IPVS

Bachelorarbeit Nr. 198

Erfassung von Plattform-Parameters mit OpenCL

Sebastian Staudenmaier

Studiengang:	Informatik
Prüfer/in:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger
Betreuer/in:	Dipl.-Inf. David Pfander
Beginn am:	22. Dezember 2014
Beendet am:	23. Juni 2015
CR-Nummer:	C.1.2, D.1.3

Kurzfassung

Mit OpenCL besteht eine einheitliche Schnittstelle, um für verschiedenste Hardwareplattformen wie CPUs, GPUs oder Beschleunigerkarten zu programmieren. In dieser Arbeit soll darauf eingegangen, wie sich diese Hardware charakterisieren lässt, um Faktoren zu entlarven, die für die Portabilität der Performance abträglich sind. Es wird die Implementierung einer Bibliothek zur Abfrage von Hardwareparametern über die OpenCL-API sowie die Erstellung von Microbenchmarks zur weiteren Untersuchung der Hardware erläutert. Für diese Microbenchmarks werden Testergebnisse präsentiert und deren Bedeutung diskutiert. Im Weiteren werden Nutzungsvorschläge wie auch Erweiterungsmöglichkeiten für die Microbenchmarks genannt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Die nächste kleine Revolution nach dem Multithreading und der Nutzung von Mehrkernprozessoren ist die massiv parallele Ausführung von Berechnungen, beispielsweise auf vergleichsweise günstigen und weit verbreiteten Grafikkarten. Egal ob zur Videobearbeitung, der Teilnahme an Projekten zum verteilten Rechnen wie Einstein@home oder dem Mining von Kryptowährungen, immer öfter werden auch im Mainstream der Computernutzung andere Recheneinheiten als die klassische CPU benutzt. So liefern selbst Grafikkarten für rund 200€ schon Rechenleistungen im Bereich von einigen Teraflops, was selbst deutlich teurere CPUs bei weitem nicht erreichen.

Doch anders als es auf den bekannten x86-CPUs üblich ist, gibt es für Grafikkarten verschiedener Hersteller keinen gemeinsamen Befehlssatz. Dies liegt sicher auch daran, dass die Nutzung von GPUs als frei programmierbare Recheneinheiten zunächst eher über Hacks im Sinne einer kreativen Nutzung der Shader entstanden ist. Um eine einheitliche Programmierschnittstelle für diese neue Kategorie an weit verbreiteter paralleler Recheneinheiten zu schaffen wurde OpenCL ins Leben gerufen. Alleine dadurch sind jedoch noch nicht alle Herausforderungen der parallelen Verarbeitung großer Datenmengen bewältigt. Trotz der einheitlichen Schnittstelle kann ein Programmierer die Eigenheiten der verschiedenen Hardwarekomponenten nicht ignorieren, ohne teils drastische Performanceeinbußen hinnehmen zu müssen.

Dies ist ein Grund, weshalb sich diese Arbeit damit beschäftigt, wie man Eigenschaften von OpenCL-fähigen Geräten, unabhängig von den verschieden Herstellern und Art der Hardware, ermitteln kann. Mit der Kenntnis dieser Eigenschaften kann man dann entweder seine eigenen Programme an die Anforderungen verschiedener Hardware anpassen, oder einen Schritt weiter gehen und zur Laufzeit entsprechende Algorithmen aufrufen lassen, die das entsprechende Gerät besonders effizient nutzen können.

Um die relevanten Informationen zu erlangen sollen einerseits Details der Geräte über die OpenCL-API direkt ausgelesen und zur Verfügung gestellt werden. So lassen sich beispielsweise relevante nach Speichergrößen abfragen, um die Nutzung besonders schnellen Speichers, der aber in seiner Menge begrenzt ist, effizient aufzuteilen. Zum anderen sollen Parameter, wie etwa die Ausführungsgeschwindigkeit von Instruktionen oder die Bandbreiten des entsprechenden Speichers, die nicht in den OpenCL-API zur Abfrage hinterlegt sind, mit Microbenchmarks erfasst werden.

Microbenchmarks sind Messungen bei denen möglichst einfache Zusammenhänge überprüft werden. Dies kann etwa durch das wiederholte Ausführen der immer gleichen Instruktion geschehen, um so einen Maximalwert der Ausführungsgeschwindigkeit zu messen. Im Gegensatz dazu beschäftigen sich reguläre Benchmarks mit komplexeren Problemen, etwa der Darstellung einer ganzen 3D-Welt, um die Leistungsfähig von Hardwarekomponenten als Ganzes zu bestimmen.

