Hochschule
Augsburg University of
Applied Sciences

Fakultät für
Informatik

Studienarbeit – Hardware-Systeme

Studienrichtung
Informatik (Master)

Bernd Fecht (954020)
Julian Hillesheimer (954018)

Heterogenes Rechnen mit OpenCL – Theoretische Grundlagen

Prüfer: Prof. Dr. Gundolf Kiefer
Abgabe der Arbeit am: 30.06.2017

Hochschule für angewandte
Wissenschaften Augsburg
University of Applied Sciences

An der Hochschule 1
D-86161 Augsburg

Telefon +49 821 55 86-0
Fax +49 821 55 86-3222
www.hs-augsburg.de
info@hs-augsburg.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Verfasser der Studienarbeit:
Bernd Fecht
Julian Hillesheimer

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Für die Berechnung von Algorithmen werden üblicherweise Hauptprozessoren verwendet. Die meisten Systeme bieten weitere Hardwarekomponenten, die für die Berechnung von Algorithmen verwendet werden könnten. Diese Studienarbeit bietet die Möglichkeit den klassischen Weg der Berechnung auf Hauptprozessoren zu verlassen und zu untersuchen, wie mit dem Framework OpenCL andere Hardware genutzt werden kann. Darüber hinaus ermöglicht diese Arbeit einen Einblick, welche Hardware sich für welche Problemstellungen geeignet nutzen lässt und wie Software hierfür effizient implementiert wird.

Die Softwareentwicklung ist mit steigenden Anforderungen konfrontiert. Die rechnergestützte Simulation, Big Data und künstliche Intelligenz, wie maschinelles Lernen und Mustererkennung, sind drei Beispiele, die hohe Anforderungen an aufwendige Berechnungen stellen. Heterogenes Rechnen mit OpenCL besitzt das Potenzial, solche Probleme anzugehen und zu lösen. Diese Studienarbeit bietet daher die Chance, eine zukunftsweisende Technologie näher zu untersuchen.

1.2. Ziel der Studienarbeit

Ziel dieser Studienarbeit ist die Ausarbeitung der theoretischen Grundlagen bezüglich des Themas heterogenes Rechnen mit OpenCL. Diese Arbeit soll dem Leser das notwendige Wissen vermitteln, damit dieser für einen beliebigen Algorithmus die geeignete Hardware zur Berechnung bestimmen und diesen mit OpenCL implementieren kann. Der Leser soll darüber hinaus in die Lage versetzt werden, ein spezifisches algorithmisches Problem effizient für die Zielhardware zu optimieren.

1.3. Aufbau der Studienarbeit

Im Rahmen dieser Studienarbeit soll zunächst in Kapitel 2 das Thema heterogenes Rechnen untersucht werden. Dabei gilt es zu klären, was heterogenes

Rechnen ist, welche Hardware dafür verwendet werden kann und was es bei heterogenem Rechnen zu beachten gilt.

Anschließend erfolgt innerhalb des Kapitels 3 eine theoretische Einführung in das Framework OpenCL. Hierbei soll zunächst erläutert werden, was OpenCL ist. Danach wird die Funktionsweise des Frameworks anhand von vier Modellen im Detail erklärt. Abschließend erfolgt eine Zusammenfassung, welche Schritte notwendig sind, um ein lauffähiges OpenCL-Programm zu erstellen und auszuführen.

Im abschließenden Kapitel 4 wird das vermittelte OpenCL-Wissen an einem einfachen Anwendungsbeispiel illustriert und ein rückblickendes Fazit auf die gesetzten Ziele der Studienarbeit gezogen.

2. Heterogenes Rechnen

2.1. Was ist heterogenes Rechnen?

„Heterogenes Rechnen umfasst sowohl serielle als auch parallele Verarbeitung. Mit heterogenem Rechnen werden die Aufgaben einer Anwendung dem besten Verarbeitungsgerät, welches auf einem System verfügbar ist, zugewiesen. Die Anwesenheit von mehreren Geräten auf einem System bieten die Möglichkeit für ein Programm Nebenläufigkeit und Parallelität zu nutzen, um die Leistung und den Stromverbrauch zu verbessern.“ [Kae+15]

Neben einer großen Anzahl an unterschiedlicher Hardware, welche für heterogenes Rechnen benutzt werden kann, können Anwendungen ein breites Spektrum an Anforderungen umfassen, wie steuer-, daten- oder rechenintensive Aufgaben. Die Herausforderung beim heterogenen Rechnen besteht somit in der Auswahl einer geeigneten Hardware zur Verarbeitung und der Anpassung der Anwendung an das jeweilige Hardwaregerät. Hierfür kann die Plattform OpenCL verwendet werden. [Kae+15]

2.2. Welche Hardware kann für heterogenes Rechnen verwendet werden?

Die verfügbare Hardware für heterogenes Rechnen steigt stetig weiter an. Die unterschiedlichen Hardwaregeräte können dabei in die vier nachfolgenden Kategorien unterteilt werden. [Kae+15]

- Hauptprozessor (CPU)
- Digitaler Signalprozessor (DSP)
- Field Programmable Gate Array (FPGA)
- Grafikprozessor (GPU)

Die Kategorien selbst können wiederum weiter unterteilt werden. So kann beispielsweise ein Hauptprozessor nur einen oder mehrere Rechenkerne besitzen, aber auch für unterschiedliche Anwendungsbereiche konzipiert sein, wie beispielsweise dem Desktop- oder Serversegment.

