TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Betriebssysteme und Kommunikationstechnologien

Seminararbeit

Aufbau eines Prototyps für verteilte CUDA Programmierung

Development of a prototype for distributed CUDA programming

Samuel Dressel

Angewandte Informatik
Vertiefung: Parallelrechner

Matrikel: 59963

2. November 2019

Betreuer/1. Korrektor:
Prof. Dr. Konrad Froitzheim

2. Korrektor:
Dr. rer. nat. Martin Reinhardt

Inhaltsverzeichnis

1. Einleitung **4**

2. Grundlagen **4**
2.1. Message-Passing-Interface (MPI) 4
2.2. CUDA 5
2.3. CUDA und MPI 5

3. Technischer Aufbau und Konfiguration **6**

4. Benchmarking und Test ausgewählter Algorithmen **9**
4.1. Einfache Vektoraddition 9
4.2. Floyd-Warshall-Algorithmus 10

5. Fazit **13**

Anhang **14**

A. Aufbau des Clusters **14**

B. Quellcode Vektoraddition **15**

Literatur **19**

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen Hilfsmittel als die angegebenen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Diese Versicherung bezieht sich auch auf die bildlichen Darstellungen.

2. November 2019 Samuel Dressel

1. Einleitung

2. Grundlagen

2.1. Message-Passing-Interface (MPI)

Das Message-Passing-Interface, kurz MPI, ist eine standartisierte Schnittstelle für die Kommunikation auf verteilten Systemen. Es wird häufig wissenschaftlichen und ingenieurstechnischen Domänen verwendet, um größere Problemstellungen effizienter zu bearbeiten und zu lösen [1]. MPI ist speziell für die Verwendung auf verteilten Systemen mit ebenso verteiltem Speicher konstruiert. Dabei kann jeder Prozessor, der Teil der Bearbeitung eines Workloads ist, nur auf seinen lokalen Speicher zugreifen. Es können jedoch durch Messages Daten von einem lokalen Speicher in den anderen übertragen werden.
MPI ist als Bibliothek implementiert, nicht als Programmiersprache. Dabei existieren Implementierungen für zahlreiche Programmiersprache wie C, C++, Python oder Fortran [2].
Die grundlegendste Art der Kommunikation findet zwischen zwei Prozessen statt. Dabei übermittelt ein Sendeprozess Informationen an einen Empfangsprozess. Prozesse lassen sich au-

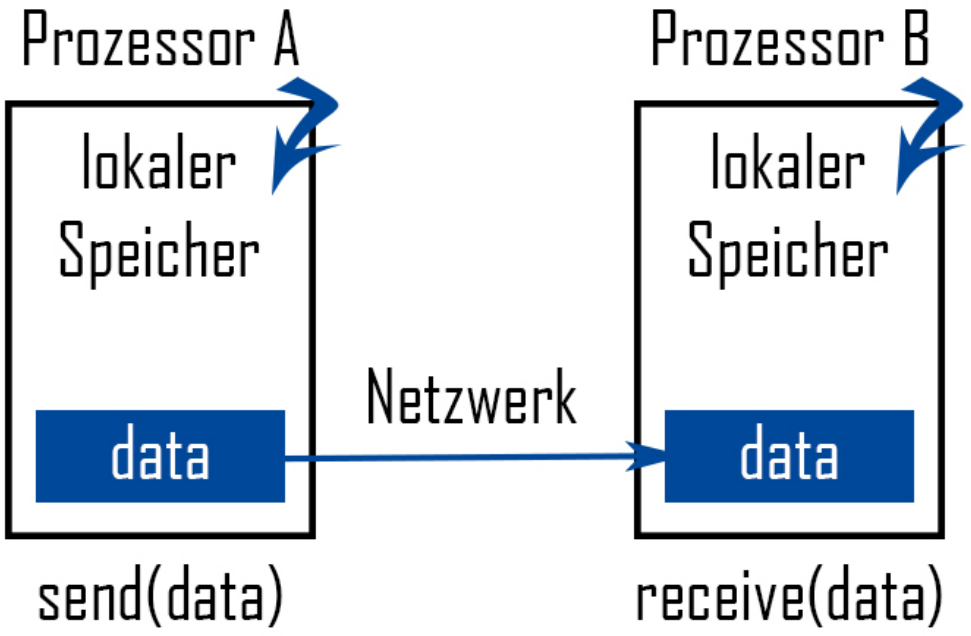

Abb. 1: Grundlegendes Konzept von MPI: Ein Sendeprozess sendet Daten an einen Empfangsprozess über ein Netzwerk. Jeder Prozess kann nur unmittelbar auf seinen zugehörigen lokalen Speicher zugreifen.

ßerdem in Gruppen zusammenfassen, wobei jeder Prozess eine eindeutige Nummer (Rang bzw. Rank) zugeorndet wird. Der Zugriff auf diese Gruppe wird über einen Kommunikator gesteuert. Innerhalb dieser Gruppe können mithilfe einer Boradcast-Operation von einem Masterprozess allen anderen Prozessen Nachrichten gesendet werden. Diese zurückgegebenen Daten werden mithilfe einer Gather-Funktion wieder eingesammelt.
Dieses Konzept wird auch für den im Rahmen dieser Arbeit erstellten Minicluster genutzt: Der Workload wird von dem Masterknoten definiert und dann über das Message-Passing-Interface and die Work-Nodes (Arbeitsknoten) gesendet. Dies erlaubt dann letztendlich die Verteilung

der Workload innerhalb des Clusters.

