, mit dem Matrizen in verschiedenen Formaten erzeugt werden können, darunter auch viele Sparse Matrix Formate. LAMA bietet für alle Matrixformate einheitliche Konstruktoren und Operationen an, so dass bereits mit LAMA entwickelte Programme einfach auf andere Matrixformate umgestellt werden können.

Weiterhin erlaubt LAMA die Verwendung von verschiedenen Hardwarebeschleunigern wie z.B. GPUs sowie die Verwendung von mehreren Rechnern in Verbund. Alle diese Funktionen können bei der Erzeugung einer Matrix oder eines Vektors ausgewählt werden, so dass die Programmlogik nicht für jeden Fall angepasst werden muss.

Ein zusätzliches Feature von LAMA ist die einfache Programmierung, so können alle Operationen mit überladenen Operatoren durchgeführt werden. d.h. Es kann in einer Art Textbuchsyntax entwickelt werden, was die Fehlerquellen beim Entwickeln von Algorithmen deutlich senken kann. Wie schon beschrieben kann auch der entwickelte Code auf einfache Weise auf der Grafikkarte oder verteilt über das Netzwerk ausgeführt werden.[3]

3.3.1 Verwendung von MultiGPU Systemen in LAMA

In LAMA gibt es mehrere Möglichkeiten um MultiGPU Systeme zu verwenden. Zum einen ist es möglich, mehrere Grafikkarten in einer Instanz zu verwenden. Dazu müssen die Speicher der Matrizen bzw. der Vektoren auf verschiedenen Grafikkarten angelegt werden. Um den Speicherort von Daten anzugeben, werden in LAMA Instanzen der Klasse `Context` verwendet. Um Daten auf einer Grafikkarte zu speichern wird beispielsweise die Klasse `CUDAContext` verwendet. Für jede eingebaute cudafähige Grafikkarte kann ein `CUDAContext` erzeugt werden. Mithilfe mehrerer `CUDAContext` Objekte können die verschiedenen Matrizen und Vektoren auf mehrere Grafikkarten verteilt werden. Wird auf diesen Strukturen, die auf mehrere Grafikkarten verteilt sind, eine Berechnung durchgeführt, so werden alle an der Berechnung beteiligten Daten auf eine Grafikkarte kopiert. Dort wird die Berechnung dann ausgeführt. Mit dieser Variante ist es nicht möglich, die Berechnung auf mehrere Grafikkarten aufzuteilen. Dieser Fall der Verwendung von mehreren Grafikkarten in LAMA wäre ein typischer Anwendungsfall für GPUDirect.

Eine weitere Möglichkeit zur Verwendung von mehreren GPUs ist die Verteilung des Problems auf mehrere Programminstanzen und somit auf mehrere Rechner oder mehrere Grafikkarten. Dazu müssen die Matrizen bzw. Vektoren angelegt werden und anschließend mit einer Instanz der Klasse `Distribution` auf mehrere Instanzen verteilt werden. Auf diesen Instanzen können die Daten dann mit Hilfe eines `CUDAContext` auf der Grafikkarte gespeichert werden.

Nachdem die Matrizen bzw. Vektoren verteilt sind, ist es mit LAMA möglich, völlig transparent mit diesen Strukturen zu rechnen. Der Benutzer muss nur zur Erzeugung der Strukturen angeben, ob und wie diese verteilt werden sollen. Der Algorithmus, der zur Lösung eines Problems eingesetzt wird, ist also vollkommen unabhängig von der Verteilung der zugrundeliegenden Strukturen.

Für die Berechnung wird auf den Instanzen temporärer Speicher angelegt. In diesen Speicher werden die Daten geladen, die von anderen Instanzen benötigt werden. Diese Kommunikation erfolgt nach einem definierten Ablauf, der die Eigenschaften der einzelnen an der Berechnung beteiligten Strukturen berücksichtigt. Dieser Ablauf wird in einer Instanz der Klasse `CommuniationPlan` gespeichert.

Mit dieser Variante der Verteilung ist es möglich den Algorithmus auf mehrere Grafikkarten aufzuteilen. Die Verteilung kann auch ohne Änderungen am Code auf mehrere Rechner aufgeteilt werden. Somit stellt diese Möglichkeit der Verteilung die flexibelste Form dar. Der Nachteil dieser Variante ist, dass sämtliche Kommunikation über das Netzwerk bzw. den RAM läuft, selbst wenn die Instanzen auf dem selben Rechner liegen. Diese Variante der Verteilung wäre ein typisches Anwendungsbeispiel für cudaIPC. Die Kommunikation zwischen den Grafikkarten kann anschließend noch weiter mit GPUDirect beschleunigt werden.

4 Das Testsystem

Als Testsystem wird ein NUMA-Rechner eingesetzt. Dieser Rechner verfügt über zwei Intel® Xeon® E5620 auf zwei verschiedenen Sockeln. Jeder Prozessor ist direkt an die Hälfte des $48\,GiB$ großen RAM angeschlossen. An dem IO Controller jedes Prozessors ist ein PCIe Bus angeschlossen. An jedem dieser PCIe Busse sind zwei Grafikkarten angebunden.

Die Grafikkarten sind vom Typ NVIDIA Tesla C2070 und besitzen jeweils $6\,GB$ Grafikspeicher. Auf dem Testsystem ist der Error Correction Code (ECC) der der Grafikkarten aktiviert, so dass effektiv nur $\frac{8}{9}$ des Speichers nutzbar sind.

Die theoretische Peak-Performance der Grafikkarten liegt bei jeweils 1030 GFlops in Single Precision. [8][11][12]

4.1 Topologie

Da das Testsystem ein NUMA-Rechner ist, besitzt es auch zwei physikalisch getrennte Speicherbereiche. Jeweils die Hälfte des Speichers ist an einen der beiden Prozessoren angebunden. Der Speicher wird aber trotzdem durchgehend adressiert, so dass jeder Prozessor auf den gesamten Arbeitsspeicher zugreifen kann. Der Zugriff auf den Arbeitsspeicher, der nicht lokal an dem Rechner angeschlossen ist, erfolgt dabei aber langsamer als der Zugriff auf den lokal angeschlossenen Rechner.

Auch der PCIe-Bus ist getrennt, so dass jeder Prozessor seinen eigenen PCIe-Bus besitzt. Jeder Prozessor ist über seinen PCIe-Bus mit zwei Grafikkarten verbunden. Die Grafikkarten sind direkt an dem PCIe-Bus des IOH Controllers angebunden.

4.2 Verbesserungsmöglichkeiten

4.2.1 PCIe-Switches

Die Performance der Grafikkarten könnte noch durch den Einsatz eines PCIe Switches verbessert werden. Ein PCIe-Switch ist meist direkt auf dem Mainboard verbaut oder muss dort für eine spätere Installation vorgesehen sein. Ein PCIe-Switch regelt die Kommunikation mehrerer PCIe Geräte über nur einen PCIe Anschluss an den darüberliegenden PCIe-Bus. Mit Hilfe eines PCIe-Switches kann auch die Transfergeschwindigkeit zwischen zwei Geräten gesteigert werden. Dazu müssen die Geräte direkt an den PCIe-Switch angeschlossen werden.

Wie In Grafik 4.1 zu sehen ist, müssen die Grafikkarten GPU2 und GPU3 über den PCIe-Bus und den IOH kommunizieren. Die Grafikkarten GPU0 und GPU1 können jedoch direkt über den PCIe-Switch kommunizieren und somit höhere Geschwindigkeiten erreichen. Der Nachteil eines PCIe-Switches liegt darin, dass die Geräte die an einem PCIe-Switch liegen sich einen Anschluss an den PCIe-Bus teilen müssen. Durch PCIe-Switches wird aber auch die Anzahl der an den PCIe-Bus angeschlossenen Geräte verringert und somit die Bandbreite erhöht. [7]

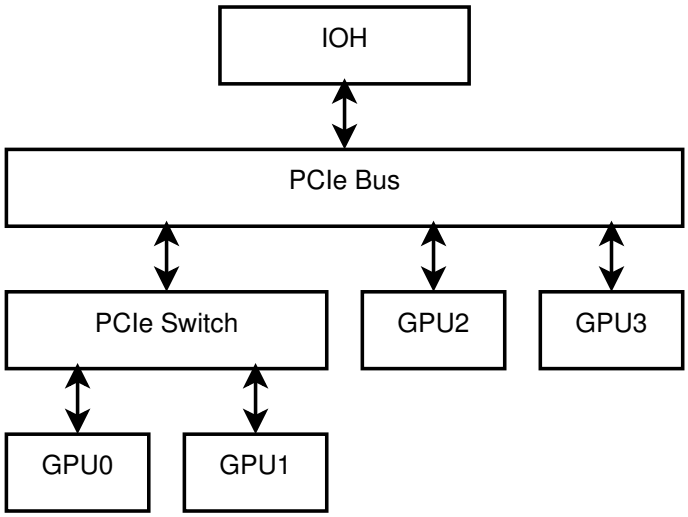

Abbildung 4.1: Topologie eines Systems mit PCIe Switch

4.2.2 Veränderung der Topologie

Eine weitere Möglichkeit die Performance für reine Grafikkartenprogramme zu beschleunigen wäre alle Grafikkarten an einen PCIe Bus anzuschließen, so kann die Performance für Übertragungen zwischen den einzelnen Grafikkarten verbessert werden. Auch diese Topologie könnte noch durch

den Einsatz von PCIe Switches verbessert werden. Auch wenn der Transfer über mehrere PCIe-Switches hinweg erfolgen muss, kann trotzdem die Beschleunigung des Transfers mit GPUDirect ausgenutzt werden.

Der Nachteil dieser Erweiterung wäre die Verringerung der gesamten möglichen Bandbreite vom RAM auf die Grafikkarten. Wenn alle Grafikkarten an einem PCIe Bus angeschlossen sind, kann für die Übertragung der Daten nur noch ein PCIe Bus verwendet werden.[7]

5 Erwartungen

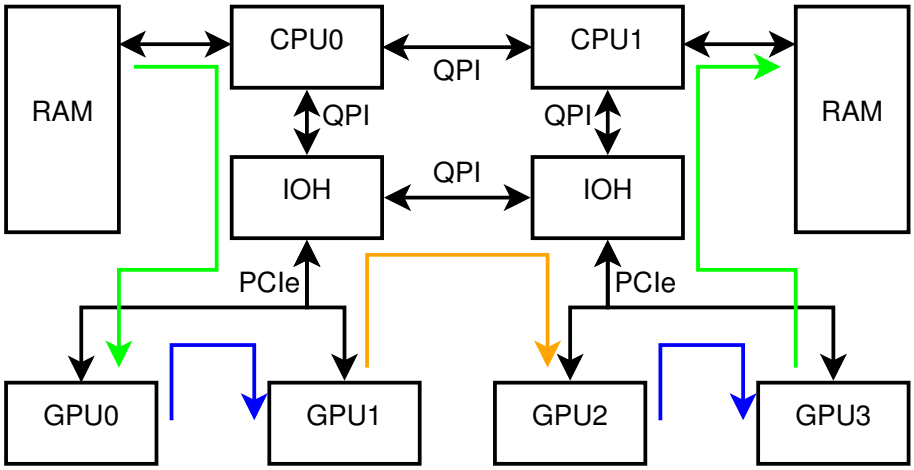

Abbildung 5.1: Topologie des Testsystems

Laut NVIDIA soll bei der Verwendung von GPUDirect von einer Grafikkarte zur anderen Grafikkarte (siehe Grafik 5.1 blau) bei Verwendung von PCIe Switches eine maximale Bandbreite von maximal $6,3\,GB/s$ erreicht werden. Da das Testsystem (siehe Abschnitt 4) keine PCIe Switches besitzt, sondern die Grafikkarten direkt mit dem PCIe Bus des IOH verbunden sind, ist auf dem Testsystem eine maximale Bandbreite von $5,3\,GB/s$ möglich. Durch den eingeschalteten ECC Modus sollte die erreichbare Bandbreite noch etwas geringer sein. Von einer auf die andere Grafikkarte über den Hostspeicher (siehe Abbildung 5.1 orange) über einen PCIe Bus hinaus soll eine maximale Bandbreite von $2,2\,GB/s$ möglich sein.

Der Transfer von der Grafikkarte in den RAM (siehe Abbildung 5.1 grün) sollte die gesamte Bandbreite des PCIe Bus ausnutzen können. Das entspricht maximal, wie beim direkten Transfer von einer zur anderen Grafikkarte, $6,3\,GB/s$.