2.3. Was muss bei der Auswahl eines Hardware-Gerätes berücksichtigt werden?

Als Vorarbeit für die Auswahl eines geeigneten Hardwaregerätes muss zunächst die zu verarbeitende Problemstellung der Anwendung untersucht werden. Dabei kann zwischen verarbeitungsintensiven Aufgaben bezüglich Steuerung, Daten und Berechnung unterschieden werden. Des Weiteren kann analysiert werden, ob sich eine gegebene Problemstellung parallelisieren kann. Bei der Parallelisierung kann zwischen Daten- und Aufgabenparallelität differenziert werden. Ein Beispiel für Aufgabenparallelität stellt die Vorsortierung von Teillisten beim Sortieren dar. Datenparallelität liegt beispielsweise bei der Vektoraddition vor. [Kae+15]

Ausgehend von der Analyse der Problemstellung können nun die Merkmale der zur Verfügung stehenden Hardwaregeräte untersucht werden, um herauszufinden, welches am geeignetsten ist. Die unterschiedlichen Hardwaremerkmale sind nachfolgend aufgelistet.

- Frequenz
- Superskalarität (Out-of-order execution)
- Very Long Instruction Word (VLIW)
- Single Instruction, Multiple Data (SIMD)
- Mehrkernprozessor
- System-on-a-Chip (SoC)
- Speicher

Frequenz ist ein Merkmal für die Verarbeitungsgeschwindigkeit von nicht optimierten, sequenziellen Anweisungen.

Superskalarität (Out-of-order execution) untersucht dynamisch zur Laufzeit die Abhängigkeiten von Anweisungen zueinander, um durch die Veränderung der Verarbeitungsreihenfolge und die parallele Ausführung von Befehlen die Leistung zu verbessern. Dieser Prozess ist grafisch illustriert im Anhang B.

Very Long Instruction Word (VLIW) untersucht die Anweisungen innerhalb des Quelltexts, um durch Umordnung und paralleler Ausführung von Befehlen, die Geschwindigkeit zu verbessern.

Single Instruction, Multiple Data (SIMD) ist eine Rechnerarchitektur, welche die Ausführung gleicher Anweisungen auf mehreren zur Verfügung stehenden Daten ausführen kann. Die Funktionsweise dieser Architektur ist im Anhang C grafisch skizziert.

Mehrkernprozessor ist ein Hauptprozessor, der über mehrere Rechenkerne besitzt, die unabhängig voneinander unterschiedliche Aufgaben verarbeiten können.

System-on-a-Chip (SoC) ist die Integration unterschiedlicher programmierbarer elektrischer Systeme auf einem Chip. Bieten daher oft einen gemeinsamen Speicher und geringere Latenz sowie einen niedrigeren Stromverbrauch.

Speicher ist das Merkmal wie Hardwaregeräte an unterschiedliche Speicher angebunden sind, welche eigenen internen Speicher (Caches) diese verfügen und welche Größe und Zugriffsgeschwindigkeit sowie welche Latenz diese bieten.

3. OpenCL

3.1. Was ist OpenCL?

„OpenCL (Open Computing Language) ist der erste, lizenzfreie Standard für Allzweckberechnungen auf heterogenen Systemen.“ [BB13]

OpenCL ist jedoch mehr als nur ein Standard. OpenCL ist auch ein Framework, welches Entwicklern eine Programmierumgebung bietet, um effiziente und portable Software zu schreiben, welche auf unterschiedlicher Hardware, unterschiedlicher Hersteller in einem heterogenen System eingesetzt werden kann. Hierfür bietet das Framework einen OpenCL-Treiber für die zu verwendende Hardware und eine Erweiterung und der Programmiersprache C. Mittels dieser erweiterten Programmiersprache, dem zuvor erwähnten Treiber und einer OpenCL Laufzeitumgebung, können OpenCL-Programme, sogenannte *Kernel* für jedes unterstützte Zielhardwaregerät kompiliert und ausgeführt werden. [BB13]

Damit mit dem OpenCL Framework die unterschiedlichste Hardware heterogener Systeme genutzt werden kann, muss OpenCL eine hohe Abstraktion bieten. Hierfür existieren in dem Framework die vier nachfolgenden Modelle. Die Modelle bilden zugleich das grundlegende Verständnis für die Funktionsweise von OpenCL. [Mun+11]

1. Plattformmodell
2. Ausführungsmodell
3. Programmierungsmodell
4. Speichermodell

3.2. Das Plattformmodell

Um unterschiedliche Arten von Hardware von den verschiedenen Herstellern unter einer gemeinsamen Abstraktion zusammenzufassen, bietet OpenCL das Plattformmodell an. In diesem Modell werden die Rollen *host* und *device* definiert. Der *host* ist diejenige Komponente eines heterogenen Systems, die die

