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

26. September 2019

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

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

Inhaltsverzeichnis

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.

26. September 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-

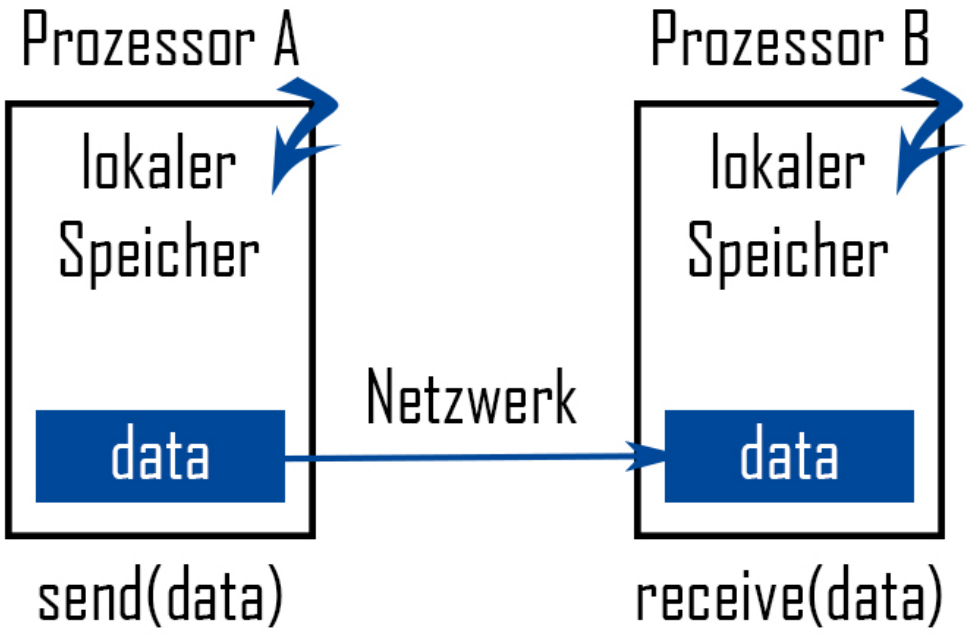

Abb. 1: Grundlegendes Konzept von MPI: Ein Sendeprozess sendet Daten an einen Empfangsprozess über ein Netzwerk. Jeder Prozess kann nur unmittelbar auf seinen zugehörigen lokalen Speicher zugreifen.

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

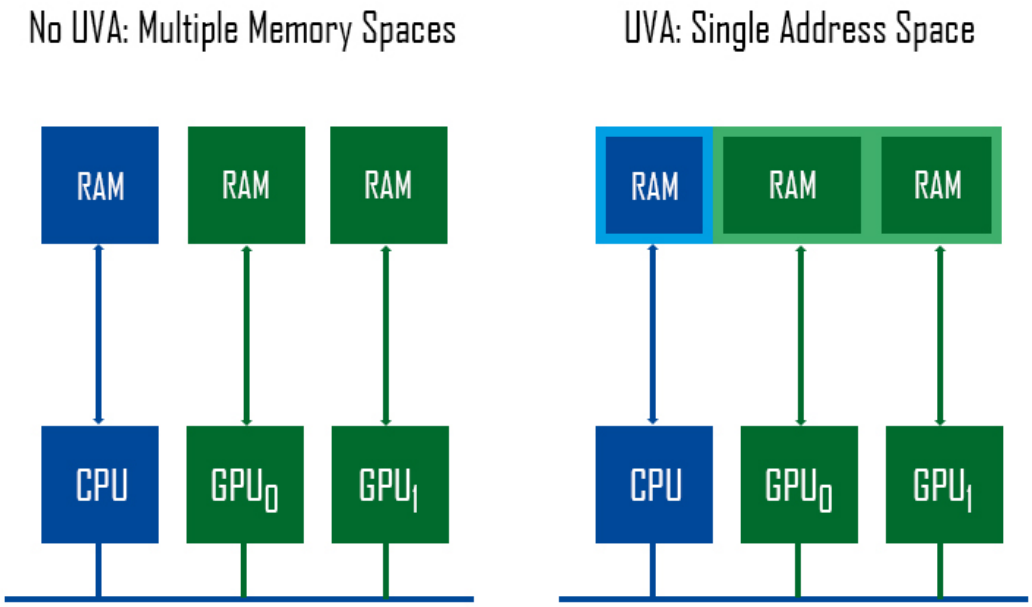

Abb. 2: Links ist das Speicherhandling ohne UVA abgebildet, rechts das Speicherhandling mit UVA. Bei der Nutzung von Unified Virtual Addressing werden die Speicherräume aller System- und Gerätespeicher in einem virtuellen Adressraum zusammengefasst.