2.2. CUDA

Im Vergleich zu einer CPU besteht eine Grafikkarte aus einer großen Anzahl parallel nutzbarer Kerne, die eine Vielzahl von Berechnungen parallel ausführen können. Beispielsweise besitzt eine *NVIDIA RTX 2080* 2944 CUDA-Kerne, während die meisten CPUs 8 bis 16 Kerne besitzen [3]. Natürlich sind die Kerne ein GPU in ihrer Struktur und Instruktionskomplexität wesentlich einfacher, da sie dafür ausgelegt sind, Aufgaben als Gruppe zu bearbeiten. Eine CPU ist auf generelle Programmanforderungen ausgelegt und muss eine Vielzahl von Befehlen und Datentypen unterstützen. Eine GPU ist hingegen auf Grafikberechnungen spezialisiert und in ihrer grundlegenden Funktion für Pixelberechnungen konzipiert worden.
Aufgrund dieser Struktur lassen sich GPUs aber nicht nur für Grafikberechnungen nutzen, sondern auch bei der Berechnung von wissenschaftlichen Problemen. Um diese Architektur nutzen und steuern zu können, wurde die Programmierschnittstelle CUDA (*Compute Unified Device Architecture*) entwickelt. CUDA wurde dabei von NVIDIA entwickelt und erlaubt die Beschleunigung von Programmen, indem neben der CPU bestimmte Programmteile von einer oder mehreren GPUs parallelisiert bearbeitet werden.
CUDA besitzt Bindings für viele wissenschaftlich genutzte Programmiersprachen wie C/C++, Fortran oder Python und ist mit allen herkömmlichen Betriebssystemen kompatibel [4].

2.3. CUDA und MPI

Sowohl CUDA als auch MPI dienen in ihrer grundlegenden Funktion zur Bereitstellung einer Schnittstelle von parallelen bzw. verteilten Berechnungen. MPI erlaubt es wie schon oben erwähnt, bestimmte Aufgaben innerhalb eines verteilten Systems zu koordinieren und aufzuteilen. Grundsätzlich wird dabei nur die Rechenleistung der einzelnen Prozessoren genutzt [5]. Für eine weitere Optimierung durch Verwendung von verschiedenen GPUs in einem verteilten System ist zusätzlich CUDA als GPU-Schnittstelle notwendig. Dabei ergibt sich jedoch folgendes Problem: Standartmäßig werden bei der Verwendung von MPI nur Zeiger auf den Hauptspeicher des Hosts weitergeleitet. Jedoch müssen bei der Kombination von MPI und CUDA meist GPU-Buffer gesendet werden. Hierzu müssen diese Buffer zunächst in den Hauptspeicher kopiert werden, um dann gesendet werden zu können:

```
//MPI rank 0
cudaMemcpy(s_buf_h,s_buf_d,size,cudaMemcpyDeviceToHost);
MPI_Send(s_buf_h,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank 1
MPI_Recv(r_buf_h,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
cudaMemcpy(r_buf_d,r_buf_h,size,cudaMemcpyHostToDevice);
```

Diese Tatsache führt zu erheblichen Zeitverlusten und kann eine Optimierung von Berechnung mit CUDA und MPI einschränken. Daher wurde eine Vielzahl von MPI-Implementation wie *MVAPICH2* oder *OpenMPI* um eine spezielle CUDA-Unterstützung erweitert. Diese Unterstützung trägt den Namen *CUDA-aware MPI*. Mit CUDA-aware MPI können GPU-Buffer direkt an MPI weitergegeben werden:

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,1,100,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,100,MPI_COMM_WORLD, &status);
```

Dabei muss eine CUDA-aware MPI-Implementation unterscheiden, ob der jeweilige Buffer auf dem Hostspeicher oder dem Speicher der GPU liegt. Seit der CUDA-Version 4.0 wurde das CUDA-Feature UVA (*Unified Virtual Addressing*) eingeführt, welches den Hauptspeicher und den Gerätespeicher kombiniert und einem virtuellen Adressraum zusammenfasst. Ein weiteres

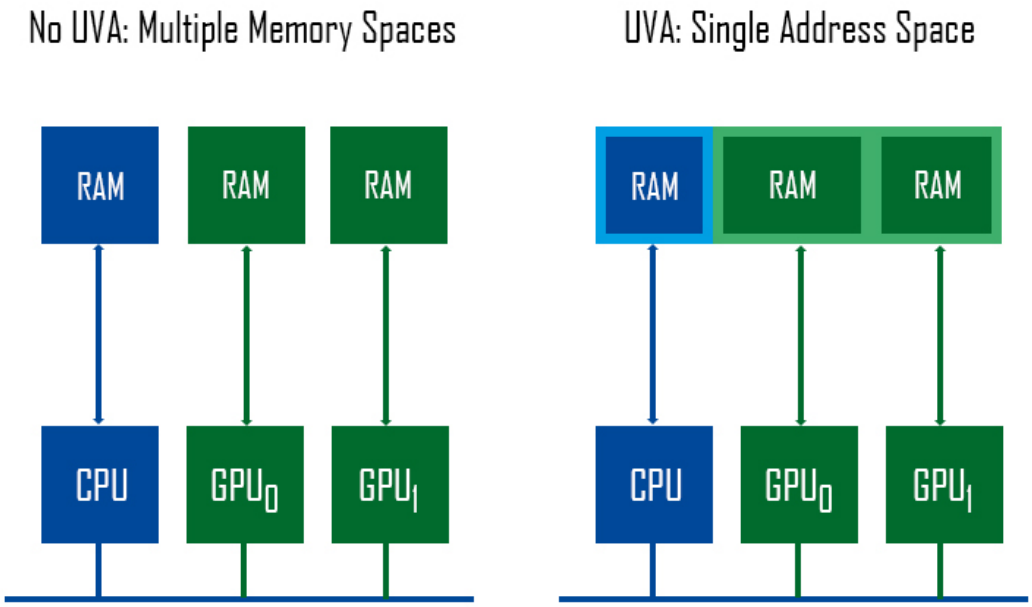

Abb. 2: Links ist das Speicherhandling ohne UVA abgebildet, rechts das Speicherhandling mit UVA. Bei der Nutzung von Unified Virtual Addressing werden die Speicherräume aller System- und Gerätespeicher in einem virtuellen Adressraum zusammengefasst.

Feature von CUDA, eingeführt mit der Version 5.0, ist die *GPUDirect*-Funktionalität. Dies erlaubt mithilfe des *Remote Direct Memory Access* (RDMA) das direkte Senden von Grafikspeicher an den Netzwerkadapter ohne Routing über den jeweiligen Hauptspeicher. Diese Funktionen bilden die Grundlage für eine gelungene Optimierung von verteilten Berechnungen auf verteilten Systemen durch Nutzung von CUDA-aware MPI.

3. Technischer Aufbau und Konfiguration

Für den Aufbau des Mini-Clusters werden zunächst folgende Dinge benötigt:

- Nvidia Jetson Nano (3x)
- SD-Karte (3x)
- Ethernet-Kabel (4x)
- 4-Port Ethernet Switch
- Optional: 40mm Lüfter (3x, in diesem Fall wurde der Noctua NF-A4x20 verwendet)

Der technische Aufbau beginnt mit der physischen Verbindung der Nvidia Jetson Nanos. Dazu werden diese alle via Ethernet-Kabel mit einem Ethernet Switch verbunden, welcher wiederum

durch LAN mit einem Router und dadurch mit dem Internet verbunden ist. An sich benötigt der Cluster keinen Internetzugang, für die Einrichtung ist dieser jedoch unersetzlich. Einer der Jetson Nanos fungiert als Head-Node (Masterknoten), die anderen zwei als Worker-Nodes.
Ist das physische Setup abgeschlossen, wird als nächstes auf jedem der Jetson Nanos das Jetson Nano Developer Kit SD Card Image installiert. Dies geschieht durch das Schreiben des Images auf die jeweilige SD-Karte und einem anschließenden Setup auf den Jetson Nanos [6]. Danach wird das System durch Aktualisierung der Pakete auf den neuesten Stand gebracht:

```
sudo apt-get update
sudo apt-get upgrade
```

Es folgt die Installation des `nano`-Texteditors und des SSH-Paketes. Dabei ist SSH für den Remotezugriff auf die einzelnen Knoten notwendig; an Stelle des `nano`-Texteditors kann alternativ jeder andere Editor benutzt werden.

```
sudo apt-get install nano
sudo apt-get install openssh-server
```

Zusätzlich ist auf dem Masterknoten der NFS-Kernel-Server zu installieren:

```
sudo apt-get install nfs-kernel-server
```

Auf den Worker-Nodes wird dagegen das `nfs-common`-Paket installiert:

```
sudo apt-get install nfs-common
```

NFS und die damit verbundenen Pakete sind notwendig, um später Dateien innerhalb des Clusters auszutauschen.
Um dieses Netzwerk einzurichten, muss zunächst jedem der drei Knoten eine statische IP zugewiesen werden. Dies erfolgt entweder über die Netzwerkeinstellungen oder über das Bearbeiten der Interface-Datei:

```
cd /etc/network
sudo nano interfaces
```

Dafür werden der Datei folgende Zeilen hinzugefügt:

```
auto eth0
iface eth0 inet static
address 192.168.178.10
gateway 192.168.178.1
netmask 255.255.255.0
```

Dabei ist die Adresse `192.168.178.10` die Adresse des Masterknotens, die Worker-Nodes erhalten dann dementsprechend die IP-Adressen `192.168.178.11` und `192.168.178.12`. Damit die Änderungen wirksam werden, müssen entweder die Jetson Nanos oder der Network-Service neugestartet werden.
Als nächstes erfolgt die Einrichtung von SSH innerhalb des Clusters. Dazu wird zuerst jedem Knoten ein Name zur besseren Identifikation zugewiesen. Hierfür wird die `hostname`-Datei bearbeitet:

```
sudo nano /etc/hostname
```

Der Masterknoten erhält in diesem Fall den Namen `master`, die Worker-Nodes die Namen `slave1` und `slave2`. Als nächstes werden in der `hosts`-Datei die statischen IP-Adressen aller Knoten hinzugefügt. Dies geschieht wie der vorige Schritt auf allen Knoten:

```
sudo nano /etc/hosts
```

Die hosts-Datei sollte dann so aussehen:

```
192.168.178.10 master
192.168.178.11 slave1
192.168.178.12 slave2
```

Nach dem Einrichten der statischen IP-Adressen folgt nun das Setup von SSH. Dazu wird zunächst ein 2048 bit RSA Schlüsselpaar auf dem Masterknoten erstellt:

```
ssh-keygen -t rsa -b 2048
```

Danach wird die SSH ID an alle Knoten inklusive des Masterknotens weitergeben:

```
ssh-copy-id master
ssh-copy-id slave1
ssh-copy-id slave2
```

Eine Kommunikation zwischen den Knoten ohne ständige Passwort- bzw. Berechtigungsabfrage ist für die Funktionalität des Cluster von großer Bedeutung. Dies geschieht durch das Generieren der known_hosts-Datei im .ssh-Ordner. Dazu wird zunächst eine Datei mit den Namen aller Knoten im .ssh-Ordner angelegt:

```
cd .ssh
sudo nano name_of_hosts
```

Die Datei enthält dann folgende Einträge:

```
master
slave1
slave2
```

Damit der ssh-keyscan die Datei lesen kann, müssen anschließend die Zugriffsberechtigungen geändert werden:

```
sudo chmod 666 ~/.ssh/name_of_hosts
```

Mit folgendem Befehl wird letztendlich die known_hosts-Datei erzeugt:

```
ssh-keyscan -t rsa -f ~/.ssh/name_of_hosts >~/.ssh/known_hosts
```

Als letztes muss diese Datei und die notwendigen Schlüssel noch an die anderen Knoten kopiert werden:

```
cd .ssh
scp known_hosts id_rsa id_rsa.pub nvidia@master:.ssh
scp known_hosts id_rsa id_rsa.pub nvidia@slave1:.ssh
scp known_hosts id_rsa id_rsa.pub nvidia@slave2:.ssh
```

Der letzte Schritt der Konfiguration ist das Erstellen und Mounten eines gemeinsamen Arbeitsordners für alle Knoten. Hierfür das Network File System benutzt, dessen Pakete anfangs installiert wurden. Auf dem Masterknoten wird dabei als erstes dieser Arbeitsordner erstellt:

```
sudo mkdir /cloud
```

Danach muss die exports-Datei auf dem Masterknoten editiert werden:

```
sudo nano /etc/exports
```

Diese Datei enthält alle Informationen über das Exportieren des Arbeitsordners auf die einzelnen Knoten:

```
/cloud slave1(rw,sync,no_root_squash,no_subtree_check)
/cloud slave2(rw,sync,no_root_squash,no_subtree_check)
```

Die zwei Worker-Nodes müssen nun diesen gemeinsamen Arbeitsordner mounten. Dazu wird auf jedem Worker-Node der `cloud`-Ordner erstellt und mithilfe des Editierens der `fstab`-Datei gemountet:

```
sudo mkdir /cloud
sudo nano /etc/fstab

master:/cloud /cloud nfs rsize=8192,wsize=8192,timeo=14,intr
```

Manuell lässt sich ein gemeinsamer Arbeitsordner auch manuell mounten:

```
sudo mount master:/cloud /cloud
```

Je nach verwendetem Linuxbetriebssystem muss nun noch sichergestellt werden, dass das CUDA-Toolkit sowie MPI mit Cuda-Unterstützung installiert ist. Das Cuda-Toolkit kann über die offizielle Website heruntergeladen werden ([7]); bei der Konfiguration und Installation von MPI muss je nach Implementierung die Cuda-Unterstützung manuell eingestellt werden.

Der technische Aufbau und die Konfiguration des Clusters ist an dieser Stelle abgeschlossen. Wird wie in dieser Arbeit ebenfalls betrachtet ein weiteres cudafähiges Gerät in den Minicluster eingebunden, so geschieht die Konfiguration analog dieser Schrittfolge.

4. Benchmarking und Test ausgewählter Algorithmen

In diesem Kapitel soll die Implementierung ausgewählter Algorithmen vorgestellt und mit verschiedenen Setups getestet und ausgewertet werden.

4.1. Einfache Vektoraddition

Um das Zusammenspiel von Cuda und MPI besser zu verstehen, wird zunächst ein einfacher Algorithmus zur Vektorenaddition betrachtet. Der Quellcode findet sich dabei in Anhang B. Im wesentlichen geschieht bei der implementierten Vektoraddition mithilfe von Cuda und MPI folgendes: Zunächst wird MPI standartmäßig initialisiert. Dann wird überprüft, ob die angegeben MPI-Hosts cudafähige Geräte besitzen und wenn ja, wie viele. Nach der Allokieren des Speichers auf dem Host und dem Initalisieren der beiden Vektoren wird der Speicher für die verteilte Berechnung allokiert. Dies geschieht in diesem Fall nicht durch eine einfache Allokierung mit `cudaMalloc`, sondern durch `cudaMallocManaged`. Dies bewirkt die Verwendung von dem sogenannten *Unified Memory* [8]. Unified Memory stellt einen einzelnen Speicheradressraum für alle Prozessoren in einem System bereit. Der Zugriff der einzelnen Prozessoren (in diesem Fall die einzelnen Grafikprozessoren) auf den Speicher wird durch Cuda selbst gehandhabt.
Nach der Allokierung des Speichers erfolgt die Aufteilung der Vektoraddition auf die einzelnen Grafikprozessoren durch die Scatterfunktionalität von MPI. Anschließend folgt der Aufruf des Kernels. Die Aufteilung auf den einzelnen Prozessoren an sich erfolgt in diesem Fall durch eine sogenannte Grid-Stride-Loop (siehe [9]).

Für das Benchmarking dieses Algorithmus ist nun zunächst die Zeit interessant, welche der Kernel benötigt und welche das Programm an sich insgesamt benötigt. Eine Zusammenfassung von generierten Ergebnissen ist in der nachfolgenden Tabelle 1 ersichtlich.

	1 GPU - 1 Rank		1 GPU - 4 Ranks		3 GPUs - 3 Ranks		4 GPUs - 4 Ranks	
Vektorgröße	Zeit Kernel	Zeit MPI	Zeit Kernel	Zeit MPI	Zeit Kernel	Zeit MPI	Zeit Kernel	Zeit MPI
5 000 000	26.3 ms	288 ms	7.9 ms	1158.2 ms	24.8 ms	7405.1 ms	27.1 ms	10935.5 ms
10 000 000	44.1 ms	516 ms	22.5 ms	1850.6 ms	34.3 ms	14691 ms	32.4 ms	21829.1 ms
15 000 000	51.3 ms	743 ms	48.7 ms	2746.2 ms	67.6 ms	22092 ms	45.2 ms	32592.9 ms
20 000 000	74.5 ms	977.4 ms	52.5 ms	4670.4 ms	70.2 ms	29351 ms	62.7 ms	43554.5 ms