Koordination und Verwaltungsaufgaben übernimmt. In der Regel ist dies der Hauptprozessor. Jegliche Hardwaregeräte werden in OpenCL als *device* abgebildet. Der *host* ist somit zugleich auch ein *device*. Ein *device* besteht aus sogenannten *compute units*, die wiederum aus *processing elements* zusammengesetzt sind. [Kae+15]

Die *compute units* und *processing elements* sind abstrakte Einheiten. Das bedeutet, die Zuordnung dieser abstrakten Einheiten ist abhängig von der Hardware. Daher soll die Grafikkarte NVIDIA GTX 750 Ti nun als Beispiel dienen. Die zuvor genannte Grafikkarte verfügt über fünf Streaming Multiprozessoren (SMs), im Falle dieser konkreten Grafikkarte auch als Maxwell Streaming Multiprozessoren (SMMs) bezeichnet. Jede dieser SMs besteht aus vier Blöcken mit jeweils 32 CUDA Kernen. Somit besitzt diese Grafikkarte fünf *compute units* und 640 *processing elements*. Die Anzahl der *processing elements* ergibt sich aus der Anzahl der SMs multipliziert mit der Anzahl der Blöcke und der Anzahl der CUDA-Kerne pro Block. Die SMMs und die CUDA-Kerne sind in der Abbildung A.1 dargestellt, und somit also auch die *compute units* und *processing elements*. [Whi]

Die Auswahl des *device* erfolgt in OpenCL mittels der beiden Befehle *clGetPlatformIDs* und *clGetDeviceIDs*. [Ope]

3.3. Das Ausführungsmodell

Das Ausführungsmodell bietet eine Abstraktionsschicht um ein OpenCL-Programm auf einem Gerät ausführen zu können und die dafür notwendigen Daten und Befehle vom *host* zum *device* weiterzuleiten. Hierfür bietet das Modell die Mechanismen Kontext, Befehlswarteschlange und Ereignisse.

Kontext stellt eine abstrakte Umgebung für die Koordination, Speicherverwaltung und die Ausführung eines OpenCL-Programms dar. [Kae+15] Für die Erstellung eines Kontexts bietet OpenCL den Befehl *clCreateContext* an. [Ope]

Befehlswarteschlange ist eine Liste von Befehlen, die vom Host an das zugehörige Gerät geschickt werden. Befehle umfassen dabei die Ausführung des OpenCL-Programms, den Datentransfer und den Mechanismus der Synchronisation. Die Befehlswarteschlange stellt somit den Kommunikationsmechanismus zwischen Host und Gerät dar. [Kae+15]

Ereignisse bilden in OpenCL Abhängigkeiten zwischen Befehlen ab. Um diese Abhängigkeiten zu ermöglichen, besitzen die Befehle in OpenCL die Status *Queued*, *Submitted*, *Ready*, *Running*, *Ended* und *Completed*. [Kae+15]

3.4. Das Programmierungsmodell

Teil des Programmierungsmodells ist der Kernel, der den Teil einer Anwendung darstellt, der auf einem Gerät ausgeführt werden soll. Des Weiteren werden in diesem Modell die Abstraktionen *work-item* und *work-group* definiert, die eine Einheit der nebenläufigen Ausführung in OpenCL darstellen. Diese Abstraktionen bilden dabei einen dreidimensionalen Indexraum. Die *work-items* stellen dabei die Indizes dar und die *work-groups* unterteilen den Indexraum in kleinere zusammenhängende lokale Indexräume. Somit werden *work-items* auch immer genau einer *work-group* zugeordnet. Für die Identifizierung eines *work-items* besitzen diese jeweils pro Dimension eine ID. Über diesen abstrakten Mechanismus können beliebige Probleme, also sowohl Aufgaben- wie auch Datenparallelität abgebildet werden. Ein Indexraum kann dabei auch nur ein- oder zweidimensional konstruiert werden. [Kae+15]

Das Programmierungsmodell definiert auch, wie ein Kernel erstellt werden kann. Hierfür sind die vier nachfolgenden Schritte notwendig. [Kae+15]

1. Kernelquelltext mittels OpenCL C als Array vom Typ *char* im Speicher abbilden
2. Quelltext in Programmobjekt umwandeln
3. Programmobjekt für Zielgerät kompilieren
4. Mittels Porgrammobjekt und Kernelname ein Kernelobjekt erzeugen

Die binäre Repräsentation eines Kernelobjekts ist dabei abhängig von der Hardware. Bei Hauptprozessoren besteht das binäre Kernelobjekt aus x86-Instruktionen. Bei Grafikkarten wird eine herstellerspezifische *high-level intermediate language* verwendet. Diese wird für die spezifische Architektur der Grafikkarte just-in-time ein einen *Instruction Set Architecture Code* (ISA) übersetzt. [Kae+15]

3.5. Das Speichermodell

Die Speichersysteme können bei unterschiedlicher Hardware und verschiedenen Herstellern stark differenzieren. Mit dem Speichermodell bietet OpenCL daher eine Abstraktion, welche die Hardwarehersteller nutzen können, um diese ihrer Hardware zuweisen zu können. Das Speichermodell umfasst dabei die beiden Abstraktionen der Speicherobjekte und der Speicherregionen. [Kae+15]

Um Eingabe- und Ausgabedaten auf Geräten nutzen zu können, müssen diese als Speicherobjekte abgebildet werden. Hierfür können *Buffer*, *Images* oder *Pipes* genutzt werden. [Kae+15]

Buffer sind eine einfache Datenstruktur, vergleichbar mit Arrays.