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
```

Der technische Aufbau und die Konfiguration des Clusters ist hiermit abgeschlossen.

4. CUDA und MPI

5. Benchmarking und Test ausgewählter Algorithmen

6. Fazit

A. Aufbau des Clusters

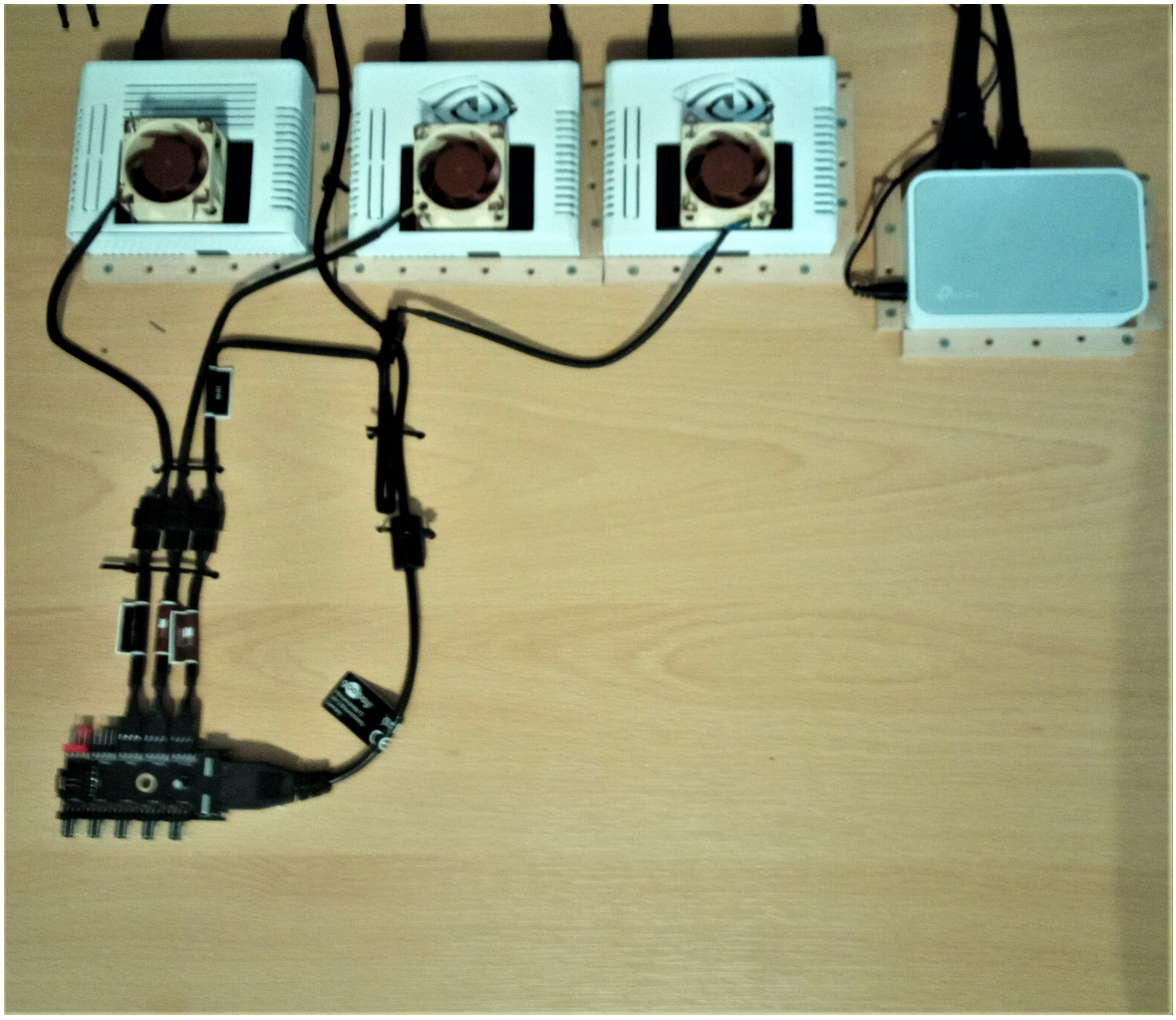

Abb. 3: Der im Rahmen dieses Projektes erstellte Mini-Cluster. Zur besseren Nutzung wurden sowohl die Nvidia Jetson Nanos als auch der Switch durch seitliche Leisten auf einem Holzbrett fixiert. Um die vollständige Leistung durch die Nutzung des Powermodus zu erzielen, wurde desweiteren auf jedem Jetson 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.

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.