Tab. 1: Übersicht der Ergebnisse des Benchmarkings des Vektoradditionsalgorithmus. Verglichen wurden hier vier verschiedene Setups: Zuerst wurde der Algorithmus mit einem Rank auf einem Jetson Nano getestet, dann mit vier Ranks auf einem Jetson Nano. Bei dem Test mit 3 GPUs handelt es um drei Nvidia Jetson Nanos; bei dem Test mit 4 GPUs um drei Nvidia Jetson Nanos und einen Nvidia Jetson TX2.

Betrachet man die Kernel Zeiten, so lässt sich feststellen, dass die insgesamte Laufzeit bei der Verwendung eines Grafikprozessors mit einem Rank am Besten ist und die benötigte Kernel-Zeit bei der Verwendung eines Grafikprozessors mit vier Ranks am Besten ist. Die Zeit, welche die Zuhilfenahme von anderen Grafikprozessoren zur Lösung des Problems benötigt, ist wesentlich höher. Dennoch lassen sich in diesem Fall die Vorteile und die Nachteile der gewählten Implementierung erkennen. Das Problem der Vektoraddition ist von seiner Komplexität sehr gering. Somit kann insgesamt weniger Zeit durch die Verwendung von mehreren Grafikprozessoren eingespart werden als durch die Datenübertragungszeit von MPI und durch das automatisierte Speichermanagement von Cuda durch die Verwendung von Unified Memory erzeugt wird. Man sieht das beispielsweise bei Vergleich der Benchmarking-Zeiten von einem Grafikprozessor mit einem MPI-Rank und der von einem Grafikprozessor mit vier MPI-Ranks. Bei der Verwendung von vier Ranks benötigt der Kernel teilweise nur ein Viertel der Zeit (statt $26.3\,ms$ nur $7.9\,ms$ bei einer Vektorgröße von 5000000). Gleichzeitig ist die gesamte Laufzeit viermal so hoch.

4.2. Floyd-Warshall-Algorithmus

Um den Mini-Cluster ausführlicher zu testen und die Prinzipien der verteilten Berechnung mit MPI und Cuda zu veranschaulichen, wurde der Floyd-Warshall Algorithmus implementiert. Dieser Algorithmus ist Bestandteil der Graphentheorie und wurde nach Robert Floyd und Stephen Warshall benannt. Dabei löst dieser Algorithmus das Problem, in einem Knotennetz mit gegebenen Gewichtungen der Kanten die minimalen Distanzen zwischen den Städten zu finden. Das Prinzip des Floyd-Warshall-Algorithmus ist die dynamische Programmierung. Das heißt, dass alle möglichen Pfade zwischen allen Paaren von Knoten schrittweise verglichen werden und letztendlich nur die besten Werte gespeichert werden [10].
Mathematisch lässt sich der Algorithmus folgendermaßen ausdrücken [11]: Es sei ein Graph G mit einer Gewichtsmatrix w. Dabei ist $w[i, j]$ das Gewicht der Kante von i nach j, falls eine solche Kante existiert. Falls es keine Kante von i nach j gibt, so ist $w[i, j]$ unendlich. Mit diesen Angaben lässt sich nun eine Matrix d mit den kürzesten Distanzen bestimmten:

Algorithmus 1 Floyd-Warshall-Algorithmus

1: **Gegeben**: Graph G mit Gewichtungsmatrix w;
2: $n = |V(G)|$
3: Für alle $i, j : d[i, j] = w[i, j]$
4: **for** $k = 1$ bis n **do**
5: **for** jedes Paar i, j **do**
6: $d[i, j] = min(d[i, j], d[i, k] + d[k, j])$

Der Floyd-Warshall Algorithmus hat eine Komplexität von $O(n^3)$, da die Zahl der Paare $(i, j$ quadratisch beschränkt ist. Zunächst wurde eine serielle nicht parallele Version des Algorithmus implementiert und auf einem der Jetson Nanos getestet. Die benötigten Rechenzeiten finden sich in Tabelle 2.

Anzahl Knoten	7	8	9	10
Zeit	48 ms	389 ms	3252 ms	24592 ms

Tab. 2: Rechenzeit der seriellen Implementierung des Floyd-Warshall-Algorithmus

Es wird hier sehr deutlich, dass die Rechenzeit bei einem Graphen mit 7 Knoten noch akzeptabel ist, aber mit jedem weiteren Knoten um ein Vielfaches ansteigt. Mit diesen Zeiten als Vergleich wurde nun eine parallele Version des Floyd-Warshall-Algorithmus implementiert. Der Quellcode dazu befindet sich in dem für diesen Projekt erstelltem Git-Repository. Die benötigten Zeiten und eine grafische Veranschaulichung finden sich in der untenstehenden Tabelle 3 und den nachfolgenden Diagrammen 3 und 4.