Beim Transfer über Prozessgrenzen hinweg z.B. per Message Passing Interface (MPI) soll mit der Verwendung von cudaIPC (siehe Abschnitt 3.2) eine vergleichbare Bandbreite erreicht werden können. Bei der Verwendung von cudaIPC kommt allerdings zum eigentlichen Transfer noch die Erzeugung und Versendung des Memory Handles hinzu, so dass die Bandbreite etwas geringer ausfallen sollte. [7]

5.1 erste Messungen

Als erstes Beispiel wurde ein Testprogramm erarbeitet, das Speicher in den verschiedenen Speicherbereichen anlegen kann. Das Programm unterstützt 3 Arten von Speicherbereichen: Speicher der mit Hilfe von `malloc()` angelegt wurde, gepinnter Speicher, der mit der CUDA Runtime angelegt wurde und Speicher, der auf der Grafikkarte angelegt wurde. Der auf der Grafikkarte angelegte Speicher kann auch auf verschiedenen Grafikkarten angelegt werden, so dass auch der Transfer von einer zur anderen Grafikkarte getestet werden kann.

Zusätzlich ist es möglich, alle Speicher auch verteilt anzulegen, so dass diese auf verschiedenen Prozessinstanzen angelegt werden. Die Kommunikation erfolgt in diesem Fall über MPI bzw. cudaIPC und MPI.

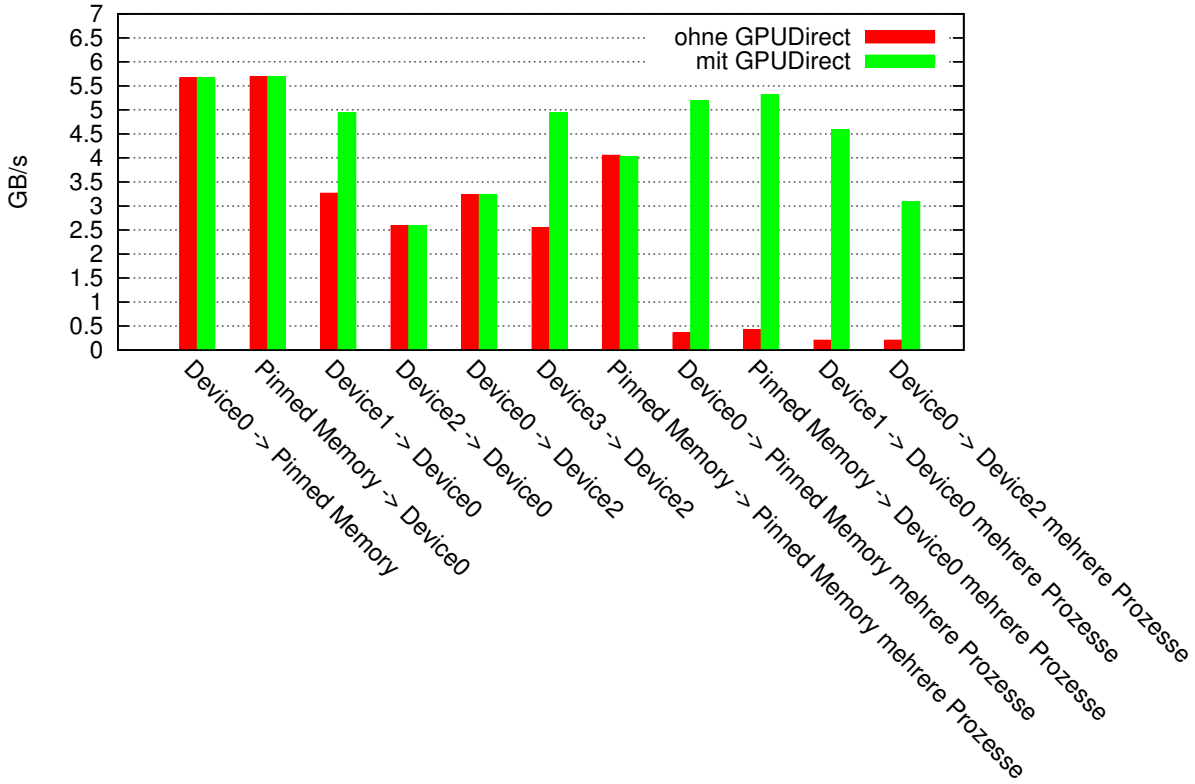

Abbildung 5.2: Bandbreitenmessung der verschiedenen Transferarten

Wie in Grafik 5.2 zu sehen ist unterscheidet sich die Bandbreite eines Kopiervorgangs vom RAM in den Grafikspeicher nicht egal ob GPUDirect verwendet wird. Beide Werte liegen mit $5,6GiB/s$ unter der Angabe von NVIDIA. In diesem Fall bringt GPUDirect auch keinen Vorteil, da GPUDirect

nur die Kommunikation zwischen mehreren Grafikkarten beschleunigt.

Die Kommunikation zwischen mehreren GPUs, die an einem PCIe-Bus angeschlossen sind, kann mit GPUDirect deutlich beschleunigt werden. Dies ist z.B. für "Device1 ->Device0" oder für "Device3 ->Device2" gut zu erkennen. Ohne GPUDirect sind alle Transfers etwa gleich schnell, wobei bezüglich der Quell Grafikkarten unterschieden werden muss. Transfers, die von GPU0 oder GPU1 ausgehen, sind schneller als Transfers, die von GPU2 oder GPU3 ausgehen. Dies ist damit zu erklären, dass die GPUs nur Daten pushen können, d.h. sie können ohne Verwendung der CPU Daten in den Arbeitsspeicher laden aber nicht eigenständig in diesen schreiben. Der Prozess, der die Bandbreitenmessung durchgeführt hat lag auf CPU0, so dass die Grafikkarten direkt in den RAM schreiben konnten. Bei GPU2 und GPU3 musste der Prozessor die Daten von der Grafikkarte lesen und anschließend auf die andere Grafikkarte schreiben. Bei GPU0 und GPU1 konnten die GPUs direkt in den Speicher schreiben und die CPU musste die Daten nur noch auf die Ziel Grafikkarte schreiben.

Beim Kopieren zwischen GPU0 und GPU2 unterscheiden sich die Transfers mit und ohne GPUDirect nicht, da die Grafikkarten an verschiedenen PCIe-Bussen angeschlossen sind, bei diesen Transfers ohnehin über den RAM kopiert werden muss.

Bei der Messung mit mehreren Prozessen ist der Vorsprung von GPUDirect sehr deutlich zu erkennen. Dies liegt vor allem daran, dass mit cudaIPC nicht alle Daten über das Netzwerk versendet werden müssen. Wie schon in Abschnitt 3.2 beschrieben muss nur ein Memory Handle übergeben werden, der dann verwendet wird, um direkt auf den GPU Speicher zuzugreifen. Die Memory Handles können nur verwendet werden, um GPU Speicher zu adressieren, so das dieses Vorgehen beim kopieren von gepinntem auf gepinnten Speicher keinen Vorteil bringt.

Die Bandbreite der Transfers mit GPUDirect über mehrere Prozesse hinweg ist vergleichbar schnell wie die Kommunikation auf nur einem Prozess. Durch die Kommunikation über das Netzwerk bzw. die Umwandlung in und von einem Memory Handle entsteht aber ein gewisser Overhead, so dass die Transfers über mehrere Prozesse etwas langsamer sind als die Transfers auf einer Instanz.

Ohne GPUDirect müssen die Daten wie in Abschnitt 3.1.1 beschrieben deutlich öfter kopiert werden, so dass die Bandbreite beim Kopieren über mehrere Prozesse hinweg nur bei ca $0,2GiB/s$ bis $0,5GiB/s$ liegt.

6 Ein praktisches Beispiel

Als praktisches Beispiel wird eine Matrix Matrix Multiplikation $C = A \cdot B$ verwendet. Diese soll auf mehreren GPUs realisiert werden. Dazu werden die Matrizen auf mehrere Grafikkarten aufgeteilt. Für diese Aufteilung werden alle vier Grafikkarten des Testsystems verwendet.

Die Matrizen werden, wie in Grafik 6.1 zu sehen, in vier Teile aufgeteilt, wobei jeweils ein Teil einer Matrix auf einer GPU liegt. Auch die Ergebnismatrix ist nach dem selben Schema auf die Grafikkarten aufgeteilt. So wird sichergestellt, dass auch mit der Ergbnismatrix ohne Umverteilen weiter gerechnet werden kann.

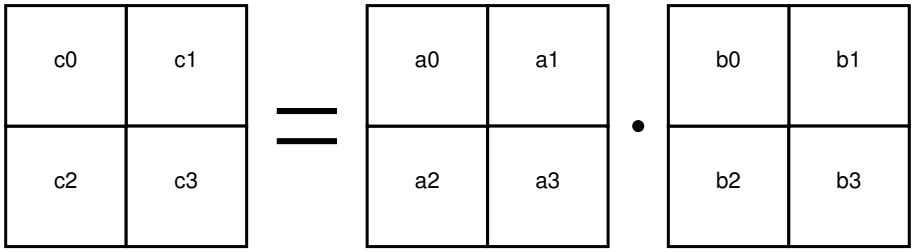

Abbildung 6.1: Schema der Verteilung der Matrix Matrix Multiplikation

6.1 Realisierung des Beispielprogramms

Das Beispielprogramm wurde basierend auf dem Programm zur Messung der Bandbreiten(siehe 5.1) entwickelt. Die Transfers laufen, wie auch bei diesem Programm, völlig transparent ab. Der Speicher muss nur mit Hilfe einer speziellen Klasse angelegt werden. Anschließend kann er auf dem Knoten, auf dem er lokal alloziert wurde, verwendet werden und transparent von einer auf eine andere Instanz übertragen werden.

Die Matrix-Matrix-Multiplikation ist für verschiedene Fälle implementiert. So ist es möglich diese mit einer, zwei oder vier Grafikkarten durchzuführen.

6.1.1 Berechnung auf einer Grafikkarte

Für die Berechnung auf einer Grafikkarte werden Matrizen mit der entsprechenden Größe erzeugt und auf die Grafikkarte kopiert. Da für die Berechnung auf einer Grafikkarte keine Kommunikation notwendig ist werden hierfür auch keine temporären Datenarrays benötigt. Die maximale Größe der Matrizen für die Berechnung auf dem Testsystem berechnet sich wie folgt: Anzahl der speicherbaren Werte unter Verwendung von Float mit eingeschaltetem ECC Speicherschutz, der $\frac{1}{9}$ des gesamten Speichers benötigt: $\frac{6GB}{4B} \cdot \frac{8}{9} = 1333333333 = S$. Die Matrizen müssen komplett auf einer Grafikkarte gespeichert werden. Für die Berechnung sind drei Matrizen (A, B, C) notwendig. Jede Matrix besitzt N^2 Einträge. Als Maximalwert für N ergibt sich also: $\sqrt{\frac{S}{3}} = N \approx 21081$.

Für die Berechnung auf einer Grafikkarte ist keine Kommunikation notwendig, so dass der Aufruf zur Berechnung der Matrix-Matrix-Multiplikation direkt an cuBlas weitergegeben werden kann. Deshalb wird die Berechnung der Matrix Matrix Multiplikation auf einer Grafikkarte als Maßstab für die maximal erreichbare Performanz mit dem Algorithmus verwendet.

6.1.2 Berechnung auf zwei Grafikkarten

Die Berechnung der Matrix-Matrix-Multiplikation auf zwei Grafikkarten erfolgt mit verteilten Matrizen. Die Matrix wird dabei wie in Grafik 6.1 zu sehen in vier Teile aufgeteilt. Wobei jeweils die Teile 0 und 1 auf `GPU0` und die Teile 2 und 3 auf `GPU1` liegen.

Die Berechnung wird nach dem Ergebnis aufgeteilt, so dass jede Grafikkarte die Ergebnisse berechnet, die lokal in ihrem Speicherbereich liegen. Um die Berechnung durchführen zu können benötigt jede Grafikkarte einen Matrixteil der anderen Grafikkarte. Für `GPU0` ergibt sich:

$$\underline{c_0} = \underline{a_0} \cdot \underline{b_0} + \underline{a_1} \cdot b_2$$

$$\underline{c_1} = \underline{a_0} \cdot \underline{b_1} + \underline{a_1} \cdot b_3$$

Die unterstrichenen Matrixteile liegen bereits lokal auf der entsprechenden Grafikkarte. So muss für diese Berechnung nur noch der Matrixteil b_2 und der Matrixteil b_3 von der anderen Grafikkarte geladen werden. Dafür werden zwei temporäre Arrays benötigt, die die Daten aus b_2 und b_3 beinhalten.

Für die zweite Grafikkarte ergibt sich entsprechend:

$$\underline{c_2} = \underline{a_2} \cdot b_0 + \underline{a_3} \cdot \underline{b_2}$$

$$\underline{c_3} = \underline{a_2} \cdot b_1 + \underline{a_3} \cdot \underline{b_3}$$

Auch hier sind die lokalen Anteile unterstrichen. In diesem Fall müssen die Matrixteile b_0 und b_1 kopiert werden. Auch hierfür sind zwei temporäre Arrays notwendig.