Images sind abstrakte Speicherobjekte, um gerätespezifische Optimierungen zu ermöglichen. Dabei wird ein Image mit einem Deskriptor erstellt, der zusätzliche Informationen über die Daten enthält.

Pipe ist eine sortierte Abfolge von Datenelementen, die nach dem FIFO-Prinzip funktioniert. Eine Pipe darf dabei jeweils nur von einem Kernel lesend und von einem einzigen anderen schreibend verwendet werden.

Um die unterschiedlichsten Speichersysteme abbilden zu können, existieren in OpenCL die Speicherregionen *globaler Speicher*, *konstanter Speicher*, *lokaler Speicher* und *privater Speicher*. [Kae+15]

Globaler Speicher ist für alle *work-items* sichtbar, die in einem Kernel ausgeführt werden. Daten vom Host werden immer in den globalen Speicher kopiert. Gewöhnlich ist der globale Speicher der langsamste, aber dafür größte.

Konstanter Speicher ist ein Teil des globalen Speichers, der für den simultanen Zugriff auf Daten ausgelegt ist.

Lokaler Speicher ist ein geteilter Speicher zwischen *work-items* einer gemeinsamen *work-group*. Dieser Speicher befindet sich gewöhnlich auf einem Chip mit niedriger Latenz und hoher Bandbreite.

Privater Speicher ist ein Speicher, der nur von einem einzelnen *work-item* genutzt werden kann. Lokale Variablen innerhalb eines Kernels befinden sich standardmäßig im privaten Speicher.

Bei der bereits betrachteten Grafikkarte NVIDIA GTX 750 Ti wird der globale und der konstante Speicher durch den DRAM abgebildet. Der lokale Speicher befindet sich im 64 KB großen *Shared Memory*, wobei jede SMM einen eigenen *Shared Memory* besitzt. Handelt es sich bei dem privaten Speicher um eine einzelne Variable eines *work-items*, wird diese in einem Register abgebildet. Ansonsten ist der private Speicher eine gecachede Region des DRAM. [HP12]

Für den Datentransfer zwischen Host und Gerät stellt OpenCL die beiden Befehle *clEnqueueWriteBuffer* und *clEnqueueReadBuffer* zur Verfügung. [Ope]

3.6. Schritt für Schritt zum lauffähigen Kernel

Abschließend werden die notwendigen Schritte erläutert um ein lauffähiges OpenCL-Programm, für welches der Quelltext vollständig im Anhang E abgebildet ist, zu erstellen. Als einfaches Anwendungsbeispiel dient hierfür die Vektoraddition.

3.6.1. Analyse der Plattformen und der Geräte

Der erste Schritt dient der Analyse der zur Verfügung stehenden Plattformen und der darin enthaltenen Geräte. Diese Informationen werden benötigt um später eine Plattform und ein zugehöriges Gerät auszuwählen auf den der Kernel ausgeführt werden soll. Die Analyse der Plattformen und der zugehörigen Geräte erfolgt wie in Listing 3.1 abgebildet.

```
cl_int status;
cl_platform_id platform;
status = clGetPlatformIDs(1, &platform, NULL);
cl_device_id device;
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &
    device, NULL);
```

Listing 3.1: Analyse der Plattformen und der Geräte in OpenCL

3.6.2. Einen Kontext erstellen

Wie in Listing 3.2 dargestellt erfolgt in Schritt zwei die Erstellung eines Kontexts. Der Kontext wird für die Koordination, Speicherverwaltung und die Ausführung eines Kernels benötigt.

```
cl_context context = clCreateContext(NULL, 1, &device, NULL
    , NULL, &status);
```

Listing 3.2: Kontext erstellen

3.6.3. Eine Befehlswarteschlange für das Zielgerät erstellen

Nachdem nun der Kontext erstellt wurde, kann ein Zielgerät, illustriert in Listing 3.3 ausgewählt werden, indem für dieses Gerät eine Befehlswarteschlange innerhalb des Kontexts erstellt wird. Über diese Warteschlange erfolgt später die Kommunikation zwischen Host und dem ausgewählten Gerät um Befehle abzusetzen, wie beispielsweise für die Ausführung eines Kernels.