Es wurde im Rahmen des Benchmarks jeweils mit einer Anzahl von 7 bis 12 Knoten getestet; dabei wurden jeweils drei Versuche durchgeführt und für die ermittelten Zeiten der Mittelwert gebildet. Dabei wurde der Algorithmus desweiteren in vier verschiedenen Szenarien getestet. Zunächst wurde selbiger auf einem Nvidia Jetson Nano mit einem MPI-Rank und mit 4 MPI-Ranks ausgeführt. Danach wurde der Algorithmus im Cluster mit drei Jetson Nanos und letztendlich mit einem zusätzlichen Nvidia Jetson TX2 getestet. Die gesamte Prozedur wurde insgesamt einmal mit einer Blockgröße von 16x16 und einmal mit einer Blockgröße von 256x256 durchgeführt.
Bei der Betrachtung der Ergebnisse fällt als erstes auf, dass die benötigte Rechenzeit sich bei den zwei gewählten Blockgrößen gegensätzlich verhält. Bei einer Blockgröße von 16x16 erzielt das Setup mit vier Geräten und einer Knotenzahl von 12 die besten Zeiten, bei einer Blockgröße von 256x256 das Setup mit einem Gerät und einem Rank. Insgesamt lässt sich durch die Erhöhung der Blockgröße ein wesentlich besseres Ergebnis erzielen. Es fällt außerdem auf, dass bei einer kleinen Problemgröße die Verwendung eines Gerätes und eines Ranks wesentlich bessere Ergebnisse aufzeigt als bei der Nutzung von mehreren Geräten bzw. Ranks. Dies lässt sich auf die durch Verwendung von MPI entstehenden Kommunikationszeiten zurückführen. Desweiteren kommt hier auch die zusätzliche Zeit durch das erweiterte Speichermanagement hinzu.
Zusammenfassend lässt sich feststellen, dass die Verwendung von mehreren Geräten bei steigender Problemgröße auszahlt. Bei einer kleinen Problemgröße werden die durch MPI entstehenden Latenzzeiten zum Nachteil. Es gilt deshalb hier jeweils die Anzahl der Geräte und Ranks je nach Problem und des Größen abzuwägen.

Blockgröße 16x16				
	1 GPU - 1 Rank	1 GPU - 4 Ranks	3 GPUs - 3 Ranks	4 GPUs - 4 Ranks
Zeit für 7 Knoten	54 ms	125 ms	80 ms	114 ms
Zeit für 8 Knoten	93 ms	135 ms	183 ms	241 ms
Zeit für 9 Knoten	448 ms	1139 ms	436 ms	535 ms
Zeit für 10 Knoten	2002 ms	1362 ms	1539 ms	1488 ms
Zeit für 11 Knoten	11494 ms	7988 ms	4106 ms	4712 ms
Zeit für 12 Knoten	89402 ms	61916 ms	22491 ms	17067 ms

Blockgröße 256x256				
	1 GPU - 1 Rank	1 GPU - 4 Ranks	3 GPUs - 3 Ranks	4 GPUs - 4 Ranks
Zeit für 7 Knoten	10 ms	91 ms	73 ms	105 ms
Zeit für 8 Knoten	14 ms	135 ms	150 ms	218 ms
Zeit für 9 Knoten	53 ms	215 ms	464 ms	523 ms
Zeit für 10 Knoten	113 ms	388 ms	1201 ms	1343 ms
Zeit für 11 Knoten	280 ms	722 ms	3555 ms	4366 ms
Zeit für 12 Knoten	839 ms	1620 ms	11794 ms	14390 ms

Tab. 3: Benötigte Rechenzeit für die parallele Implementierung des Floyd-Warshall Algorithmus. Die angegebenen Zeiten bilden dabei einen Mittelwert aus je drei aufgezeichneten Zeitwerten. Bei dem Test auf einer GPU wurde ein Nvidia Jetson Nano verwendet, bei der Verwendung von 3 GPUs wurden jeweils 3 Jetson Nanos benutzt. Für den letzten Benchmark mit 4 GPUs wurde zusätzlich noch ein Nvidia Jetson TX2 verwendet. Der Benchmark erfolgte einmal mit einer Blockgröße von 16x16 und 256x256.

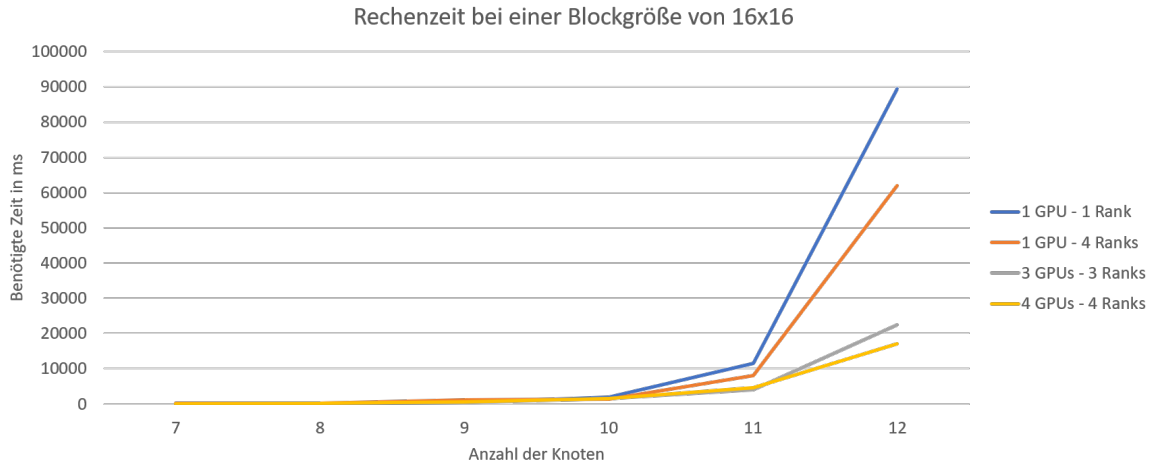

Abb. 3: Grafische Darstellung der Rechenzeit der parallelen Implementierung des Floyd-Warshall-Algorithmus bei einer Blockgröße von 16x16

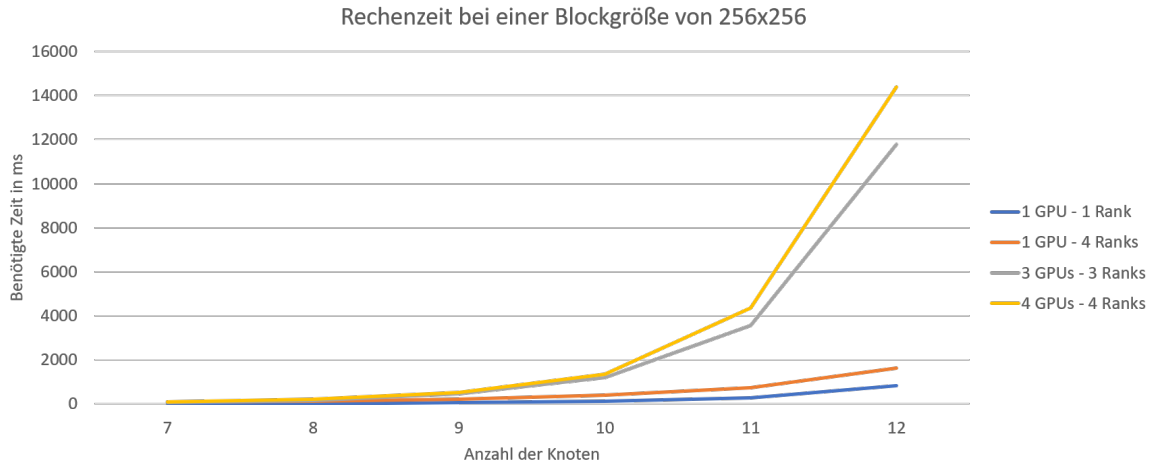

Abb. 4: Grafische Darstellung der Rechenzeit der parallelen Implementierung des Floyd-Warshall-Algorithmus bei einer Blockgröße von 256x256

5. Fazit

A. Aufbau des Clusters

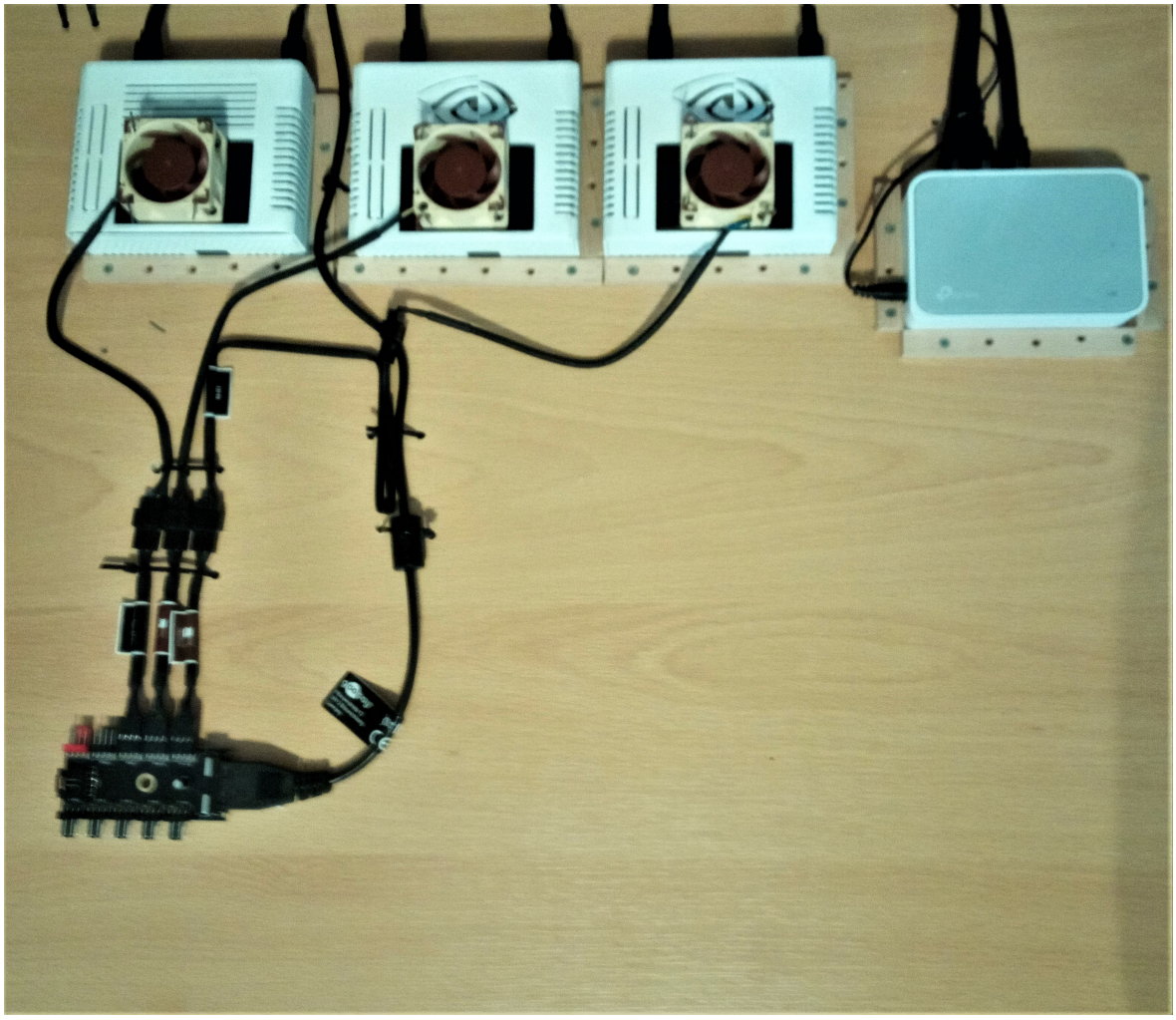

Abb. 5: Der im Rahmen dieses Projektes erstellte Mini-Cluster. Zur besseren Nutzung wurden sowohl die Nvidia Jetson Nanos als auch der Switch durch seitliche Leisten auf einem Holzbrett fixiert. Um die vollständige Leistung durch die Nutzung des Powermodus zu erzielen, wurde desweiteren auf jedem Jetson Nano ein 40mm Lüfter installiert. Für die Steuerung dieser Lüfter wurde ein externer Lüfter-Controller genutzt, weil die Lüfter eine Betriebsspannung von 12V benötigen, der Jetson Nano jedoch nur Lüfter mit 5V Betriebsspannung unterstützt.

B. Quellcode Vektoraddition