Die maximale Größe der Matrix ist in diesem Fall:

$$N = \sqrt{\frac{S}{8}} \cdot 2 \approx 25818$$

Bei der Berechnung der Matrix-Matrix-Multiplikation werden bei der maximalen Größe der Matrix zwei Matrixteile je GPU ausgetauscht: $2 \cdot (\frac{N}{2})^2 \cdot 4B = 2 \cdot (\frac{25818}{2})^2 \cdot 4B \approx 1,271GiB$

Programmablauf

Bei der Berechnung auf zwei GPUs liegen immer drei Matrixteile auf einer GPU. Zwei dieser Matrixteile müssen miteinander multipliziert werden und einer dieser Matrixteile muss mit einem Matrixteil von einer entfernten GPU multipliziert werden. Um das volle Potential der Grafikkarte auszunutzen wird die Berechnung mit den lokalen Matrixteilen sofort gestartet. Während der Teil der lokalen Matrix berechnet wird, wird der benötigte entfernte Matrixteil auf die Grafikkarte kopiert. Nachdem die Berechnung der lokalen Teile erfolgt ist, wird auf die Beendigung des Transfers gewartet. Wenn der Transfer beendet ist wird die noch fehlende Berechnung gestartet.

6.1.3 Berechnung auf vier Grafikkarten

Bei der Verteilung der Berechnung auf vier Grafikkarten erhält jede Matrix jeweils einen Teil einer Matrix. Der Teil M einer Matrix liegt dabei auf der GPU M.

Nach der Verteilung des Speicherorts der Matrizen lässt sich auch die Berechnung aufteilen. Dazu berechnet jede Grafikkarte die Daten für den lokalen Anteil der Ergebnismatrix. Diese Berechnung lässt sich, wenn die Matrizen auf vier GPUs aufgeteilt werden, auf zwei Berechnungen aufteilen. `GPU0` berechnet:

$$\underline{c_0} = \underline{a_0} \cdot \underline{b_0} + a_1 \cdot b_2$$

Die Daten, die bei dieser Berechnung lokal vorliegen sind unterstrichen. Für den Fall von `GPU0` kann also erst kommuniziert werden, anschließend, während auf die Beendigung des Transfers gewartet wird, kann die Berechnung der lokalen Daten erfolgen. Dieses Muster lässt sich auch bei `GPU3` wiederfinden:

$$\underline{c_3} = a_2 \cdot b_1 + \underline{a_3} \cdot \underline{b_3}$$

Die Berechnung der Teile c_1 und c_2 unterscheidet sich dabei von diesem Muster:

$$\underline{c_1} = a_0 \cdot \underline{b_1} + \underline{a_1} \cdot b_3$$

$$\underline{c_2} = \underline{a_2} \cdot b_0 + a_3 \cdot \underline{b_2}$$

Hierbei muss für jede der beiden Berechnungen die Kommunikation von einem Datensatz erfolgen. Bei dem verwendeten Testsystem muss, da immer 2 Grafikkarten am selben PCIe Bus angeschlossen sind, immer ein Datensatz von einer Grafikkarte geladen werden, die am selben PCIe Bus angebunden ist, und ein Datensatz von einer Grafikkarte am anderen PCIe Bus. Die Daten von der Grafikkarte am anderen PCIe Bus müssen zwingend über den Hostspeicher kopiert werden.

Die maximale Größe der Matrix ist in diesem Fall:

$$N = \sqrt{\frac{S}{5}} \cdot 2 \approx 32658$$

Bei der Berechnung der Matrix Matrix Multiplikation werden bei der maximalen Größe der Matrix zwei Matrixteile je GPU ausgetauscht: $2 \cdot (\frac{N}{2})^2 \cdot 4B = 2 \cdot (\frac{32658}{2})^2 \cdot 4B \approx 2,034GiB$

Programmablauf

Bei der Berechnung mit vier Grafikkarten kommt zusätzlich dazu, dass die Grafikkarten auf zwei PCIe-Busse aufgeteilt sind. Jeweils ein Teil der zu empfangenden Daten liegt auf einer Grafikkarte, die an einem anderen PCIe-Bus angeschlossen ist, d.h. jede Grafikkarte muss einen langsamen Transfer von bzw. zu einer Grafikkarte auf einem entfernten PCIe-Bus durchführen und einen schnellen Transfer auf dem lokalen PCIe-Bus.

Um auch in diesem Fall die Transferzeit möglichst gut hinter den Berechnungzeiten verstecken zu können, muss mit einer Berechnung begonnen werden. Da wie oben gezeigt nicht auf jeder Grafikkarte genug Matrixteile vorhanden sind, um eine Matrix-Matrix-Multiplikation durchzuführen, können nur zwei Grafikkarten simultan rechnen. Wenn zu Anfang bereits zwei Grafikkarten gerechnet haben, müssen später die anderen beiden Grafikkarten rechnen. So wird auf jeden Fall die doppelte Berechnungszeit für diese Matrixteile benötigt, da die Berechnung zweimal gestartet wird.

Um die Transferzeiten effektiv verstecken zu können muss also schon zu Anfang ein Transfer ausgeführt werden. Deshalb tauschen zu Anfang des Algorithmus die Grafikkarten die Teile aus, die auf der anderen Grafikkarte am selben PCIe Bus gespeichert sind.

Anschließend wird die Berechnung einer Matrix-Matrix-Multiplikation gestartet. Während dieser

Berechnung werden die Daten ausgetauscht, die auf dem anderen PCIe-Bus liegen.

Wenn die Berechnung und der Transfer der Daten beendet sind, wird die Berechnung der neu erhaltenen Daten durchgeführt. Anschließend ist die Berechnung der Matrix abgeschlossen und jeder Teil der Ergebnismatrix liegt auf der entsprechenden Grafikkarte vor.

6.2 Modifikationen des Beispielprogramms

6.2.1 GPUDirect

Um GPUDirect in das Projekt zu integrieren, müssen Daten von einer auf die andere GPU mit einer speziellen CUDA Funktion kopiert werden. Bevor der Kopiervorgang gestartet werden kann, muss der direkte Zugriff auf den Speicher der anderen Grafikkarte aktiviert werden. Anschließend können die Daten mit einem Aufruf der Funktion zum Kopieren der Daten von der Quellgrafikkarte auf die Zielgrafikkarte direkt übertragen werden.

Um GPUDirect transparent in das Programm zu integrieren, müssen alle Transfers in eine eigene Funktion ausgelagert werden. In dieser Funktion kann auf einfache Weise GPUDirect integriert werden. Dazu muss vor dem eigentlichen Kopieren festgestellt werden, auf welcher Grafikkarte der zu kopierende Speicher liegt. Anschließend kann der direkte Zugriff auf diese Grafikkarte aktiviert werden. Nachdem der Zugriff aktiviert ist, kann der eigentliche Kopiervorgang mit GPUDirect stattfinden. [1][2]

6.2.2 cudaIPC

Die Integration von cudaIPC gestaltet sich als etwas schwieriger. Um cudaIPC nutzen zu können, muss zum einen festgestellt werden auf welchem Knoten die verschiedenen Instanzen liegen, da cudaIPC nur auf einem Knoten lokal funktioniert. Es muss also zur Laufzeit abgefragt werden können, ob eine andere Instanz auf dem lokalen Knoten liegt. Weiterhin muss festgestellt werden können, ob der Quell- oder der Zielspeicher auf der Grafikkarte liegt, da (wie bereits in 3.2 beschrieben) mit cudaIPC nur Grafikspeicher verwendet werden kann.

Um diese Bedingungen zu erfüllen, wurde für das Beispielprogramm ein `MPICommunicator` entwickelt, der anhand eines Hashes des Hostnames den lokalen Knoten identifiziert. So wird zu Beginn eines Programms der Hostname ausgelesen und daraus ein Hash gebildet. Dieser Hash hat für jeden beliebigen Hostname die selbe Länge, wodurch die Hostnamen von jeder Instanz zu jeder Instanz mit einer all2all Kommunikation übertragen werden können.

Bei einer all2all Kommunikation kommuniziert jeder Knoten mit jedem Knoten. Jeder Knoten besitzt ein Array mit Daten, das an die anderen Knoten verteilt wird. Dabei wird jeweils der N-te Datensatz an den N-ten Prozess gesendet. In einem weiteren Array werden die empfangen Daten gespeichert. Die Daten von Prozess N werden dabei an der Arrayposition N gespeichert.

So kann zu Anfang des Programms ein Array mit den Informationen aufgebaut werden, welche Instanzen auf dem lokalen Knoten liegen. Mit diesem Array ist es später zur Laufzeit einfach möglich, festzustellen, ob die Quell- bzw. Zielinstanz auf dem lokalen Knoten liegt.

Um festzustellen ob der Speicher auf der entfernten Instanz im Speicher der Grafikkarte oder im Hauptspeicher liegt, gibt es mehrere Möglichkeiten. Entweder muss vor jedem Transfer der Status des zu kopierenden Speichers ausgetauscht werden. Diese Möglichkeit erzeugt aber einen deutlichen Overhead, da die Latenz der Netzwerkverbindung doppelt in einem Transfer enthalten ist.

Eine weitere Möglichkeit ist es, die Daten wo der Speicher liegt in eine Klasse zu kapseln und auf allen Knoten zu speichern. Der Speicher wird jedoch nur auf der Instanz angelegt, auf der er lokal ist. So kann zum einen das Programmieren vereinfacht werden, indem der entfernte Speicher wie lokaler Speicher kopiert werden kann. Zum anderen liegen alle Daten, die für den Transfer notwendig sind, lokal auf der Instanz, so dass ohne vorherige Kommunikation der Transfer sofort gestartet werden kann.

In dem Beispielprogramm wurde die zweite vorgestellte Möglichkeit realisiert. Dazu wurde eine abstrakte Klasse `Memory` eingeführt. Diese Klasse stellt das grundsätzliche Interface für alle verschiedenen Speicherarten bereit (siehe Abbildung 6.2). Von dieser Klasse erben wiederum zwei weitere abstrakte Klassen. Die `LocalMemory` Klasse und die `DistributedMemory` Klasse, die Klassen die von `LocalMemory` erben legen je nach Implementierung einen bestimmten Speicherbereich z.B. gepinnten Speicher oder Grafikkartenspeicher an und geben diesen beim Aufruf des Destruktors wieder frei. Die Klassen die von der `DistributedMemory` Klasse erben speichern alle Attribute des Speichers, z.B. auf welchem Rank liegt der Speicher oder wie groß ist der Speicher. Nur auf dem Knoten, auf dem der Speicher liegen soll, wird ein passender `LocalMemory` angelegt. So besitzt jede Instanz die Daten über den Speicher, der eigentliche Speicher wird aber nur auf einer Instanz angelegt.

Alle Transferoperationen des Beispielprogramms werden mit einer `copy()` Funktion durchgeführt. Diese `copy()` Funktion ist überladen. Eine Variante der `copy()` Funktion erhält zwei `LocalMemory` Objekte. Diese Variante der Funktion kopiert lokal auf der Instanz.

Die zweite Variante der `copy()` Funktion erhält als Eingabedaten zwei `DistributedMemory` Objekte. Diese Variante kopiert von einem `DistributedMemory` Objekt den Speicher auf ein anderes `DistributedMemory` Objekt. Liegen beide Speicher lokal auf der Instanz wird der Aufruf direkt