2. Verwendung von OpenCL

Zunächst soll hier kurz auf OpenCL eingegangen werden, da es in OpenCL eventuell einige Begrifflichkeiten gibt, die manchem nicht in diesem Zusammenhang bekannt sind. Detaillierte Informationen zu OpenCL finden sich in der OpenCL-Spezifikation[19].

Ausgang nimmt jedes OpenCL-Programm im Host-Programm. Dies kann beispielsweise ein ganz normales C- oder C++-Programm sein, welches die OpenCL-API nutzt und deren Headerdatei einbindet. Dieses Hostprogramm kommuniziert mit den einzelnen OpenCL-Devices. Das können z.B. Grafikkarten, Beschleunigerkarten oder CPUs sein, also auch die CPU auf der das Host-Programm läuft.

Diese einzelnen Devices gehören je zu einer OpenCL-Plattform, die einen Hardwarehersteller wie z.B. AMD, Intel oder Nvidia repräsentiert. Anders als das Host-Programm wird auf den OpenCL-Devices nicht über die klassischen Funktionen der Programmiersprache Speicher reserviert oder Programmcode ausgeführt. Um Speicher auf einem OpenCL-Device zu reservieren muss zunächst ein OpenCL-Buffer angelegt werden. Dieser kann dann mit Daten befüllt werden, die dann beispielsweise im globalen Speicher des OpenCL-Device landen.

Es gibt nicht nur den globalen Speicher, der z.B. bei aktuellen GPUs den mehrere GB großen GDDR5-Speicher repräsentiert, sondern auch noch den konstanten, den lokalen und den privaten Speicher. Eine grobe Veranschaulichung der verschiedenen Speichertypen ist in Abbildung 2.1 zu sehen. Der konstante Speicher ist meist schneller als der globale Speicher und kann Werte die vom Host-Programm an das OpenCL-Device übergeben wurden enthalten. Die Ergebnisse können allerdings nicht wieder zurück in den konstanten Speicher geschrieben werden. Die anderen zwei Speicherarten repräsentieren Speicher, der nicht aus dem gesamten Compute-Device heraus, wie OpenCL z.B. eine einzelne CPU oder GPU nennt, genutzt werden, sondern was Zugriffe anbelangt auf einen kleineren lokalen Bereich beschränkt ist.

Aufgeteilt ist ein Compute-Device bei OpenCL in sogenannte Compute-Units, welche wieder die Processing-Elemente enthalten. Dies ist vereinfacht in Abbildung 2.2 dargestellt. Auf Grafikkarten haben diese Compute-Units üblicherweise einen gemeinsamen lokalen Speicher. Der private Speicher ist üblicherweise eine Bezeichnung für die Register in den Recheneinheiten. Einzelne Instanzen von Code der auf OpenCL-Geräten parallel ausgeführt wird nennen sich Work Items. Damit klar ist, welche dieser zu tausenden nebeneinander laufenden Work-Items z.B. auf den lokalen Speicher mit welchen Work-Items zugreifen kann, werden diese zu sogenannten Work-Groups zusammengefasst. Eine Work-Group wird auf einer Compute-Unit ausgeführt. Insgesamt kann es wiederum mehrere Work-Groups geben. Diese arbeiten nach und nach jeweils Teilprobleme ihrer Größe von einer Menge an Aufgaben ab die global für das ganze Compute-Device zur Ausführung anstehen.

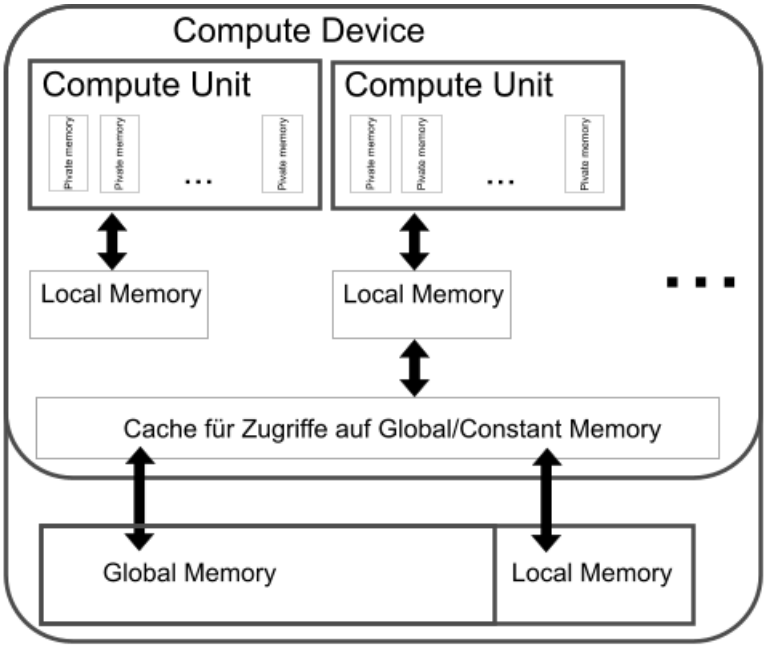

Abbildung 2.1.: Die verschiedenen Speicherarten in OpenCL[19]