Ausgehend von dem Wissen, welches in Kapitel 2.3 vermittelt wurde, kann Folgendes bei der Auswahl des Zielgeräts berücksichtigt werden. Die Problemstellung der Vektoraddition kann nach dem Prinzip der Datenparallelität verarbeitet werden. Als Zielgerät wurde ein Grafikprozessor ausgewählt, da dieser nach dem SIMD-Prinzip arbeitet und sich daher für die Verarbeitung von Datenparallelität anbietet.

```
cl_command_queue cmdQueue = clCreateCommandQueue(context,
    device, NULL, &status);
```

Listing 3.3: Auswahl eines Zielgeräts und Erstellung einer Befehlswarteschlange

3.6.4. Speicherobjekte für die Daten erstellen

Nun können zwar über die Befehlswarteschlange Befehle an das Zielgerät übermittelt werden, aber bisher wurden noch keine Daten für die Berechnung erzeugt. Wie in Listing 3.4 abgebildet, werden für die Vektoraddition drei Speicherobjekte vom Typ Buffer erzeugt. Die ersten beiden Buffer bilden dabei die beiden Vektoren ab, wohingegen der dritte Buffer das Ergebnis der Addition aufnimmt.

```
const int elements = 4096;
size_t datasize = sizeof(int) * elements;

cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY,
    datasize, NULL, &status);
cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY,
    datasize, NULL, &status);
cl_mem bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    datasize, NULL, &status);
```

Listing 3.4: Speicherobjekte für die Daten erstellen

3.6.5. Daten auf das Zielgerät kopieren

Nachdem nun die Speicherobjekte für die Daten erstellt wurden, gilt es diese in den globalen Speicher des Zielgerätes zu kopieren. Dafür müssen jedoch zunächst die Daten mit Werten initialisiert werden, um anschließend, wie in Listing 3.5 dargestellt über die Befehlswarteschlange auf das Gerät kopiert zu werden. Da der Buffer für das Ergebnis der Vektoraddition zu Beginn keine Daten enthält, muss hierfür nichts kopiert werden.

```
int* A = (int*)malloc(datasize);
int* B = (int*)malloc(datasize);
int* C = (int*)malloc(datasize);
```

```
for (int i = 0; i < elements; i++) {
    A[i] = i;
    B[i] = i;
    C[0] = 0;
}

status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_FALSE, 0,
   datasize, A, 0, NULL, NULL);
status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_FALSE, 0,
   datasize, B, 0, NULL, NULL);
```

Listing 3.5: Initialisierung und Kopiervorgang der Daten

3.6.6. Einen Kernel programmieren und kompilieren

Nachdem nun der Kontext, die Befehlswarteschlange und die Daten erstellt wurden, ist alles vorhanden, um einen Kernel auszuführen. Daher kann nun der Kernel programmiert und kompiliert werden. Für die Abbildung eines Kernels gibt es unterschiedliche Möglichkeiten. Um das Beispiel kurz und übersichtlich zu gestalten, wurde darauf verzichtet den Quelltext des Kernels aus einer Datei einzulesen. Stattdessen wurde dieser direkt im Programm als Array vom Typ *char* abgebildet und anschließend, wie in Listing 3.6 illustriert, kompiliert.

```
const char* programSource =
    "__kernel \n"
    "void vecadd(__global int* A, __global int* B, __global
        int* C) \n"
    "{ \n"
    "\n"
    "int idx = get_global_id(0); \n"
    "C[idx] = A[idx] + B[idx]; \n"
    "} \n";

cl_program program = clCreateProgramWithSource(context, 1,
   (const char**)&programSource,            NULL, &status);
status = clBuildProgram(program, 1, &device, NULL, NULL,
   NULL);
```

Listing 3.6: Kernel für Vektoraddition

3.6.7. Ein Kernelobjekt erstellen

Nachdem zuvor der Kernel programmiert und kompiliert wurde, muss daraus ein Kernelobjekt extrahiert und mit der gewünschten Funktion, welche im

Quelltext des Kernels definiert wurde, verknüpft werden. Dieser Vorgang wird im Listing 3.7 erläutert.

```
cl_kernel kernel = clCreateKernel(program, "vecadd", &
    status);
```

Listing 3.7: Kernel für Vektoraddition

3.6.8. Einen Kernel ausführen

Nun sind alle Objekte vorhanden um die Ausführung des Kernels zu starten. Bevor die eigentliche Ausführung beginnen kann, sind jedoch noch zwei weitere Schritte notwendig. Die zuvor erstellen Speicherobjekte müssen dem Kernel als Argument zugewiesen werden, damit dieser Zugriff auf die Daten erhält. Des Weiteren muss der Indexraum erstellt werden. Damit werden die *work-items* und *work-groups* festgelegt und somit definiert, wie der Kernel von der Zielhardware verarbeitet wird. Der Indexraum ist im AnhangD grafisch skizziert. Nach diesen zwei Schritten kann nun der Befehl für die Ausführung des Kernels über die Befehlswarteschlange abgesetzt werden. Dieser ganze Vorgang ist in Listing 3.8 dargestellt.

```
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);

size_t indexSpaceSize[1], workGroupSize[1];
indexSpaceSize[0] = elements;
workGroupSize[0] = 1024;

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
    indexSpaceSize, workGroupSize, 0,         NULL, NULL);
```