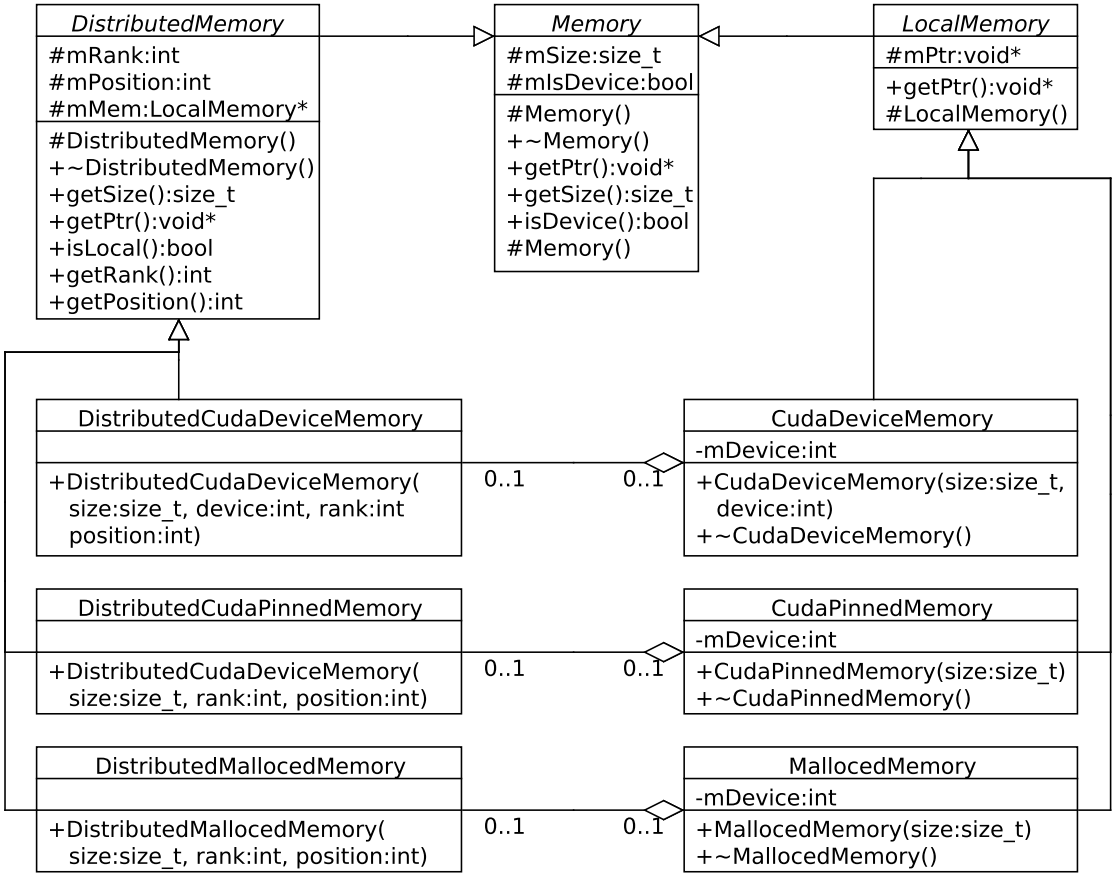

Abbildung 6.2: Klassenhierarchie der Speicherklassen

weitergeleitet. Liegt keiner der beiden Speicher lokal auf der Instanz, springt die Funktion sofort zurück. So wird sichergestellt, dass nur beteiligte Instanzen in dieser Funktion Rechenzeit benötigen. Liegt nur einer der beiden Speicher lokal auf der Instanz, so wird überprüft ob dieser auf der Grafikkarte liegt. Wenn der Quellspeicher auf der Grafikkarte liegt wird ein Memory Handle des Quellspeichers erzeugt. Dieser wird auf die Zielinstanz kopiert. Dort wird aus dem Memory Handle wieder ein Pointer erzeugt. Anschließend werden die Daten von diesem Pointer auf den Zielspeicher kopiert. Anschließend wird auf beiden Instanzen der Memory Handle wieder freigegeben.

Liegt der Quellspeicher nicht auf der Grafikkarte, jedoch der Zielspeicher, so erzeugt die Zielinstanz einen Memory Handle und versendet diesen an die Quellinstanz. Die Quellinstanz kopiert dann die Daten auf den daraus erzeugten Pointer bevor dieser wieder freigegeben wird.

Wenn keiner der Speicher auf der Grafikkarte liegt, so ist auch kein cudaIPC nutzbar, so dass in diesem Fall direkt per MPI kopiert wird.

6.3 Performanzmessungen anhand des Beispielprogramms

Um die Performanz des Beispielprogramms zu testen wurde die reine Berechnungszeit der General Matrix Multiply (GEMM) Operation gemessen. Um sicherzugehen, dass nicht ein Prozess eine kürzere Zeit misst und somit eine höhere Performanz werden alle Prozesse sowohl vor als auch nach der Berechnung synchronisiert. Die gemessene Zeit beinhaltet nicht die Transferzeit vom Host auf das Device. Sie beinhaltet jedoch die Zeit, die die Grafikkarten untereinander zum kommunizieren benötigen, also die gesamte Zeit, die vom Aufruf der Berechnung bis zu deren Fertigstellung auf allen Knoten vergeht.

6.3.1 ein Prozess

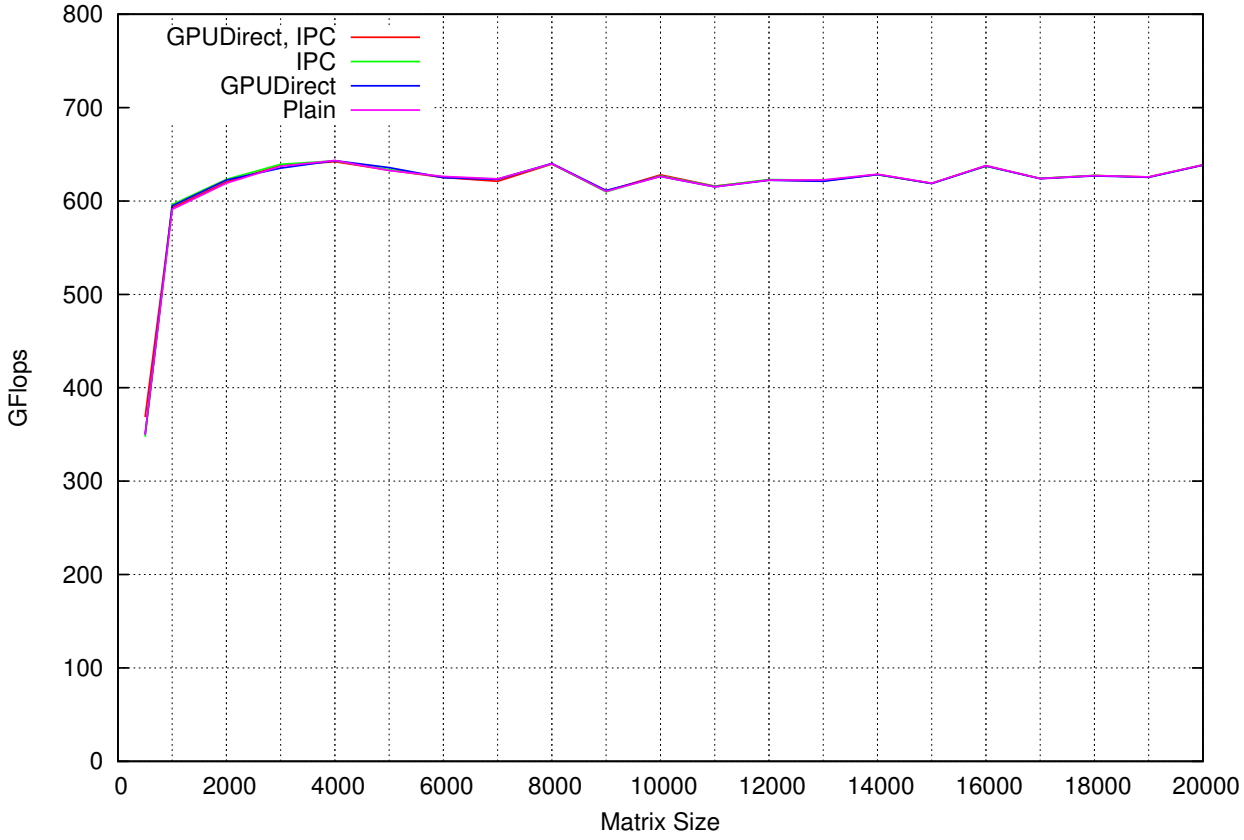

Abbildung 6.3: Performanzmessung mit verschiedenen Transferarten 1 Prozess

Wie in Grafik 6.3 zu sehen, unterscheidet sich die Performanz unter Verwendung von nur einem Prozess nicht, egal ob GPUDirect bzw. cudaIPC verwendet wird. Dieses Verhalten lässt sich dadurch erklären, dass ein Prozess auch nur eine Grafikkarte verwendet. Bei der Verwendung von nur einer

Grafikkarte ist keine Kommunikation notwendig. Es kann also auch keine Kommunikation mit GPUDirect oder cudaIPC beschleunigt werden.

Für die Ergebnisse in Grafik 6.3 wurden Messungen bei einer Matrixgröße von 500 und Matrixgrößen zwischen 1000 und 20000 in 1000er Schritten aufgenommen. Um auch die folgenden Messungen erklären zu können wurde nochmal eine detaillierte Messreihe im Bereich von 11000 bis 11050 in einer Schritten durchgeführt.

Wie in Grafik 6.4 zu sehen lassen sich auch bei kleinen Unterschieden der Matrixgröße deutliche Abweichungen der Performanz feststellen. So ist das Programm bzw. der GEMM Kernel von NVIDIA für ungerade Größen am langsamsten. Für Vielfache von 2 ist das Programm schon etwas schneller. Bei Vielfachen von 4 ist nochmals eine Steigerung zu sehen. Eine weitere Steigerung der Performanz ist wieder jeweils bei Vielfachen von 8, 16 und 32 zu sehen.

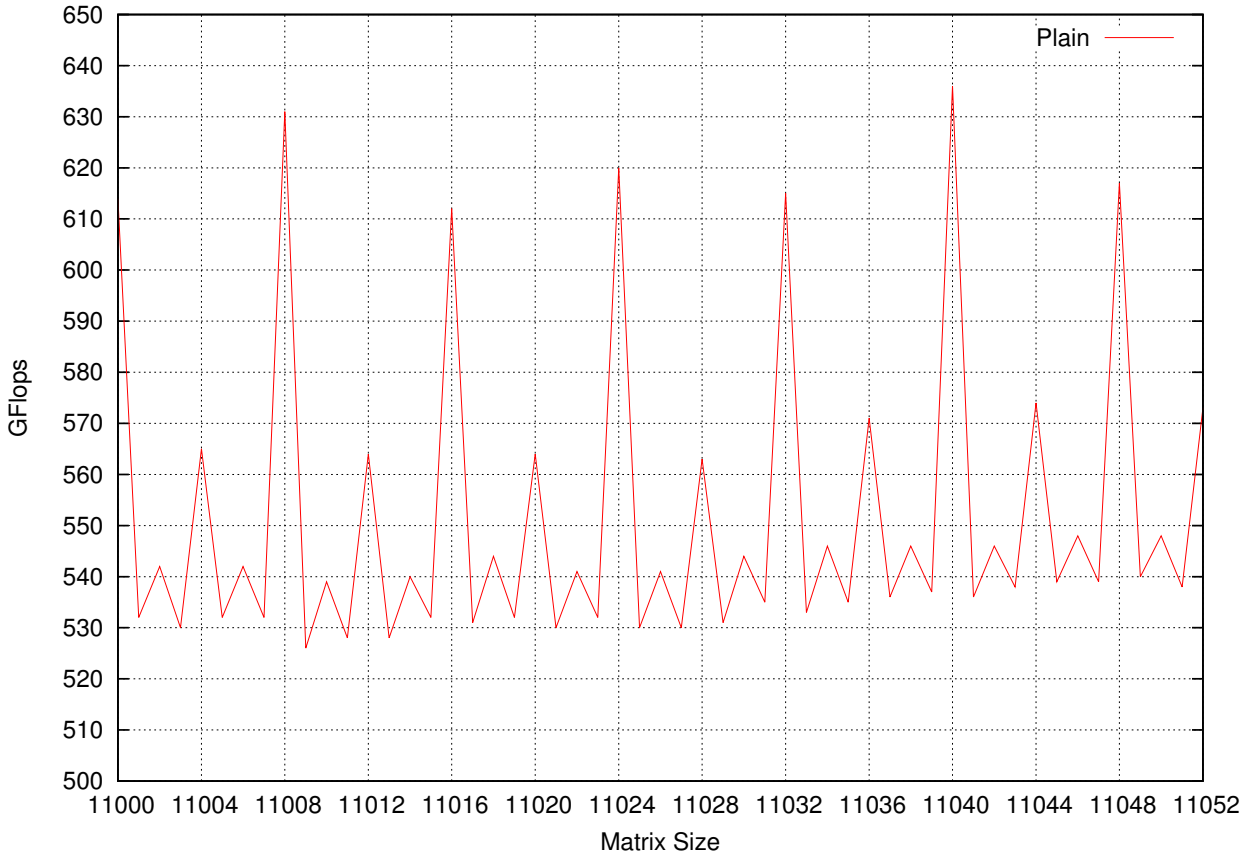

Abbildung 6.4: detaillierte Performanzmessung mit verschiedenen Transferarten 1 Prozess

Diese Unterschiede der Performanz lassen sich durch Konflikte beim Zugriff auf die Speicherbänke der Grafikkarten erklären. Wenn mehrere Threads des gleichen Warps gleichzeitig auf unterschiedliche Werte zugreifen, die auf einer Speicherbank liegen, so müssen diese beiden Zugriffe serialisiert

werden. Dies hat zur Folge, dass der Zugriff auf diesen Wert länger dauert. Jeder Warp besitzt nur einen Instruction Pointer, so dass alle Threads auf die Beendigung des Transfers warten müssen, bevor das Programm weiter ausgeführt werden kann.

6.3.2 zwei Prozesse

Wie in Grafik 6.5 zu sehen ist, steigt die Performanz an, bis sie schließlich gegen einen Maximalwert tendiert. Bei der Berechnung mit zwei Prozessen kann der größte Speedup durch GPUDirect erwartet werden, da hier nur zwei Grafikkarten, die am selben PCIe Bus angeschlossen sind, verwendet werden. Deshalb können alle Transfers mit GPUDirect beschleunigt werden. Der Speedup von cudaIPC sollte etwas geringer als bei der Variante mit vier Prozessen ausfallen, da bei zwei Prozessen weniger Daten übertragen werden müssen und sich somit die Verbesserung der Transfergeschwindigkeit weniger auf den Gesamtprozess auswirkt.