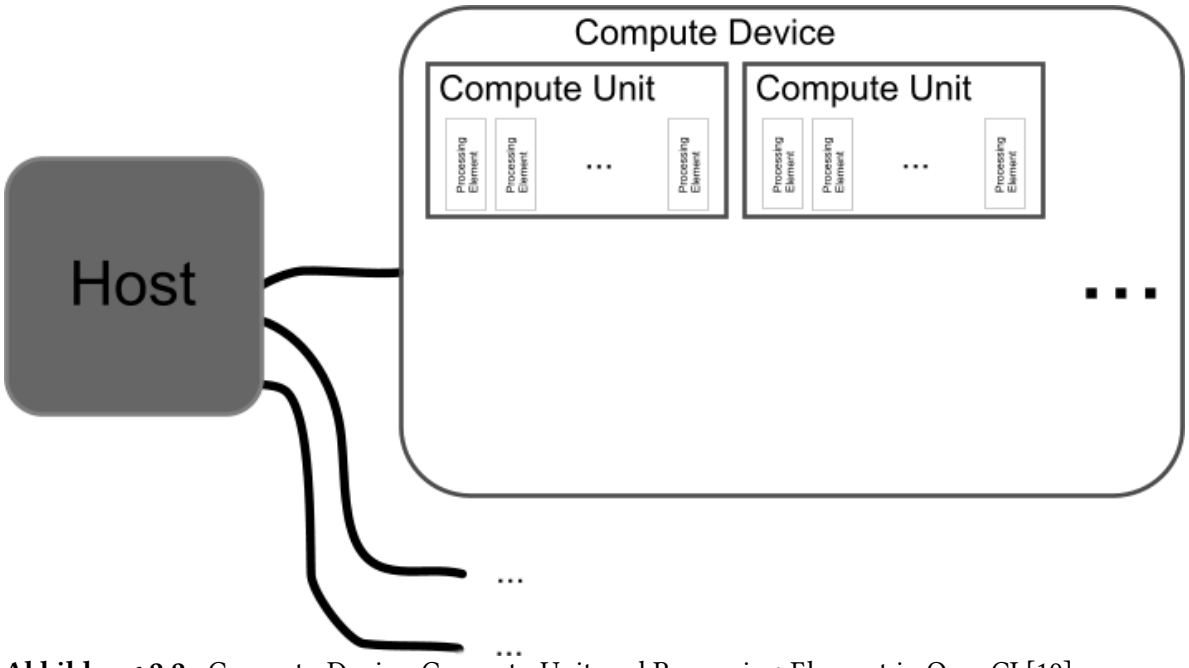

Abbildung 2.2.: Compute-Device, Compute-Unit und Processing-Element in OpenCL[19]

Ein einzelnes Work-Item repräsentiert dabei die Instanz eines Kernels. Mit einem Kernel ist nicht etwa ein Betriebssystemkern gemeint, sondern eine Funktion in die in der Programmiersprache OpenCL-C geschrieben wurde, und auf den Compute-Devices ausgeführt werden kann. OpenCL-C baut wie der Name schon vermuten lässt auf der Programmiersprache C auf, unterstützt jedoch einige Bestandteile von C wie zum Beispiel rekursive Funktionsaufrufe nicht. Dafür sind andere OpenCL-spezifische Elemente vorhanden. Dazu gehören beispielsweise vektorisierte Variablentypen, so gibt es neben den bekannten Typen wie int und float auch noch solche die je mehrere Elemente dieses Typs enthalten, float8 würde beispielsweise aus 8 Fließkommazahlen einfacher Genauigkeit bestehen.

Die Kernel werden im Gegensatz zum Host-Programm zur Laufzeit compiliert. Hierfür muss jeder Hersteller eigene Compiler in seiner OpenCL-Implementierung zur Verfügung stellen, die auf dem System, das das OpenCL-Programm ausführen soll, installiert sein müssen. Dieses vorgehen hat den Vorteil, dass der Programmierer den Code nur einmal in einer quasi abstrahierten Form schreiben muss, und er dann von allen OpenCL-Devices verwendet werden kann. Es bedeutet aber auch, dass zur Ausführungszeit jeweils die Zeit für das Kompilieren der OpenCL-Kernel hinzukommt.