```
#include <cstdlib>
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <string>

#include <mpi.h>
#include <unistd.h>

#define BLOCKSIZE 256

// Fehlerbehandlung

#define TRY(command) { cudaError_t cudaStatus = command;
  if (cudaStatus != cudaSuccess)
  {fprintf (stderr, "Error %d - %s\n", cudaStatus,
  cudaGetErrorString(cudaStatus));
  goto Error; }}

// Berechnung der Summe zweier Vektoren

__global__ void sum(int n, float* x, float* y) {

    // Grid-Stride-Loop

    size_t const index = blockIdx.x * blockDim.x + threadIdx.x;
    size_t const stride = blockDim.x * gridDim.x;
    for (size_t i = index; i < n; i += stride) {

            y[i] = x[i] + y[i];
    }

    __syncthreads();
}


int main(int argc, char* argv[]) {

    int N; // Groesse des Vektors

    N = atoi(argv[1]); // Vektorgroesse aus Commandline-Paramter lesen

    // Check Input

    if (N <= 0) {

        printf("Die Vektorgroesse darf nicht Null oder negativ sein!");
        return EXIT_FAILURE;
    }
```

```
// MPI-Init

MPI_Init(&argc, &argv);

int MPIRank, MPISize;
MPI_Comm_rank(MPI_COMM_WORLD, &MPIRank);
MPI_Comm_size(MPI_COMM_WORLD, &MPISize);

// Ueberpruefung, ob und wie viele cudafaehige Geraete vorhanden sind

int devicenumber;
cudaError cudaResult = cudaGetDeviceCount(&devicenumber);
if (cudaResult != cudaSuccess || devicenumber == 0) {

        printf("Es wurden keine cudafaehigen Geraete gefunden!");
        MPI_Finalize();
    return EXIT_FAILURE;
}