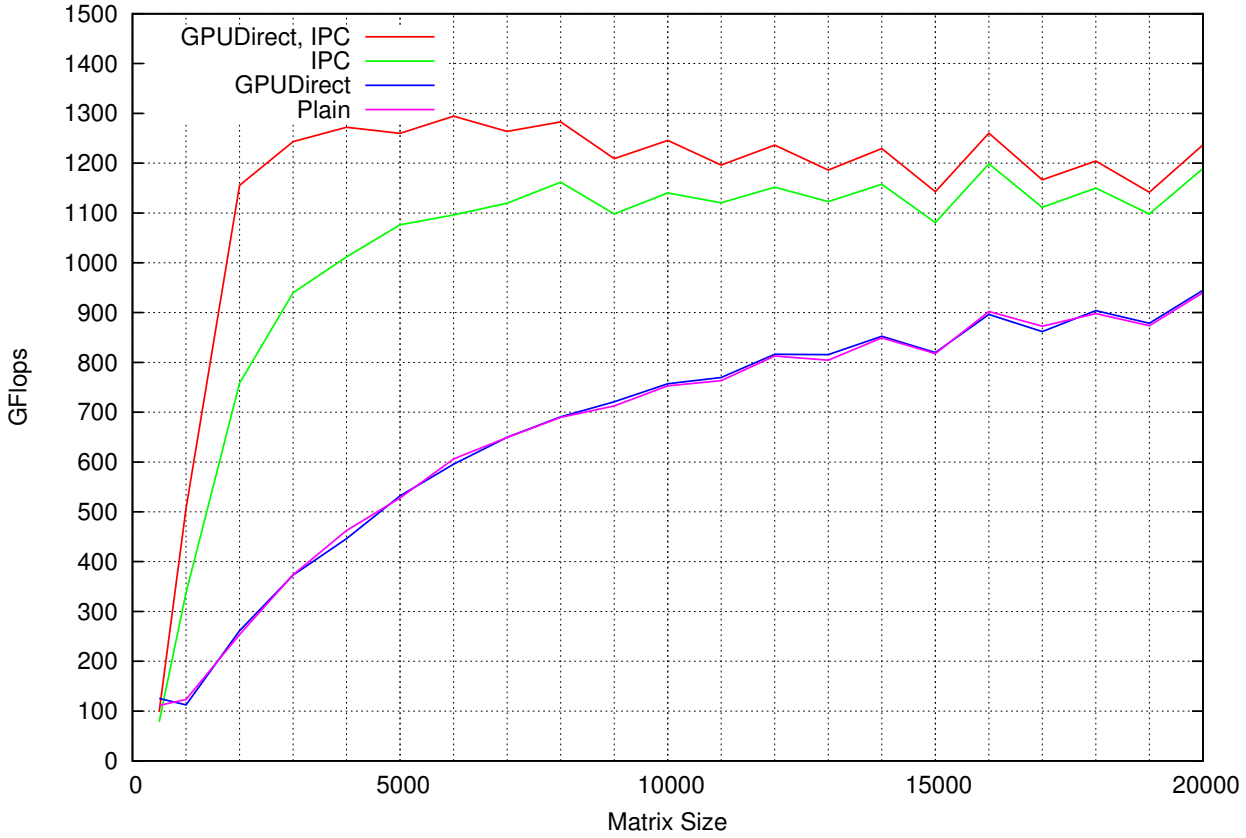

Abbildung 6.5: Performanzmessung mit verschiedenen Transferarten 2 Prozesse

Die Performanzverläufe der Programme ohne cudaIPC sind sehr ähnlich. Ohne cudaIPC kann der

Transfer von einer Grafikkarte zur anderen Grafikkarte nicht direkt durchgeführt werden. Die Daten müssen erst auf den Host kopiert werden um anschließend über das Netzwerk versendet zu werden und schließlich wieder auf die Grafikkarte kopiert werden zu können. Bei diesem Vorgang kann keine Verbesserung der Geschwindigkeit durch GPUDirect erwartet werden. Die beiden Programme haben somit vergleichbare Performanz. Die kleinen Abweichungen der beiden Graphen sind durch die Messungenauigkeit bzw. durch die Arbeit des Schedulers auf der Grafikkarte zu erklären.

Mit der Verwendung von cudaIPC kann die Berechnung deutlich beschleunigt werden. Anfangs kann durch cudaIPC ein sehr hoher Geschwindigkeitsvorteil von bis zu 118% bei einer Matrixgröße von 4000×4000 erreicht werden. Der Vorteil, den cudaIPC bietet, schrumpft aber mit der Größe des Problems. So liegt der Geschwindigkeitsvorteil bei einer Matrixgröße von 20000×20000 nur noch bei 25%. Dies ist dadurch zu erklären, dass bei jeder Matrix-Matrix-Multiplikation $2N^3$ Berechnungen ausgeführt werden müssen. Jeder Prozess muss aber nur, wie bereits in 6.1.2 erläutert, zwei Matrixteile mit jeweils N^2 Werten senden und empfangen. Die Zeit, die benötigt wird, um die benötigten Daten auszutauschen wächst also quadratisch. Die Anzahl der Berechnungen und damit die erforderliche Rechenzeit wächst aber kubisch. So ist es immer besser möglich auch die langsameren Transfers ohne cudaIPC schon während der Berechnung der Werte abzuschließen. Deshalb nähert sich der Graph des Programms ohne cudaIPC immer näher an die Graphen der Programme mit cudaIPC an.

Der Performanzvorteil, der durch GPUDirect erreicht wird, ist ähnlich dem Vorteil, der durch cudaIPC erreicht wird. Auch dieser Vorteil ist bei kleinen Problemgrößen stärker ausgeprägt. So kann durch die Verwendung von GPUDirect bei einer Problemgröße von 4000×4000 ein Performanzgewinn von 25% gegenüber der Variante mit cudaIPC erreicht werden. Bei einer Problemgröße von 20000×20000, bei der der Anteil der Berechnungszeit überwiegt, wird nur noch ein Performanzgewinn von 3% erreicht.

Ab einer Matrixgröße von etwa 8000×8000 wird jeweils abwechselnd eine schnelle und anschließend wieder eine langsamere Ausführung gemessen. Wie bereits in Kapitel 6 beschrieben, wird die Matrix in vier Teile aufgeteilt und anschließend die vier Teilmatrizen berechnet. Die eigentliche Matrix-Matrix-Multiplikation wird nur auf den Teilmatrizen durchgeführt. Die Matrixgröße viertelt sich also für die eigentliche Rechenoperation. Bei der Berechnung einer 8000×8000 Matrix werden die Berechnungen also mit Matrizen der Größe 4000×4000 durchgeführt. Wie bereits in Grafik 6.4 dargestellt, ist die Berechnung der Matrix Multiplikation für Größen, die ein vielfaches von 8 sind deutlich schneller. Bei den Messungen wird die Größe der Matrix für den nächsten Messpunkt jeweils um 1000 erhöht. Die Matrizen, mit denen gerechnet wird, vergrößern sich also immer um 500. Da 500 kein Vielfaches von 8 ist, ist die Berechnung für jeden Wert für den giltMatrixbreite $N = n \cdot 1000 + 500$ langsamer. Für die anderen Werte gilt dann $N = n \cdot 1000$. Da 1000 ein Vielfaches von 8 ist ist auch $n \cdot 1000$ ein Vielfaches von 8 und kann somit schneller abgearbeitet werden.

6.3.3 vier Prozesse

Wie in Grafik 6.6 zu sehen verhalten sich die Graphen mit vier Prozessen ähnlich wie mit zwei Prozessen. Mit vier Prozessen sollte sich gegenüber der Berechnung der Einfluss von cudaIPC auf die Gesamtperformance vergrößern, da mehr Daten kopiert werden müssen. Weiterhin kann der erste Transfer nicht mit einer Berechnung überlagert werden, so dass die Beschleunigung des Transfers einen deutlicheren Performanzgewinn bringen sollte.

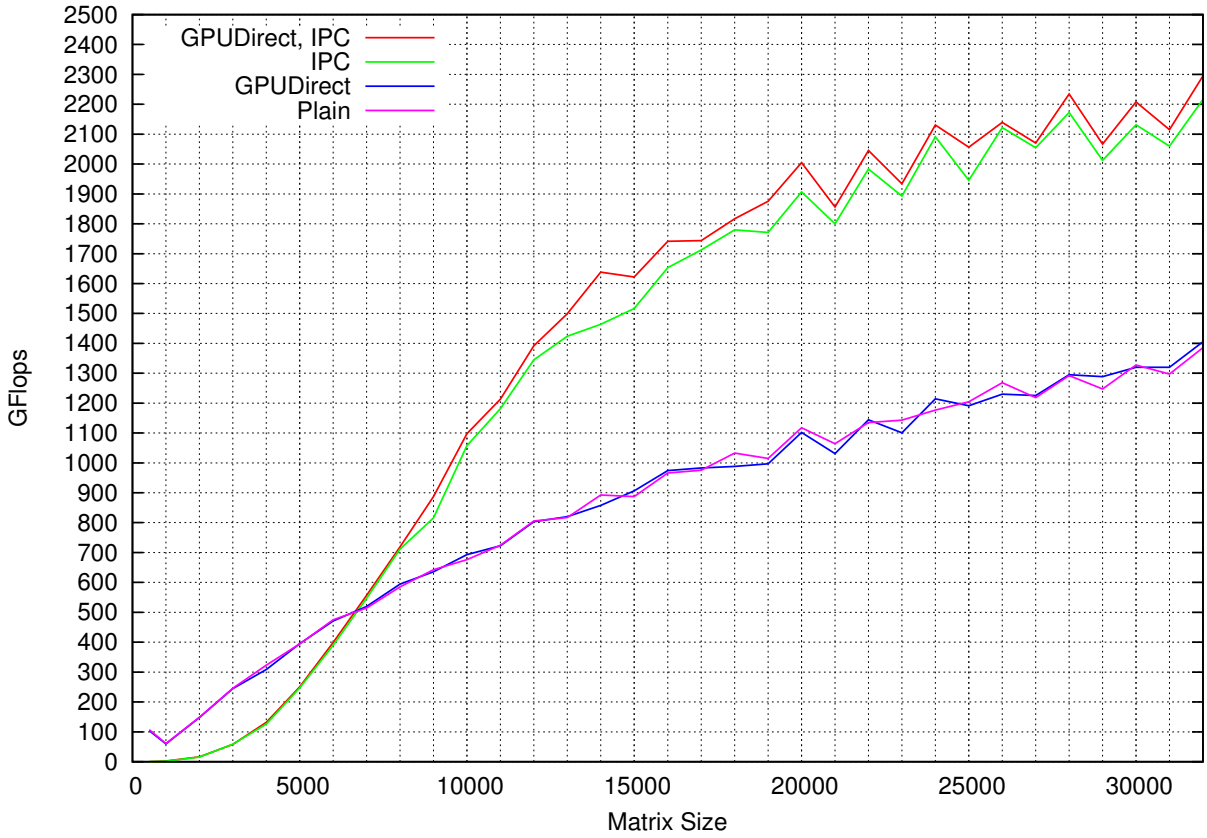

Abbildung 6.6: Performanzmessung mit verschiedenen Transferarten 4 Prozesse

Bei kleinen Matrizen ($N < 8000$) sind die beiden Varianten mit IPC langsamer, da hier ein Memory Handle erzeugt werden muss. Anschließend muss dieser Memory Handle auf die andere Instanz übertragen werden. Dort wird er wieder dereferenziert. Erst jetzt kann der Kopiervorgang gestartet werden. Wie bereits in Abschnitt 6.1.3 beschrieben wird schon vor der ersten Berechnung ein Transfer ausgeführt, so dass der erste Transfer nicht versteckt werden kann.

Besonders für kleine Datenmengen lohnt sich die Übertragung mit cudaIPC nicht, da die beiden kommunizierenden Prozesse mehrfach aufeinander warten müssen. So fließt die Latenz des Netz-

werkes mehrmals in die Transferzeit ein. Bei der Kommunikation von vier GPUs müssen auch vier Memory Handles erzeugt werden. Jeweils einer auf jeder GPU. Nachdem über cudaIPC kommuniziert wurde muss der Memory Handle wieder freigegeben werden. Deshalb müssen beide Prozesse auf die Fertigstellung des Kopiervorgangs warten. Die Erzeugung eines Pointers aus dem Memory Handle ist aber recht teuer. Die Erzeugung dauert etwa zwischen $500ms$ und $600ms$. Wenn alle vier Prozesse kommunizieren sind immer zwei Prozesse mit einem Kopiervorgang blockiert, da der empfangende Prozess auch auf die Beendigung des Kopiervorgangs warten muss um den Memory Handle wieder freigeben zu können. Bei vier Kopiervorgängen muss jeder Prozess also einen Kopiervorgang selbst durchführen und auf einen Kopiervorgang warten. Dadurch ergibt sich eine Verzögerung durch diese Kommunikation von mindestens $1000ms$ bis $1200ms$. Bei kleinen Matrizen lohnt sich der Overhead durch cudaIPC also nicht. Ab einer Größe von ca 8000×8000 wird der gewöhnliche Transfer über das Netzwerk so langsam, dass es sich lohnt die Zeit für die Erzeugung der Memory Handles zu investieren.