Damit die Instanz eines Kernels weiß welchen Teil des gesamten Problems sie zu berechnen hat ist es notwendig, dass im Kernel anhand der globalen ID und der lokalen ID auf Bereiche im Speicher verwiesen wird, die dann im Work-Item konkret die Daten für dessen Berechnungen beinhalten. Ein Problem kann dabei auch in mehr als nur eine Dimension zerlegt werden, die ID eines Work-Items kann also auch als eine Art Koordinate gesehen werden, wo es sich im Problembereich befindet.

Die Ausführung von Work-Groups erfolgt üblicherweise in Schüben, die unterschiedliche Namen je nach Hersteller haben. Nvidia nennt diese Warps, sie umfassen je 32 Work-Items die gemeinsam ausgeführt werden. Bei AMD heißen sie Wavefront, hier sind es je 64 Work-Items. Deshalb ist es sinnvoll die Größe der lokalen Workgroups jeweils mindestens so groß wie die Anzahl an Work-Items in einem Warp bzw. einer Wavefront zu wählen. Denn sollten bei der Ausführung eines Warps oder einer Wavefront Plätze für Work-Items unbelegt bleiben, so können dort keine Work-Items aus anderen Teilproblemen mit ausgeführt werden, es wird also Rechenleistung nicht genutzt.

2.1. Eignung von OpenCL für Benchmarks

OpenCL stellt, wie bereits beschrieben, nur abstrakte Konzepte zur Beschreibung von Hardware bereit. So muss z.B. der lokale Speicher auf CPUs keine Entsprechung haben, und kann genauso wie der globale Speicher einfach im RAM liegen, da CPUs meist nur transparente Caches haben. Diese Abstraktion muss also bei der Implementierung von Benchmarks immer bedacht und beachtet werden, da man ansonsten gegebenenfalls etwas misst, was man gar nicht messen will.

Ein weiteres Hindernis zur Erstellung verlässlicher Benchmarks stellt der Compiler da, der zur Laufzeit die Kernel übersetzt. Zwar kann man versuchen diese mit der Option -cl-opt-disable auszuschalten, jedoch ist das kein Garant, dass der Compiler den eigenen Code tatsächlich unangetastet lässt.[03](Lecture4 Seite54) Deshalb muss jeweils auch immer darauf geachtet werden, dass der Compiler gar nicht erst die Möglichkeit hat Instruktionen zu entfernen, auf Speicherzugriffe zu verzichten oder diese zusammenzufassen. Dies kann erreicht werden, indem alle gelesenen Informationen aus

dem Speicher tatsächlich genutzt werden und auch ein Ergebnis wieder zurück in den Speicher geschrieben wird, das von allen Operationen zumindest mittelbar abhängig ist.

Andererseits darf man sich auch nicht auf die Intelligenz des Compilers verlassen, so gibt es beispielsweise Compiler die nicht automatisch Instruktionen zu Vektoren zusammenfassen, die gemeinsam in einer SIMD-Einheit verarbeitet werden können. Hier muss man nachhelfen, indem man explizit Vektortypen verwendet. Es gibt die Möglichkeit mit Tools wie z.B. AMD CodeXL das Verhalten der Kernel im Detail zu überprüfen und eine Art Assembler-Code auszulesen, der bei der Übersetzung des Kernels erzeugt wurde. Hier muss man allerdings vorsichtig sein, da es sich nur um eine Art Zwischencode handelt.

3. Aktueller Stand zur Bestimmung von Hardwareparametern durch Microbenchmarks

Die Idee die Charakteristiken von moderner Hardware anhand von Microbenchmarks, auch auf Basis von OpenCL zu bestimmen ist nicht völlig neu. So konnten im Zuge der Recherche für diese Arbeit diverse Softwarelösungen wie auch Literatur zu diesem Thema gefunden werden. Einige davon, die sich am nächsten am Thema dieser Arbeit bewegen, sollen im Folgenden vorgestellt werden.