// Auslesen der Cuda-Device-Properties

for (int i = 0; i < devicenumber; ++i) {

        cudaDeviceProp properties;
        cudaGetDeviceProperties(&properties, i);
        std::printf("Rank: %d CUDA device %d name: %s \n", MPIRank,
       i, properties.name);
}

// Definition der zwei Vektoren

float* x;
float* y;

double MPIStart, MPIEnd, MPITimePassed;
MPIStart = MPI_Wtime();

// Initialisieren der Vektoren

if (MPIRank == 0) {

        x = new float[N * MPISize];
        y = new float[N * MPISize];

        for (int i = 0; i < N * MPISize; ++i) x[i] = 1.f;
        for (int i = 0; i < N * MPISize; ++i) y[i] = 2.f;
}

int result;

// Allokieren des Geraetespeichers; MPI-Scattering

float* d_x;
float* d_y;
```

```
    // cudaMallocManaged fuer Unified Memory

    cudaResult = cudaMallocManaged(&d_x, N * sizeof(float));
    cudaResult = cudaMallocManaged(&d_y, N * sizeof(float));

    if (cudaResult != cudaSuccess) {

            printf("Fehler beim Allokieren von Cuda-Managed-Memory:
^           %s\n", cudaGetErrorString(cudaResult));
            MPI_Finalize();
            std::exit(EXIT_FAILURE);
    }

    result = MPI_Scatter(x,              // SendBuffer
                         N,                  // SendCount
                         MPI_FLOAT,          // SendType
                         d_x,                // ReceiveBuffer
                         N,                  // ReceiveCount
                         MPI_FLOAT,          // ReceiveType
                         0,                  // Root
                         MPI_COMM_WORLD // Communicator
                        );


    if (result != MPI_SUCCESS) {

        printf("Fehler beim MPI-Scattering");
            MPI_Finalize();
            std::exit(EXIT_FAILURE);
    }


    result = MPI_Scatter(y,              // SendBuffer
                         N,                  // SendCount
                         MPI_FLOAT,          // SendType
                         d_y,                // ReceiveBuffer
                         N,                  // ReceiveCount
                         MPI_FLOAT,          // ReceiveType
                         0,                  // Root
                         MPI_COMM_WORLD // Communicator
                        );


    if (result != MPI_SUCCESS) {

        printf("Fehler beim MPI-Scattering");
            MPI_Finalize();
            return EXIT_FAILURE;
    }

    // Ermitteln der Blockanzahl

    int blocksize = BLOCKSIZE;
    int blocknumber = (N + blocksize - 1) / blocksize;
```

```
    // Kernel

    float CUDATimePassed; //
    cudaEvent_t CUDAStart, CUDAStop; //

    TRY(cudaEventCreate(&CUDAStart));
    TRY(cudaEventCreate(&CUDAStop));
    TRY(cudaEventRecord(CUDAStart, 0));

    // Kernel

    sum<<<blocknumber, blocksize>>>(N, d_x, d_y);

    cudaResult = cudaDeviceSynchronize();
    if (cudaResult != cudaSuccess) {

            printf("Fehler: Cuda asynchron:
      %s\n", cudaGetErrorString(cudaResult));
            MPI_Finalize();
            return EXIT_FAILURE;
    }

    // Zeit-Benchmarking

    TRY(cudaEventRecord(CUDAStop, 0));
    TRY(cudaEventSynchronize(CUDAStop));
    TRY(cudaEventElapsedTime(&CUDATimePassed, CUDAStart, CUDAStop));

    MPI_Barrier(MPI_COMM_WORLD);
    MPIEnd = MPI_Wtime();
    MPITimePassed = MPIEnd - MPIStart;

    if (MPIRank == 0) {

        printf("\nBenoetigte Zeit Kernel: %3.1f ms \n", CUDATimePassed);
        printf("Gesamt benoetige Zeit: %f ms \n", MPITimePassed*1000);
    }



Error:

    cudaFree(d_y);
    cudaFree(d_x);

    if (MPIRank == 0) {

            delete[] y;
            delete[] x;
    }

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Literatur

[1] William Gropp, Torsten Hoefler, Rajeev Thakura, and Ewing Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[2] Oliver Rheinbach. Modul Parallel Computing: MPI - The Message Passing Interface. http://www.mathe.tu-freiberg.de/files/personal/253/parallel-computing-2019-slides-3-mpi.pdf. Zuletzt besucht: 29.08.2019.

[3] Ryan Smith. The NVIDIA GeForce RTX 2080 Super Review: Memories of the Future. https://www.anandtech.com/show/14663/the-nvidia-geforce-rtx-2080-super-review, July 2019. Zuletzt besucht: 26.09.2019.

[4] Stefan Luber. Was ist CUDA? https://www.bigdata-insider.de/was-ist-cuda-a-851005/, August 2019. Zuletzt besucht: 28.09.2019.

[5] Jiri Kraus. An Introduction to CUDA-Aware MPI. https://devblogs.nvidia.com/introduction-cuda-aware-mpi/. Zuletzt besucht: 27.08.2019.

[6] Getting Started With Jetson Nano Developer Kit. https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit. Zuletzt besucht: 28.08.2019.

[7] Cuda-Toolkit. https://developer.nvidia.com/cuda-toolkit. Zuletzt besucht: 28.09.2019.

[8] Mark Harris. Unified Memory for CUDA Beginners. https://devblogs.nvidia.com/unified-memory-cuda-beginners, June 2017. Zuletzt besucht: 29.10.2019.

[9] Mark Harris. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. https://devblogs.nvidia.com/cuda-pro-tip-write-flexible-kernels-grid-stride-loops, April 2013. Zuletzt besucht: 28.09.2019.

[10] Aleksejs Voroncovs. Der Floyd-Warshall Algorithmus. https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_de.html. Zuletzt besucht: 01.11.2019.

[11] Algorithmus von Floyd und Warshall. https://de.wikipedia.org/wiki/Algorithmus_von_Floyd_und_Warshall. Zuletzt besucht: 01.11.2019.