Diese Verzögerung tritt auch bei der Berechnung mit zwei Prozessen auf. Bei der Berechnung mit zwei Prozessen kann aber schon der erste Transfer während der Berechnung durchgeführt werden, so dass hier der Overhead durch cudaIPC nicht auffällt.

Der Performanzgewinn durch cudaIPC ist wie auch in dem letzten Beispiel mit zwei Prozessen sehr groß. Der erste Transfer kann bei vier Prozessen nicht versteckt werden und fließt somit unmittelbar in die Laufzeit mit ein. Weiterhin müssen bei der Berechnung mit vier Prozessen auch mehr Daten ausgetauscht werden als bei der Berechnung mit zwei Prozessen. Bei sehr großen Problemen kann mit der Verwendung von cudaIPC eine deutliche Performanzsteigerung festgestellt werden. Wie in der Grafik 6.6 zu sehen ist, werden bei einer Matrixgröße von 32000×32000 ohne cudaIPC etwa 1400 GFlops erreicht, mit cudaIPC werden bis zu 2200GFlops erreicht. Dies entspricht einer Performanzsteigerung von etwa 57%.

Wenn zusätzlich GPUDirect verwendet wird, kann die Performanz auf 2300GFlops erhöht werden. Gegenüber der Variante mit cudaIPC ist dies nur eine Steigerung von ca. 4%, gegenüber der Variante ohne cudaIPC kann aber insgesamt eine Performanzsteigerung von ca. 65% erreicht werden. In diesem Fall kann auch durch GPUDirect keine große Performanzsteigerung erwartet werden, da die Hälfte aller Transfers über einen PCIe Bus hinausgeht.

6.3.4 Skalierbarkeit

Um die Skalierbarkeit des implementierten Algorithmus zu testen, wurde ein Problem der Größe 20000×20000 erzeugt. Dieses Problem wurde mit allen Varianten des Programms gerechnet und jeweils die Zeit gemessen.

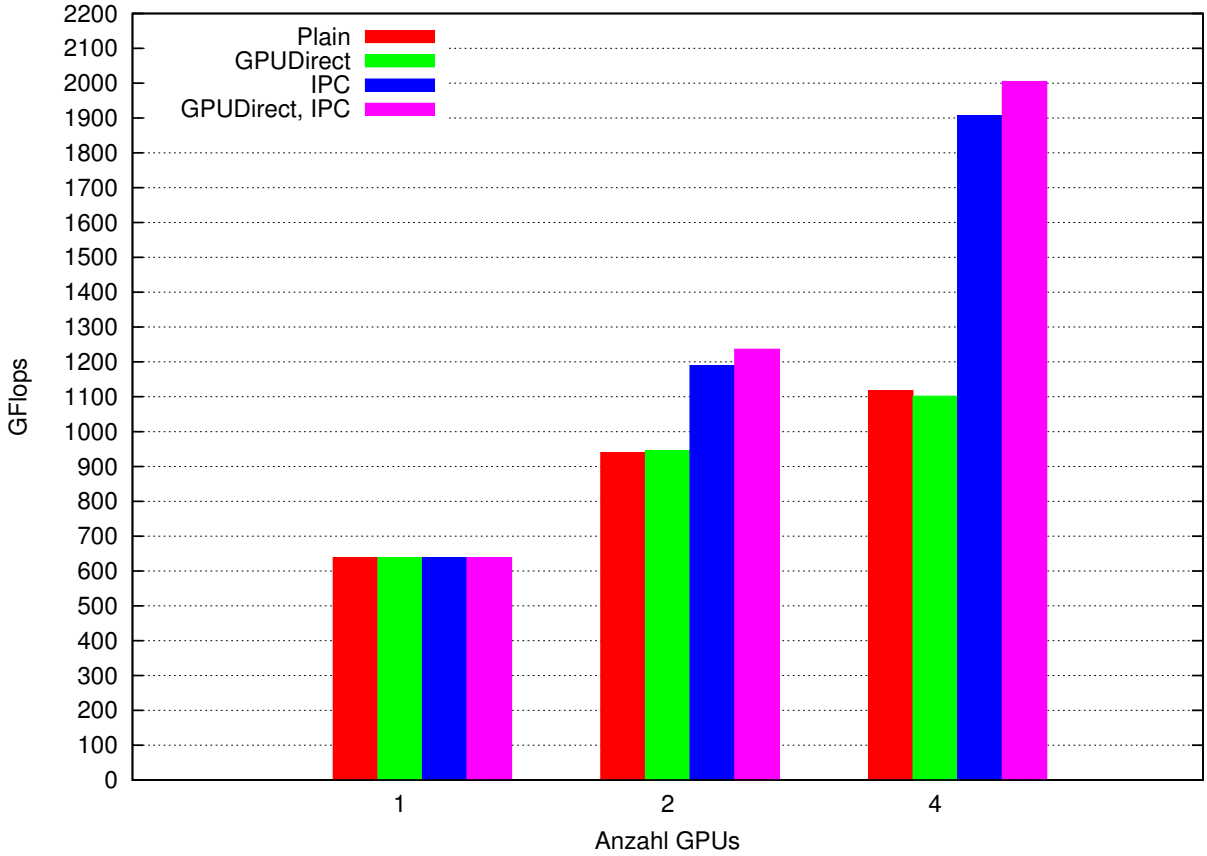

Abbildung 6.7: Performanzmessung mit verschiedenen Transferarten bei einer Matrixgröße von 20000×20000

Wie in Grafik 6.7 zu sehen kann mit dem Beispielprogramm mit mehreren Grafikkarten eine höhere Performanz erreicht werden, als mit einer einzelnen Grafikkarte.

Ohne cudaIPC kann durch die Erhöhung der Rechenresourcen nur ein kleiner Geschwindigkeitsvorteil erreicht werden. Mit der Erhöhung der Rechenresourcen steigert sich auch die Menge der zu kommunizierenden Daten. Wenn diese Datenübertragung so langsam abläuft, dass mit der Berechnung zu lange auf die Fertigstellung des Kopiervorgangs gewartet werden muss, verschlechtert sich die Performanz, die das System erreichen kann. Mit der doppelten Rechenleistung kann in diesem Fall also nicht die doppelte Performanz erreicht werden.

Dieses Verhalten ist auch bei den Varianten ohne cudaIPC zu erkennen. Hier kann mit der doppelten Rechenleistung nur ein Speedup von $1,5$ erreicht werden. Ähnlich kann mit der vierfachen Leistung nur ein Speedup von $1,74$ erreicht werden.

Unter Verwendung von cudaIPC kann eine deutlich bessere Skalierung erreicht werden. So führt

die doppelte Rechenleistung zu einem Speedup von $1,86$. Bei vier Prozessen kann ein Speedup von $2,98$ erreicht werden.

Durch GPUDirect kann diese Skalierung nochmals verbessert werden. So kann unter Verwendung von GPUDirect bei der doppelten Rechenleistung ein Speedup von $1,93$ festgestellt werden. Bei der Vervierfachung der Rechenleistung kann ein Speedup von $3,13$ erreicht werden.

Wie in Grafik 6.6 zu sehen ist konvergiert die Performanz der Berechnung mit vier Grafikkarten unter Verwendung von GPUDirect erst bei sehr großen Problemen. In den aufgenommenen Werten bis zu einer Problemgröße von 32000×32000 kann noch keine eindeutige Konvergenz festgestellt werden, die Performanz könnte also noch weiter steigen. Dieser Sachverhalt lässt sich leider nicht nachweisen, da (wie in Abschnitt 6.1.3 beschrieben) die maximale Matrixgröße für die Verwendung von vier Grafikkarten bei 32658×32658 liegt.

Betrachtet man die Skalierung anhand der maximal messbaren Matrixgröße, so kann mit cudaIPC mit vier Prozessen eine Performanz von $2215.78 GFlops$ erreicht werden. Dies entspricht einem Speedup von $3,46$. Durch die Verwendung von GPUDirect können sogar $2294.56 GFlops$ erreicht werden, was einem Speedup von $3,59$ entspricht.

7 Konzept zur Integration in LAMA

7.1 GPUDirect

Um Daten in LAMA anzulegen werden Kontexte verwendet. Jeder Kontext repräsentiert eine bestimmte Art von Hardware. So gibt es Kontexte für CUDA Grafikkarten, Host Speicher, gepinnten Hostspeicher usw. Ein `LAMAArray` stellt das Gegenstück zu einem gewöhnlichen Array dar. Im Gegensatz zu einem gewöhnlichen Array kann ein `LAMAArray` auf mehreren Geräten gleichzeitig gespeichert werden. Die Klasse `LAMAArray` übernimmt weiterhin das Management der Daten. So kann auf die Daten eines `LAMAArray` nicht wie auf die Daten eines gewöhnlichen Arrays zugegriffen werden. Um die Daten eines `LAMAArray` zu lesen oder zu schreiben ist ein `LAMAReadAccess` bzw. ein `LAMAWriteAccess` auf einem Kontext nötig. Mithilfe dieser Zugriffsobjekte stellt das `LAMAArray` sicher, dass die Daten auf jedem Kontext, auf dem sie gelesen werden, aktuell sind und keine Zugriffe gleichzeitig auf mehreren Kontexten erfolgen. Die Erzeugung eines `LAMAWriteAccess` auf einem Kontext invalidiert alle anderen Kontexte des `LAMAArray`. Bevor die Daten auf den andern Kontexten wieder verwendet werden können müssen sie auf den entsprechenden Kontext kopiert werden. Dieser Transfer geschieht durch das `LAMAArray` implizit.

Jeder Kontext erbt von der abstrakten Klasse `Context`. Dazu müssen wie in Grafik 7.1 zu sehen einige Methoden überladen werden.

Einige Kontexte können die Daten des anderen Kontexts verwenden, z.B kann der `HostContext` den Speicher verwenden, der von einem `CUDAHostContext` angelegt wurde. Beide Speicher sind im RAM angelegt. der Unterschied besteht darin, dass der Speicher, der mit dem `CUDAHostContext` angelegt wurde, gepinnt ist. Um festzustellen, ob auf Speicher von einem Kontext aus zugegriffen werden kann, kann die Funktion `canUseData` verwendet werden. Ist dies der Fall, so entfällt das Kopieren des Speichers.

Kann ein Kontext die Daten des anderen Kontextes nicht verwenden, so müssen diese kopiert werden. Dazu wird zu erst auf dem Zielkontext mit der Methode `cancpy` abgefragt, ob dieser Kontext von dem Quellkontext auf den Zielkontext kopieren kann. Ist dies der Fall wird die `memcpy` Funktion auf dem Zielkontext verwendet um die Daten zu kopieren. Kann der Zielkontext die Daten

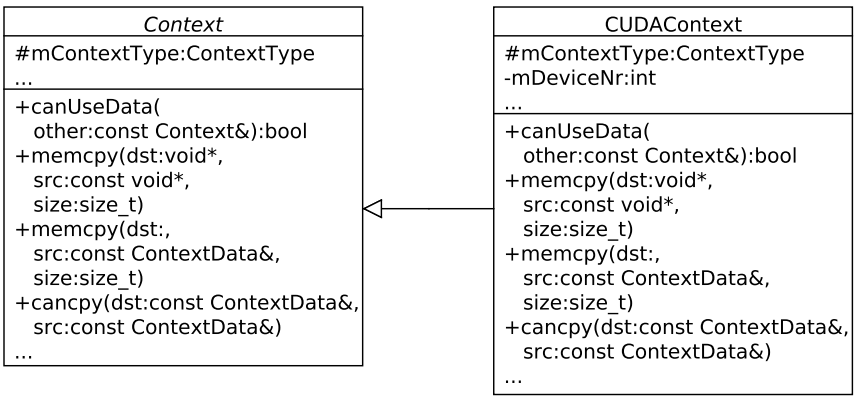

Abbildung 7.1: Die Klasse `CUDAContext` in LAMA

nicht kopieren wird versucht, die Daten mittels des Quellkontextes zu kopieren. Ist auch dies nicht möglich, so werden die Daten zu erst auf den `DefaultHostContext` kopiert und anschließend auf den Zielkontext. So ist gewährleistet, dass es auf jeden Fall möglich ist, von jedem auf jeden Kontext zu kopieren, ohne dass der Benutzer Ausnahmefälle abdecken muss.