3.1. Automatic OpenCL device characterization: guiding optimized kernel design, uCLbench

Diese Arbeit[05] stellt eine im Zusammenhang mit ihr entstandene Sammlung von Microbenchmarks zur Charakterisierung von OpenCL-fähigen Geräten vor, welche den Namen uCLbench trägt und frei zugänglich ist.[06]

Als Ziel der Arbeit wird angegeben, die ständig größer werdende Zahl an Hardware und Software besser verstehen zu können und die für eine gute Performance ausschlaggebenden Faktoren automatisch erkennen und zwischen mehreren Geräten vergleichen zu können. Analysiert werden die Rechenleistung sowohl in bei paralleler Ausführung als auch bei der Ausführung von einzelnen Threads für verschiedene Instruktionen, die Speicherlatenzen als auch Bandbreiten auf sowie zwischen OpenCL-fähigen Geräten, der Einfluss von vielen Verzweigungen im Programmcode auf die Ausführungsgeschwindigkeit und die verlorene Zeit durch das Erstellen und Ausführen von OpenCL-Kerneln abseits der eigentlichen Berechnungen.

Als dabei aufgetretene Herausforderungen werden eine möglichst hohe Genauigkeit der Ergebnisse auch auf unterschiedlichen Geräten, das Minimieren von OpenCL-spezifischen unerwünschten Effekten wie Overhead sowie das Verhindern von Compiler-Optimierungen, welche das Ergebnis der Microbenchmarks ungewollt verfälschen könnten, genannt.

Mit den durch die Benchmarks gewonnen Informationen wird anhand eines modellhaften Problems gezeigt, wie sich diese für die Optimierung von Algorithmen nutzen lassen. Als relevante Faktoren werden die Nutzung von entsprechend passenden Vektorbreiten für das jeweilige OpenCL-Device, eine Verminderung der Anzahl an Verzweigungen im Programm als auch eine manuell forcierte Nutzung von schnellem Speicher und Caches beschrieben. Diese Maßnahmen hatten in den Tests unterschiedliche Auswirkungen je nach Gerätetyp und Hersteller. So konnten beispielsweise Nvidia-Grafikkarten kaum von den eingestellten Vektorbreiten profitieren, da hier das automatische Vektorisieren schon

von den Treibern gut übernommen wurde, andere Geräten verhalf diese Maßnahme jedoch zu einem Performanceschub. Wie zu erwarten hatte die Nutzung von weniger Verzweigungen vor allem bei Geräten die keine CPUs sind Erfolg.

Getestet wurde hier auf damals aktuellen CPUs von Nvidia (GeForce GTX 460 und GTX275, Tesla 2050) und AMD (Radeon HD5870) sowie Server-Prozessoren von Intel (Xeon X5570) und AMD (Opteron 2435), auch eine Beschleunigerkarte vom Typ IBM PowerXCell 8i kam zum Einsatz. Insbesondere die GPUs hiervon konnten sich in den Tests zur Rechenleistung gut schlagen und die Herstellerangaben recht genau erreichen. Bei den CPUs war im Gegensatz dazu eine stärkere Diskrepanz zwischen den theoretischen Maximalleistungen und den tatsächlich gemessenen Werten zu erkennen, sie schafften teils nur rund halb so viele Rechenoperationen in der vorgegebenen Zeit.

3.2. Demystifying GPU microarchitecture through microbenchmarking

Diese Arbeit[07] ist vor allem auch deshalb von besonders Interessant, da sie sich als einzige gefundene stärker mit Instruktionslatenzen beschäftigt. Leider ist die Software die genutzt wurde um die Daten für diese Arbeit zu erlangen nicht frei verfügbar, wie bei anderen erwähnten Projekten.

Als Ziel der Arbeit wird angegeben Informationen gerade für Berechnungen auf GPUs über die Herstellerangaben hinaus zu erlangen, um Analysen der Hardware und Optimierungen durchführen zu können. Auch hier werden Microbenchmarks zu diesem Zweck genutzt, jedoch wird hier Nvidias CUDA und nicht OpenCL verwendet. Dementsprechend wurden die Ergebnisse auch auf einer Grafikkarte von Nvidia ermittelt, der GeForce GTX280.

Die Bestimmung der Instruktionslatenzen in dieser Arbeit geht auf die verschiedenen auf der GPU vorhandenen Ausführungseinheiten ein. So gibt es neben den SIMD-Einheiten in einer Recheneinheit noch andere Einheiten, die für die Berechnung komplexerer Funktionen oder Berechnungen von Werten mit doppelter Genauigkeit geeignet sind. Es wird dabei eine ausführliche Tabelle angegeben, die zu vielen häufig verwendeten Instruktionen sowohl deren Instruktionslatenz, also die Verweildauer bis zur Abarbeitung der Instruktion in der Pipeline, als auch den maximalen Durchsatz aufzählt. Die schnellsten Operationen sind die einfache Addition, Subtraktion oder Multiplikation, hierfür befindet sich der Befehl bei ganzzahligen Werten wie auch bei Fließkommazahlen für 24 Taktzyklen in der Pipeline. Die ebenfalls wichtige MAD Instruktion wird für Float-Werte laut den Angaben in der Arbeit auch in 24 Zyklen ausgeführt. Andere Operationen wie die Division von Ganzzahlwerten benötigen hier aber schon über 600 Taktzyklen beziehungsweise 137 Takte für eine Fließkommaoperation, was eine enorme Verlangsamung der Programmausführung bedeuten würde, wenn diese Instruktionen voneinander abhingen. Die Ausführungszeiten für die grundlegenden Instruktionen in doppelter Genauigkeit liegen laut den Messungen bei 48 Taktzyklen auf der GeForce GTX280, können aber natürlich nicht so stark parallel verarbeitet werden, da es auf der Erwähnten Grafikkarte pro Recheneinheit mit acht Skalarprozessoren nur eine Einheit gibt, welche die Operationen in doppelter Genauigkeit ausführen kann.