Listing 3.8: Ausführung des Kernel für die Vektoraddition

3.6.9. Ergebnis ausgeben

Um die Ergebnisse der Kernelausführung, also der Vektoraddition ausgeben zu können, müssen diese Ergebnisse, welche sich in einem Buffer und somit im Speicher des Zielgerätes befinden, zunächst zurück zum Host kopiert werden. Dieser Kopiervorgang und die Ausgabe der Ergebnisse sind in Listing 3.9 abgebildet.

```
status = clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0,
    datasize, C, 0, NULL, NULL);

for  (int i = 0; i < elements; i++) {
```

```
    printf("%d \n", C[i]);
}
```

Listing 3.9: Ergebnisse der Vektoraddition ausgeben

3.6.10. Ressourcen freigeben

Als letzter und abschließender Schritt sollten alle Ressourcen, welche in den vorherigen Schritten erstellt wurden, wieder freigegeben werden. Wie die angeforderten Ressourcen freigegeben werden können, wird in Listing 3.10 illustriert.

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseContext(context);

free(A);
free(B);
free(C);
```

Listing 3.10: Ressourcen freigeben

4. Fazit

In Kapitel 2 wurde erläutert, was heterogenes Rechnen ist, welche Hardware dafür verwendet werden kann, und welche Kriterien bei der Auswahl eines geeigneten Hardwaregeräts betrachtet werden müssen, wobei dabei sowohl die Perspektive der Problemstellung, also die Sicht der Software, sowie des ausführenden Geräts, also auch die Sicht der Hardware eingenommen wurde.