Mit dem CUDAContext ist es bisher schon möglich, Daten von einer Grafikkarte auf eine andere Grafikkarte zu kopieren. Diese Funktion muss für die Verwendung von GPUDirect erweitert werden. Um in dieser Funktion GPUDirect zu verwenden, muss vor dem Start des Kopiervorgangs der Zugriff auf die andere Grafikkarte, wie in Abschnitt 6.2.1 beschrieben, erlaubt werden. Dazu muss, da der Transfer sowohl auf der Ziel als auch auf der Quellgrafikkarte ausgeführt werden kann, zu erst festgestellt werden, welches die Grafikkarte ist, auf die der Zugriff erlaubt werden muss. Nachdem dieser Zugriff erlaubt wurde, kann wie gewöhnlich kopiert werden. Es sind somit keine weiteren Änderungen an LAMA nötig.

Um die Abwärtskompatibilität zu gewährleisten, muss sichergestellt werden, dass die Aufrufe, die für GPUDirect nötig sind nur dann mit kompiliert werden, wenn eine ausreichend neue CUDA Version verfügbar ist. Dazu wird das Makro `CUDA_VERSION` verwendet. Damit kann per Preprozessormakro abgefragt werden, ob eine entsprechende CUDA Version vorliegt. Wenn eine ausreichende CUDA Version vorliegt, wird ein Codeblock, der den Code für GPUDirect beinhaltet, in den Code eingefügt und mit kompiliert.

7.2 cudaIPC

Die Kommunikation zwischen mehreren Prozessinstanzen erfolgt in LAMA mittels einer speziellen `Communicator` Klasse. Da sich viele Kommunikationsmuster bei der erneuten Berechnung eines Rechenschrittes wiederholen, wird in LAMA keine direkte Kommunikation zwischen den Knoten durchgeführt. Es wird anhand der Besetzung der Matrizen bzw. Vektoren bestimmt, welche Daten ausgetauscht werden müssen. So können die einmal aufgebauten Kommunikationsmuster z.B. bei iterativen Lösungsverfahren in jeder Iteration verwendet werden.

Durch die Verwendung dieser Kommunikationsmuster kann erreicht werden, dass nur die benötigten Daten ausgetauscht werden und jeder Prozess zu jeder Zeit weiß mit welchem Prozess er kommunizieren muss. Somit kann die Kommunikation durch die Verwendung solcher Kommunikationsmuster beschleunigt werden. Der Nachteil besteht darin, dass die Kommunikationsmuster einmalig aufgebaut werden müssen. Der Aufbau solcher Kommunikationsmuster ist relativ teuer, so dass sich dieser Aufbau nur lohnt, wenn die Rechenoperation, für die das Kommunikationsmuster aufgebaut wurde mehrmals durchgeführt wird. Wenn eine Berechnung mehrmals mit unterschiedlichen Werten durchgeführt wird, lohnt sich der Aufbau eines solchen Kommunikationsmusters, da nicht mehr gesucht werden muss, welcher Prozess welche Werte hat und wann diese gesendet werden.

Die Kommunikationsmuster werden, wie in Grafik 7.2 dargestellt, in einem Objekt der Klasse `CommunicationPlan` gespeichert. Die Klasse `CommunicationPlan` beinhaltet einen Vektor von Instanzen der Struktur `Entry`. Ein `Entry` enthält drei Attribute

- partitionID: die Nummer der entfernten Instanz
- quantity: die Anzahl der zu sendenden Daten
- offset: der Offset innerhalb des zu sendenden Arrays

Für jede Kommunikation gibt es zwei Instanzen der Klasse `CommunicationPlan`. Ein Objekt regelt den Empfang der Daten, das andere Objekt regelt das Senden dieser.

Die eigentliche Kommunikation wird, wie in Abblildung 7.2 zu sehen, von der `Communicator` Klasse durchgeführt. Dazu erhält die Funktion `exchangeByPlan` zwei temporäre Arrays, eines zum Senden und eines zum Empfangen, und zwei Instanzen der Klasse `CommunicationPlan`. Die `Communicator` Klasse führt die komplexeren Kommunikationsfunktionen meist auf die `send` bzw. `receive` Funktionen der Kommunikationsbibliothek zurück.

Die Funktion `exchangeByPlan` gibt es sowohl mit `LAMAArray` als Eingabedaten wie auch direkt mit Pointern auf Daten. Werden diese Funktionen mit Pointern aufgerufen, so liegen alle verwen-

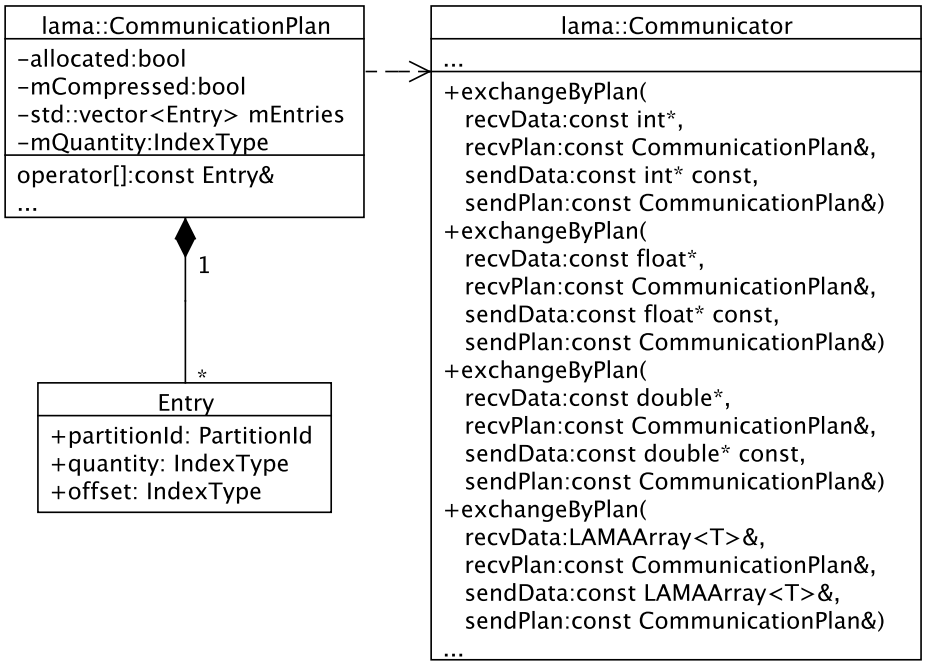

Abbildung 7.2: Die Klasse `CommunicationPlan` in LAMA

deten Werte im RAM. Wird diese Funktion mit `LAMAArray` aufgerufen, so können die Daten auf verschiedenen Kontexten vorliegen. Bisher werden in diesem Fall alle Daten auf den `Context` kopiert, den die aktuell verwendete Implementierung mit der Funktion `getCommunicationContext` zurück gibt. Diese Funktion wurde im Rahmen der Integration einer Schnittstelle für einseitige Kommunikation eingeführt [4]. Im Normalfall liefern alle Kommunikatoren auf diesen Aufruf den `DefaultHostContext` zurück. Wenn dieser Kontext mit einer anderen, hinzuzufügenden, Funktion gesetzt werden kann, so ist es möglich, über das Setzen des `CUDAContext` die Kommunikation mit cudaIPC zu aktivieren.

Bei der Verwendung von cudaIPC bzw. dem `CUDAContext`es als Kommunikationskontext muss sichergestellt werden, dass alle Instanzen den selben Kommunikationskontext verwenden, da es sonst zu einer Blockade der Ausführung kommen kann. Wie in Grafik 7.3 zu sehen ist, wartet Prozess 1 auf N Byte an Daten von Prozess 0. Prozess 0, welcher als Kommunikationskontext den `CUDAContext` verwendet, sendet aber nur 64 Byte an Daten, also einen Memory Handle für cudaIPC. Prozess 0 wird nach dem Senden des Memory Handles wie gewöhnlich fortfahren, wohingegen Prozess 1 weiterhin auf eingehende Daten wartet. Wartet Prozess 0 anschließend auf Prozess 1 z.B. bei einer Synchronisation, so entsteht ein Deadlock.

Leider ist die Verhinderung eines solchen Vorgangs nicht trivial, beide Prozesse müssen vor der

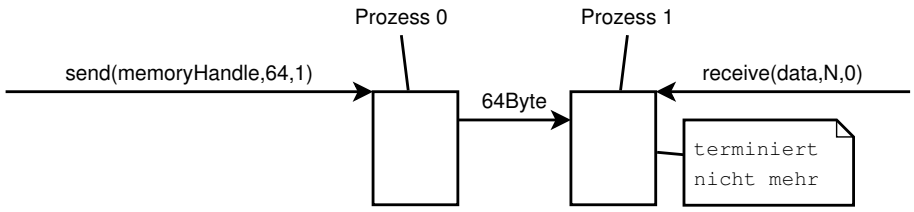

Abbildung 7.3: Verklemmung der Kommunikation bei nutzung von cudaIPC auf nur einer Instanz

Kommunikation festlegen, ob sie mit oder ohne cudaIPC. Dazu muss unmittelbar vor der Kommunikation eine weitere Kommunikation durchgeführt werden. Auf diese muss gewartet werden, bevor die eigentliche Kommunikation beginnen kann. Da diese Überprüfung, vor allem bei beliebig vielen Kommunikationspartnern, nicht performant durchführbar ist, muss der Nutzer dafür sorgen, dass alle Instanzen den selben Kommunikationskontext verwenden.

Die einzige Möglichkeit cudaIPC effektiv in LAMA zu integrieren, ist die Annahme der Voraussetzung, dass alle Instanzen den selben Kommunikationskontext verwenden. Dieses Prinzip widerspricht allerdings dem Prinzip von LAMA, dass heterogene Systeme verwendet werden können. LAMA soll auch auf Rechnersystemen benutzbar sein, von denen nur einige Rechner CUDA fähige Grafikkarten besitzen. Diese Systeme ohne CUDA fähige Grafikkarte können kein cudaIPC verwenden, die Systeme mit einer solchen Grafikkarte hingegen schon. Die Zusammenarbeit dieser Systeme funktioniert bei einer Integration von cudaIPC nicht mehr ohne Einschränkungen. Bei der gemeinsamen Verwendung von Systemen mit und ohne CUDA Grafikkarten muss deshalb auf die Verwendung von cudaIPC verzichtet werden.

Wenn als Kontext zur Kommunikation ein CUDAContext übergeben wird, so müssen alle temporären Arrays auf der Grafikkarte vorliegen. Anschließend muss festgestellt werden, welche der verwendeten Instanzen auf welchem Host liegen bzw. welche der verwendeten Instanzen lokal sind. Nur an diese Instanzen können mit cudaIPC Daten übertragen werden. Zum Kopieren auf die Instanzen müssen die Daten, wie bisher auch, in den RAM kopiert werden und von dort wieder auf die Grafikkarte. In diesem Fall muss ein zusätzliches `LAMAArray` angelegt werden, in das die Daten von den Instanzen auf anderen Knoten kopiert werden. Anschließend müssen diese Daten manuell mit Hilfe der `memcpy` Funktion des `CUDAContext` in das Datenarray auf der Grafikkarte kopiert werden. Dieses zusätzliche Array ist nötig, da ein `LAMAArray` nicht gleichzeitig auf mehreren Kontexten, in diesem Fall auf dem Host und Grafikkarte, bearbeitet werden kann.

Die verschiedenen Funktionalitäten werden in LAMA in verschiedene Bibliotheken aufgeteilt, um LAMA auch auf Systemen verwenden zu können, die einige Features von LAMA nicht unterstützen. So gibt es eine spezielle Bibliothek für die Unterstützung von CUDA und spezielle Bibliotheken für die Kommunikation. Für die Unterstützung von CUDA gibt es nur eine Bibliothek. Für die Kommunikation gibt es mehrere Bibliotheken, die verschiedene Kommunikationsframeworks wie z.B. MPI

oder OpenSHMEM unterstützen. Die Kommunikation über cudaIPC soll mit allen verschiedenen Kommunikationsbibliotheken möglich sein.

Um keine Abhängigkeiten von den Kommunikationsbibliotheken auf die CUDA Bibliothek zu haben, müssen die speziellen Kommunikationsmuster, die für cudaIPC nötig sind, in der CUDA Bibliothek implementiert werden. Durch die Implementierung dieser Kommunikationsmuster innerhalb der CUDA Bibliothek sind diese Kommunikationsmuster auch für andere Beschleunigerkarten oder ähnliche Hardware erweiterbar. Da aber auch die CUDA Bibliothek keine Abhängigkeiten auf die Kommunikationsbibliotheken haben darf, muss das für cudaIPC verwendete Kommunikationsmuster für die allgemeine Form des `Communicators` implementiert werden.

Um alle möglichen Vorgänge modellieren zu können müssen zur `Communicator` Klasse noch `send` und `receive` Funktionen hinzugefügt werden. Diese Methoden sollen Daten direkt senden bzw. empfangen. Dazu müssen sie einen Pointer auf Daten und die Größe der Daten erhalten. Zusätzlich muss der Funktion die Ziel- bzw. Quellinstanz übergeben werden. Mit diesen beiden Funktionen können alle nötigen Vorgänge modelliert werden. Um diese Befehle auch asynchron durchführen zu können sollten jeweils noch asynchrone Funktionen für `send` und `receive` bereitgestellt werden. Mit diesen vier Funktionen des Kommunikators können auch viele der vom Kommunikator zur Verfügung gestellten Funktionen als Standardimplementierung umgesetzt werden. Als Nebeneffekt kann somit auch der Aufwand verringert werden ein neues Kommunikationsframework zu integrieren.