Weiter geht auch diese Arbeit auf abweichende Ausführungsgeschwindigkeiten im Falle von Verzweigungen im Programmcode, die Nutzung der verschiedenen Speicherarten und Caches und deren Eigenheiten, etwa die Speicherlatenzen, auf der getesteten Grafikkarte ein.

3.3. An OpenCL Micro-Benchmark Suite for GPUs and CPUs

Diese Arbeit[08] setzt auf eine Sammlung von OpenCL-Microbenchmarks auf, die leider auch nicht frei verfügbar sind, deren Ergebnisse allerdings präsentiert werden. Die Benchmarks wurden hier auf relativ schwacher Hardware getestet, ein AMD Athlon II X2 250 und eine Nvidia GeForce GTX 260 zum kamen zum Einsatz. Diese Geräte wurden genutzt um die Bus-Bandbreite zwischen Host und OpenCL-Device sowie beim verschieben von Speicher auf den jeweiligen OpenCL-fähigen Geräten zu bestimmen. Hierbei ergab sich, dass vor allem das verschieben von Speicherinhalten auf der GPU vergleichsweise schnell abläuft, es wurden rund 85GB/s erreicht, wobei die Übertragungsraten im Hauptspeicher der CPU und über die PCI-Express-Schnittstelle mit 1-2GB/s deutlich langsamer waren.

Im Weiteren werden die Speicherbandbreiten beim Zugriff auf den globalen Speicher gemessen, und die Effekte von verschiedenen Zugriffsmustern bestimmt. Hier zeigte sich die GPU variabler als die CPU wenn verschiedene Speicherzugriffe in den Zugriff auf einen einzelnen Speicherblock zusammengefasst werden sollten. Der konstante und lokale Speicher der OpenCL-Devices wird ebenfalls Tests der Lese- und Schreibgeschwindigkeit unterzogen, hier ergaben sich deutlich höhere Bandbreiten.

Der Test der Auswirkung von vielen Verzweigungen zeigte auch in dieser Arbeit wieder, dass CPUs auch mit vielen Verzweigungen keine Einbußen in der Ausführungsgeschwindigkeit zeigen, während sich die Ausführungsdauer auf der Grafikkarte deutlich erhöht.

3.4. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite

Diese Arbeit[08] von den Erstellern der SHOC-Benchmark-Suite, welche auch im Quellcode verfügbar ist[01], erläutert zunächst die Fähigkeiten der Software. SHOC ist in zwei verschiedenen Schnittstellen sowie den dazugehörigen Programmiersprachen implementiert. Neben OpenCL sind die Benchmarks auch in CUDA vorhanden. Des Weiteren findet eine Unterscheidung zwischen Microbenchmarks und Benchmarks die auf höherer Ebene komplexere Algorithmen testen statt.

So sind in der Kategorie Level0 die Microbenchmarks zu finden, die im Zusammenhang mit LibOpenCLBench von besonderem Interesse sind da sie teils als Referenzwerte für diese Implementierung dienen sollen, enthalten. In der Kategorie Level1 finden sich einzelne grundlegende Algorithmen wie beispielsweise die schnelle Fourier-Transformation oder Sortieralgorithmen, die auch für Benchmarks zur Verfügung stehen. In der Kategorie Level2 befindet sich ein Beispiel, wie es bei realen Problemstellungen auftreten könnte.

Die enthaltenen OpenCL-Microbenchmarks bieten Funktionalitäten zur Ermittlung der maximalen Rechengeschwindigkeit mit Fließkommazahlen. Hier werden grundlegende Rechenoperationen in verschiedenen Vektorbreiten getestet und auch Kombinationen wie z.B. Abfolgen aus Multiplikation und Addition, die gegebenenfalls von den ausführenden Geräten zu einer einzigen schnellen in Hardware implementierten Funktion zusammengefasst werden können.