In Kapitel 3 wurde zunächst erläutert, was OpenCL ist und wie es in Zusammenhang mit heterogenem Rechnen steht. Im Anschluss wurde die Funktionsweise anhand von vier Modellen und Verweisen auf die API erklärt.

Für diese Studienarbeit wurde im Kapitel 1.2 das Ziel formuliert, dem Leser das notwendige theoretische Wissen zu vermitteln, um eine Problemstellung mittels OpenCL auf einem heterogenen System zu lösen. Das gesetzte Ziel konnte erreicht werden. Wie Kapitel 3.6 illustriert, konnte mit dem zuvor vermittelten theoretischen Wissen die Problemstellung Vektoraddition auf einem heterogenen System erfolgreich mit OpenCL umgesetzt und auf einer Grafikkarte durchgeführt werden.

Anhang

A. Architektur der NVIDIA GTX 750 Ti

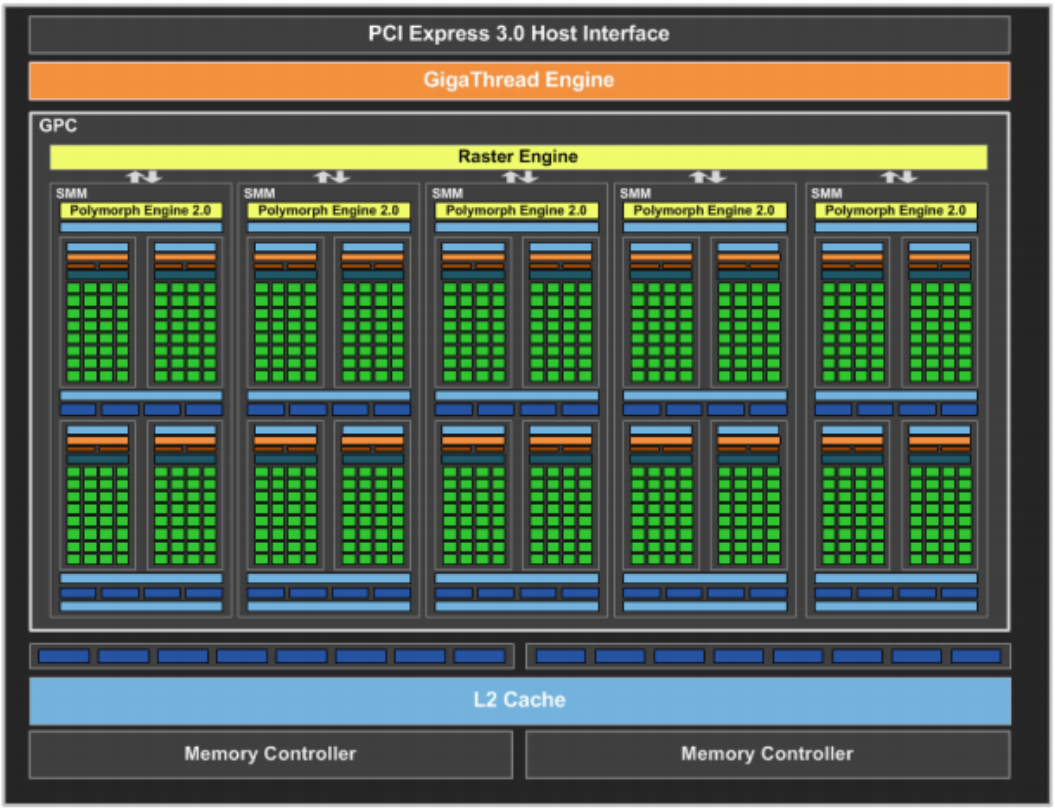

Abbildung A.1.: GM107 Full-Chip Blockdiagramm [Whi]

B. Superskalarität - Out-of-order execution

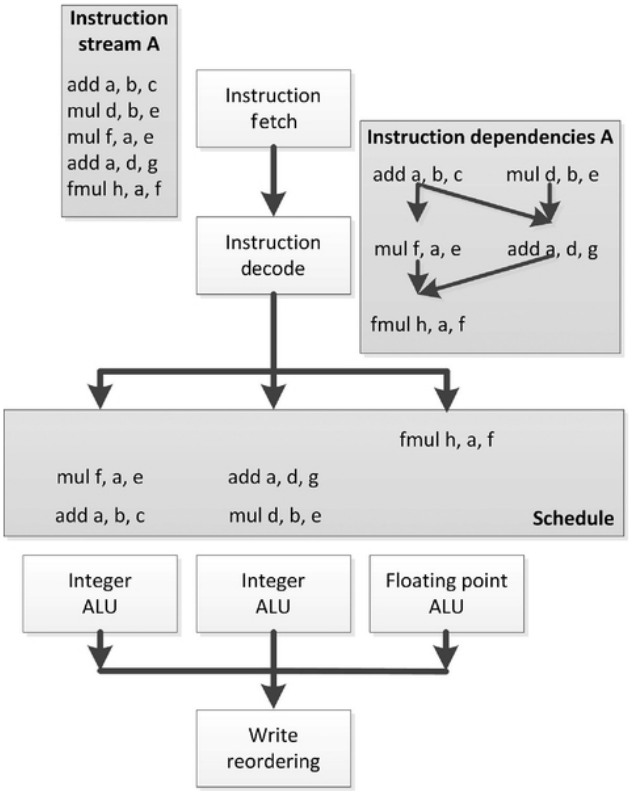

Abbildung B.1.: Out-of-order execution [Kae+15]

C. Single Instruction, Multiple Data (SIMD)

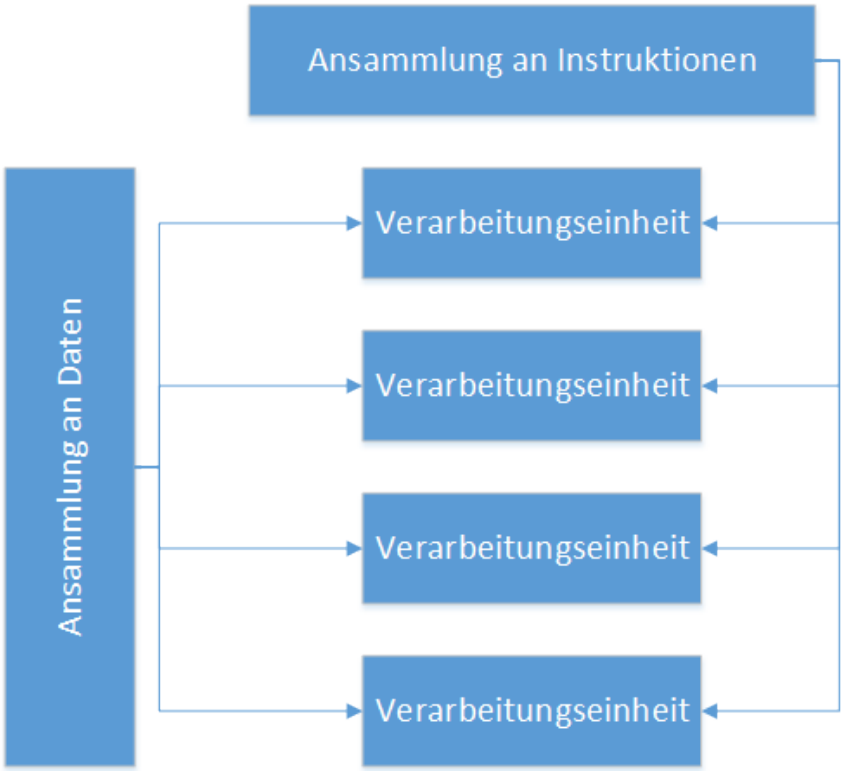

Abbildung C.1.: Funktionsweise SIMD

D. Indexraum

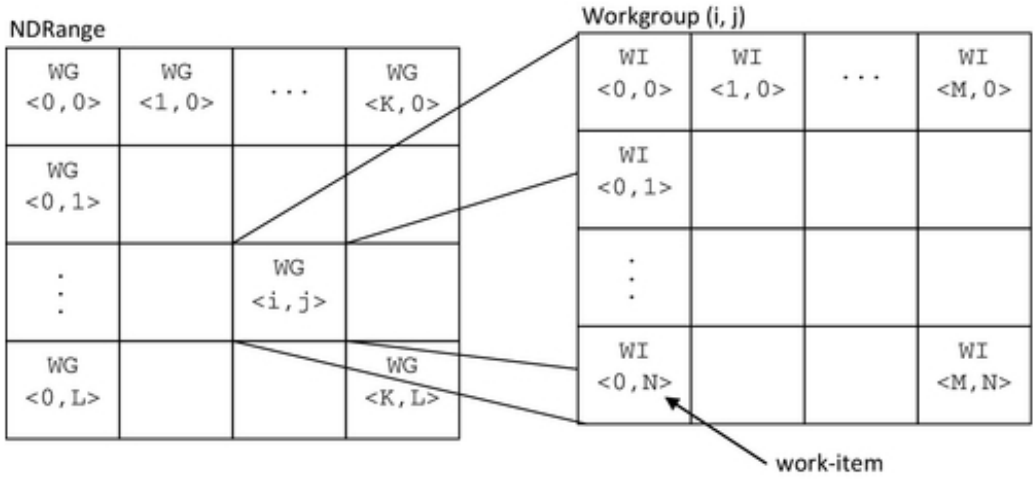

Abbildung D.1.: Indexraum unterteilt in *work-items* und *work-groups* [Kae+15]

E. Quelltext Vektoraddition