Um nun noch die Kommunikation und die Hardwarebibliotheken vollständig voneinander unabhängig zu machen sollten die Kontexte eigene `send` und `receive` Methoden implementieren. Diese Methoden erhalten noch zusätzlich einen Kommunikator. So kann die Kommunikation auch über Instanzen hinweg über den Kontext abgewickelt werden. Der Kontext an sich implementiert dabei nur die Kommunikationsmuster, die für ihn notwendig sind. Wenn ein Kontext keine besonderen Kommunikationsmuster benötigt, so kann mit einer Standardimplementierung in der `Context` Klasse direkt die `send` bzw. `receive` Funktion des Kommunikators aufgerufen werden. So sind auch keine Änderungen an den bisher bestehenden Kontexten, die keine speziellen Kommunikationen anbieten, nötig.

Der Kommunikator muss dann, um die Beschleunigung durch spezielle Funktionen der Kontexte zu unterstützen statt direkt die Kommunikationsbibliothek aufzurufen, wie in Grafik 7.4 und Grafik 7.5 zu sehen, den Kontext aufrufen, welcher anschließend den Kommunikator verwendet um Aufrufe an die Kommunikationsbibliothek durchzuführen. So kann der Kontext entscheiden, ob die Kommunikation durch spezielle Kommunikationsmuster (wie in Grafik 7.5 zu sehen) beschleunigt wird und die Beschleunigung der Kommunikation für spezielle Hardware wird in die Bibliothek ausgelagert, die für diese Hardware zuständig ist. Dabei besteht keine Abhängigkeit zwischen den

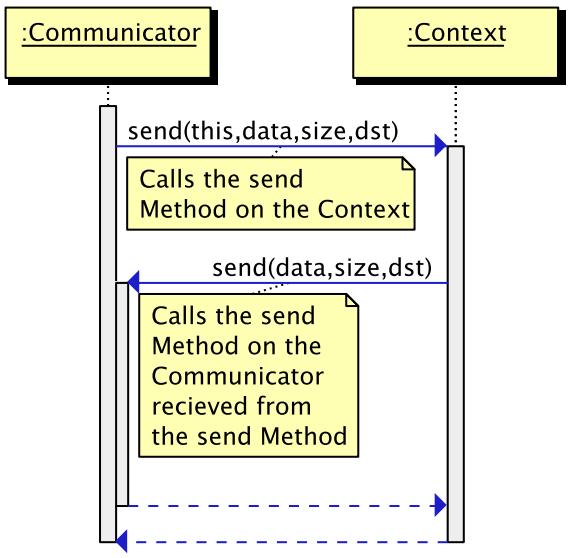

Abbildung 7.4: Kommunikation mit einem Kontext ohne zusätzliche Beschleunigung

Kommunikations- und den Hardwarebibliotheken.

Um die Datenübertragung zu beschleunigen, sollten bei der Kommunikation möglichst wenige Memory Handles erzeugt werden, da dieser Vorgang, wie in Abschnitt 6.3.3 beschrieben, verhältnismäßig lange dauert. Statt dessen sollte, so weit dies möglich ist, einmal ein Memory Handle erzeugt werden, der dann mehrmals mit einem Offset verwendet wird. Die sendende Instanz sollte in diesem Fall zusätzlich zu einem Memory Handle ein Offset an die Zielinstanz übertragen. So kann ein Memory Handle für mehrere zusammenhängende Transfers verwendet werden. Der Memory Handle und der Offset sollten in eine Struktur vereint werden, so dass nur eine Übertragung zur Zielinstanz notwendig ist. So kann der mehrfache Overhead, welcher durch das Netzwerk erzeugt wird, vermieden werden.

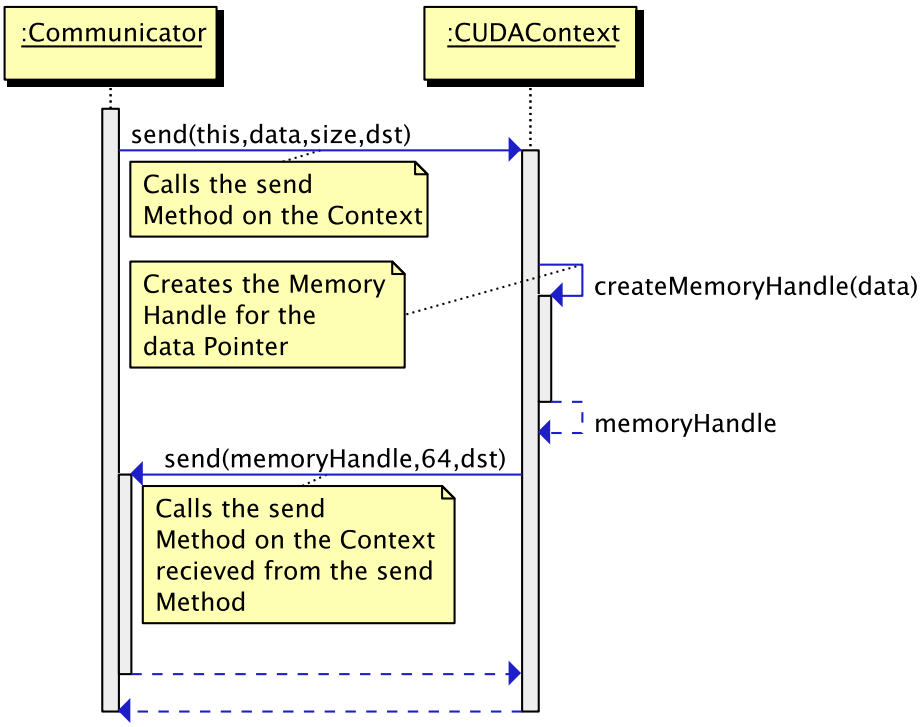

Abbildung 7.5: Kommunikation mit dem CUDAContext und cudaIPC

8 Fazit und Ausblick

Abschließend bleibt zu sagen, dass die Verwendung von GPUDirect und cudaIPC großes Potential hat die Transfers einer Berechnung zu beschleunigen. Der Vorteil von GPUDirect und cudaIPC ist vor allem bei der losgelösten Betrachtung der Transferleistung sehr deutlich.

Bei praktischen Problemen wie der Matrix-Matrix-Multiplikation ist es allerdings deutlich schwieriger von der Beschleunigung der Transferleistung zu profitieren, da die meisten Algorithmen bereits darauf optimiert sind, den Datentransfer möglichst gering zu halten bzw. mit der Berechnung zu überlappen. So kann meist nur ein relativ kleiner Geschwindigkeitsvorteil durch GPUDirect erzielt werden. Der Geschwindigkeitsvorteil, der durch cudaIPC erreicht werden kann, ist dennoch sehr groß, was vor allem daran liegt, dass die Alternative, der Versand der Daten über das Netzwerk, deutlich langsamer ist als der direkte Transfer mit cudaIPC.

Die Integration von GPUDirect in ein bestehendes Projekt is sehr einfach, da hierzu nur, wie in Abschnitt 6.2.1 beschrieben, der entsprechende Zugriff auf den Speicher der anderen Grafikkarte aktiviert werden muss. Deshalb sollte GPUDirect immer dann integriert werden, wenn ein Projekt Daten zwischen mehreren Grafikkarten transferiert.

CudaIPC in ein Projekt zu integrieren gestaltet sich etwas komplexer. Wie in Abschnitt 7.2 beschrieben ist es teilweise sogar gar nicht möglich, cudaIPC effektiv in ein Programm zu integrieren, da beide Kommunikationspartner vorher festlegen müssen wie sie kommunizieren. Wenn nicht jeder Instanz bekannt ist, in welchem Speicher die Daten auf der anderen Instanz liegen, kann cudaIPC nicht ohne Einschränkungen verwendet werden (siehe Abschnitt 7.2).

Deshalb sollte mit einer Integration von cudaIPC in bestehende Projekte noch etwas gewartet werden. Im vierten Quartal 2012 sollen die ersten NVIDIA Tesla Grafikkarten mit Kepler K20 Chip auf dem Markt kommen. Mit diesen Chips ist es, wie in Abschnitt 3.1.3 beschrieben, möglich, Daten direkt aus dem Grafikspeicher heraus über das Netzwerk zu versenden. Durch diese Neuerung sollte die Verwendung von cudaIPC abgelöst werden. Dabei bietet diese Funktion, die Daten direkt aus dem Grafikspeicher heraus versenden zu können, eine deutlich einfacher zu integrierende Möglichkeit, die Kommunikation zwischen mehreren Instanzen zu beschleunigen. Des Weiteren ist es mit dieser Funktion auch möglich, nicht nur die Kommunikation zwischen Instanzen auf dem

selben Knoten zu beschleunigen, sondern auch den Transfer zwischen Instanzen auf verschiedenen Knoten.

Zusammenfassend kann gesagt werden, dass eine Beschleunigung der Berechnung durch GPUDirect und cudaIPC möglich ist. Dabei können gute Ergebnisse und Verbesserungen erreicht werden. Um wirklich von der Verwendung von GPUDirect und cudaIPC profitieren zu können sind aber auch sehr datenintensive Anwendungen nötig. Trotz der Beschleunigung des Transfers durch GPUDirect und cudaIPC ist trotzdem eine Optimierung des eigentlichen Algorithmus notwendig um nennenswerte Performanceverbesserungen zu erreichen.

Literaturverzeichnis

[1] NVIDIA Corporation
NVIDIA CUDA C Programming Guide
Version 5 - NVIDIA Corporation Mai 2012

[2] NVIDIA Corporation
CUDA API REFERENCE MANUAL
Version 5 - NVIDIA Corporation Juli 2012

[3] Jiri Kraus, Malte Förster, Thomas Brandes und Thomas Soddemann
Using LAMA for efficient AMG on hybrid clusters
Computer Science - Research and Development, Online First, 23 Mai 2012

[4] Michael Drost
Entwicklung und Evaluation einer PGAS Kommunikationsschicht für LAMA
DHBW Mosbach 2012

Internetreferenzen

Im Folgenden sind die Quellen aus dem Internet aufgelistet. Um gewährleisten zu können, dass diese Quellen auch dann noch zur Verfügung stehen, wenn der entsprechende Inhalt nicht mehr auf den hier aufgelisteten Servern zu finden sein sollte, befinden sich alle Quellen als pdf auf einer beiliegenden CD.

[5] NVIDIA Corporation
NVIDIA GPUDirect
"Online im Internet" unter `http://developer.nvidia.com/gpudirect`
Stand 05.07.2012

[6] Paulius Micikevicius
Multi-GPU Programming (Vortrag auf der GTC)
"Online im Internet" `http://develper.download.nvidia.com/GTC/PDF/GTC2012/PresentationPDF/S0515-GTC2012-Multi-GPU-Programming.pdf`
Vortrag als Video unter: `http://www.gputechconf.com/gtcnew/on-demand-gtc.php?sessionTopic=&searchByKeyword=S0515&submit=&select=+&sessionEvent=&sessionYear=&sessionFormat=#1451`
Stand 30.7.2012

[7] Axel Koehler
Scalable Cluster Computing with NVIDIA GPUs
"Online im Internet" unter `http://www.hpcadvisorycouncil.com/events/2012/Switzerland-Workshop/Presentations/Day_3/3_NVIDIA.pdf`
Stand 30.07.2012

[8] Joe Chang
Intel Nehalem based systems with QPI
"Online im Internet" unter `http://www.qdpma.com/SystemArchitecture/SystemArchitecture_QPI.html`
Stand 06.09.2012

[9] Carsten Spille
GK110: Big Kepler mit bis zu 2.880 ALUs auf GTC 2012 vorgestellt - Video-Update
"Online im Internet" unter `http://www.pcgameshardware.de/aid,883969/GK110-Big-Kepler-mit-bis-zu-2880-ALUs-auf-GTC-2012-vorgestellt/Grafikkarte/News/`
Stand 16.07.2012

[10] NVIDIA Corporation
Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110
"Online im Internet" unter `www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`
Stand 06.09.2012

[11] NVIDIA Corporation
Grafikprozessor Tesla C2050/C2070
"Online im Internet" unter `http://www.nvidia.de/object/product_tesla_C2050_C2070_de.html`
Stand 30.07.2012

[12] NVIDIA Corporation
TESLA C2050 AND TESLA C2070 COMPUTING PROCESSOR BOARD
"Online im Internet" unter `http://www.nvidia.com/docs/IO/43395/Tesla_C2050_Board_Specification.pdf`
Stand 30.07.2012

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit mit dem Titel

Evaluierung der Performanz von GPUDirect 2.0 auf einem Multi-GPU System

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Sankt Augustin, den 03.07.2012

Michael Drost