Zwei getrennte Benchmarks sind für die Bestimmung der Übertragungsgeschwindigkeit über den Bus zwischen Host-Programm und Device vorhanden, es wird hier hinsichtlich der Übertragungsrichtung unterschieden. Dem Speicherdurchsatz in den einzelnen OpenCL-Devices widmet sich ein weiteres enthaltenes Microbenchmark. Hier können sowohl Zugriffe auf den globalen als auch auf den lokalen Speicher getestet werden. Weitere Unterscheidungen gibt es hier hinsichtlich neben der Nutzung des Caches in der Zusammensetzung der Speicherzugriffe. So können viele Zugriffe auf ein einzelnes Element im Speicher oder verschiedene Zugriffe auf zufällige Elemente im Speicher genauso getestet werden wie lineare Speicherzugriffe. Auch verschiedene Größen der Work-Groups werden als Faktor für die Übertragungsraten gesehen und durch das Benchmark erfasst.

Neben diesen Eigenschaften der Hardware selbst sind Benchmarks enthalten, welche Eigenheiten von OpenCL selbst messen können. Dazu wird gemessen wie lange die Übersetzung von Kerneln zur Laufzeit dauert und wie sich die OpenCL-Queues verhalten. Neben diesen Benchmarks sind auch Stresstests enthalten die ermitteln ob sich ein Gerät auch unter starker Last noch korrekt verhält oder etwa falsche Ergebnisse liefert.

Für die Daten, welche die zugehörige wissenschaftliche Arbeit über das SHOC-Benchmark liefert wurden unter anderem verschiedene GPUs von Nvidia der GeForce 8000er-Serie sowie eine Tesla C160 und GPUs von AMD der Radeon 5000er-Serie verwendet. Als CPUs für die Tests kamen Xeon-Prozessoren von Intel und Opteron-Prozessoren von AMD zum Einsatz. Neben den Benchmarkergebnissen der einzelnen Geräte, wo die Radeon 5870 mit ihren 1600 Stream-Prozessoren die höchste Leistung der verglichenen OpenCL-Devices erreichte, wurden auch die unterschiede der OpenCL zu der Implementierung in CUDA betrachtet. Während die Ergebnisse hier für die beiden Programmierschnittstellen bei den Lowlevel-Benchmarks nahezu gleich waren konnte mit CUDA bei den komplexeren Algorithmen eine deutlich höhere Leistung erreicht werden. Da die Arbeit mittlerweile aber schon 5 Jahre alt ist, ist davon auszugehen, dass auch die Compiler für OpenCL sich zwischenzeitlich weiterentwickelt haben.

Die SHOC-Benchmark-Suite wurde auch von anderweitig aufgegriffen, so nutzte etwa die Arbeit Improving Performance Portability in OpenCL Programs[10] die SHOC-Benchmarks aus Level1 um gesondert auf das Problem der Performance-Portabilität einzugehen. OpenCL bietet einen Portabilität über mehrere Hersteller und Gerätearten hinweg, was das Ausführen von Code anbelangt. Die Performance-Portabilität dient als Maß dafür, dass Code auf allen unterstützten Geräten nicht nur ausgeführt wird sondern, dass die Ausführung auch mit einer für das entsprechende OpenCL-Device akzeptablen Geschwindigkeit stattfindet. So ist beispielsweise nicht garantiert, dass auf GPUs optimierter OpenCL-Code auch auf CPUs noch performant ausgeführt wird.

Die Arbeit beschreibt mögliche Verbesserungen der OpenCL-Performance auf unterschiedlichen Geräten, so kann etwa die Anordnung der Daten im Speicher anzupassen, sodass sie von GPUs effizient genutzt werden können, ein entscheidender Faktor sein. Für CPUs wird empfohlen auf die verschiedenen automatisch verwalteten Cache-Level zu achten, die in OpenCL nicht direkt abgebildet

werden. Generell sollte falls vorhanden auf Funktionen zurückgegriffen werden, die in der Hardware implementiert sind.

3.5. OpenCL-Benchmarks aus den AMD-APP-SDK-Beispielen

AMD liefert mit seinem APP-SDK[02] etliche OpenCL-Beispiele mit. Darunter befinden sich auch vier Microbenchmarks, die vor allem die Eigenheiten des Speichers von OpenCL-Devices im Detail ausleuchten können. Diese Programme lassen sich allerdings unmodifiziert nur auf AMD-Hardware ausführen. Geprüft werden können mit den Benchmarks von AMD unter anderem die Speicherbandbreiten im lokalen und globalen Speicher je mit verschiedenen Zugriffsmustern.