```
#include <stdio.h>
#include <stdlib.h>
#include <CL\cl.h>

const char* programSource =
    "__kernel \n"
    "void vecadd(__global int* A, __global int* B, __global int
       * C) \n"
    "{ \n"
    "\n"
    "int idx = get_global_id(0); \n"
    "C[idx] = A[idx] + B[idx]; \n"
    "} \n";

int main() {
    const int elements = 4096;
    size_t datasize = sizeof(int) * elements;

    int* A = (int*)malloc(datasize);
    int* B = (int*)malloc(datasize);
    int* C = (int*)malloc(datasize);

    for (int i = 0; i < elements; i++) {
        A[i] = i;
        B[i] = i;
        C[0] = 0;
    }

    cl_int status;
    cl_platform_id platform;
    status = clGetPlatformIDs(1, &platform, NULL);
    cl_device_id device;
    status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &
       device, NULL);

    cl_context context = clCreateContext(NULL, 1, &device, NULL
       , NULL, &status);
    cl_command_queue cmdQueue = clCreateCommandQueue(context,
       device, NULL, &status);
    cl_mem bufA = clCreateBuffer(context, CL_MEM_READ_ONLY,
```

```
    datasize, NULL, &status);
cl_mem bufB = clCreateBuffer(context, CL_MEM_READ_ONLY,
    datasize, NULL, &status);
cl_mem bufC = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    datasize, NULL, &status);

status = clEnqueueWriteBuffer(cmdQueue, bufA, CL_FALSE, 0,
    datasize, A, 0, NULL, NULL);
status = clEnqueueWriteBuffer(cmdQueue, bufB, CL_FALSE, 0,
    datasize, B, 0, NULL, NULL);

cl_program program = clCreateProgramWithSource(context, 1,
    (const char**)&programSource,            NULL, &status);
status = clBuildProgram(program, 1, &device, NULL, NULL,
    NULL);
cl_kernel kernel = clCreateKernel(program, "vecadd", &
    status);
status = clSetKernelArg(kernel, 0, sizeof(cl_mem), &bufA);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem), &bufB);
status = clSetKernelArg(kernel, 2, sizeof(cl_mem), &bufC);

size_t indexSpaceSize[1], workGroupSize[1];
indexSpaceSize[0] = elements;
workGroupSize[0] = 1024;

status = clEnqueueNDRangeKernel(cmdQueue, kernel, 1, NULL,
    indexSpaceSize, workGroupSize, 0,        NULL, NULL);
status = clEnqueueReadBuffer(cmdQueue, bufC, CL_TRUE, 0,
    datasize, C, 0, NULL, NULL);

for  (int i = 0; i < elements; i++) {
    printf("%d \n", C[i]);
}

clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufA);
clReleaseMemObject(bufB);
clReleaseMemObject(bufC);
clReleaseContext(context);

free(A);
free(B);
free(C);

system("pause");

return 0;
```

```
}
```

Listing E.1: Vektoraddition mit OpenCL

Literatur

[BB13] Ravishekhar Banger und Koushik Bhattacharyya. *OpenCL Programming by Example*. 2013.

[HP12] John L. Hennessy und David A. Patterson. *Computer Architecture: A Quantitative Approach*. 2012.

[Kae+15] David R. Kaeli u. a. *Heterogeneous Computing with OpenCL 2.0*. 2015.

[Mun+11] Aaftab Munshi u. a. *OpenCL Programming Guide*. 2011.

[Ope] *OpenCL API 1.0 Quick Reference Card*. 27. Juni 2017. URL: https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/OpenCL-1.0-refcard.pdf.

[Whi] *NVIDIA GeForce GTX 750 Ti*. 27. Juni 2017. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.