3.6. MPBenchmarks

Die MPBenchmarks[11] sind eine weitere Sammlung von in OpenCL implementierten Microbenchmarks, deren Quellcode offen zugänglich ist. Sie verfügen über Messungen für die maximalen Rechengeschwindigkeit mit Fließkommazahlen, das verschieben von Speicherinhalten vom Host-Programm zum OpenCL-Device und innerhalb des OpenCL-Devices sowie ein Benchmark für die Multiplikation und Addition von beliebig großen Zahlen. Auf der zugehörigen Webseite finden sich Referenzmessungen die auf die Nvidia GeForce GTX285 zurückgreifen.

3.7. Kommerzielle OpenCL-Microbenchmarks

Neben den erwähnten Benchmarks die entweder für eine wissenschaftliche Arbeit geschrieben wurde, oder zur zur allgemeinen Verfügbarkeit quelloffen zur Verfügung gestellt werden, gibt es auch kommerzielle Produkte in diesem Bereich. Hier sei das AIDA64 GPGPU Benchmark[12] genannt, welches für GPU-Berechnungen ebenfalls auf OpenCL zurückgreift, allerdings die Benchmarks für CPUs nicht mit OpenCL-Kerneln durchführt sondern optimierten normalen Code verwendet. Unterstützt werden die Bestimmung der maximalen Rechenleistung sowohl für Gleitkommazahlen einfacher als auch doppelter Genauigkeit, der Lese- und Schreibgeschwindigkeiten auf den globalen Speicher, die Bandbreite zwischen Host und OpenCL-Device sowie die Berechnungsgeschwindigkeit einiger für kryptografischer Algorithmen.

Ein weiteres Produkt, welches Microbenchmarks für OpenCL-fähige Geräte enthält, ist Sandra von SiSoftware[13]. Es ermöglicht die Bestimmung der maximalen Rechenleistung, auch wieder für Werte in einfacher und doppelter Genauigkeit, sowie die Bestimmung von Bandbreiten und Latenzen des Speichers und der Bandbreite des Caches.

4. Programmbibliothek LibOpenCLBench

Die im Rahmen dieser Arbeit entwickelte Bibliothek nennt sich LibOpenCLBench. LibOpenCLBench stellt zum einen die Abfrage von einigen ausgewählten Parametern aus der OpenCL-API und daraus abgeleiteten Informationen zur Verfügung. Zum anderen wurden auch Microbenchmarks implementiert, welche Eigenschaften der verbauten Hardware bestimmen können, die nicht ohne weiteres direkt auszulesen sind. Konkret handelt es sich hierbei um die Bestimmung der maximalen Rechenleistung der jeweiligen Geräte, der Bandbreiten verschiedener Speicherarten im Gerät, der Bus-Bandbreiten zur Anbindung des Geräts an den Host, der Auswirkungen vieler Verzweigungen im Programmcode sowie der Instruktionslatenzen.

4.1. Implementierung

LibOpenCLBench ist eine in der Programmiersprache C++ geschriebene Bibliothek. Die enthaltenen OpenCL-Kernel sind in der eigenen Programmiersprache OpenCL-C umgesetzt. Für das Host-Programm, welches unter anderem den Programmablauf regelt und die Buffer verwaltet, wird die C++-Wrapper-API von OpenCL genutzt. Zur Nutzung von LibOpenCLBench muss deren Header-Datei libopenclbench.hpp ins Programm eingebunden werden. Danach kann ein C++-Objekt erstellt werden, dessen Konstruktor optional auch gleich den Befehl entgegennimmt, alle Benchmarks auf sämtlichen verfügbaren Geräten auszuführen.

Um LibOpenCLBench mit einem eigenen Programm zu kompilieren muss dem Compiler der Pfad zu den OpenCL-Header-Dateien mitgeteilt werden, da die cl.hpp-Headerdatei genutzt wird. Ansonsten werden nur Header der C- und C++-Standardbibliotheken genutzt. Für den Linker muss außerdem zum erfolgreichen Erstellen des Programms mitgeteilt werden, wo sich die OpenCL-Bibliotheken der jeweiligen Platformhersteller befinden. Da es sich bei LibOpenCLBench nicht um ein eigenständiges Programm handelt und es keine main()-Funktion enthält, muss für zur Erstellung eines Programms aus der Bibliothek mindestens eine weitere C++-Datei erstellt werden. Eine entsprechende Testdatei die LibOpenCLBench einbindet liegt allerdings bei.

Da bei 32-Bit-Betriebssystemen und Anwendungen pro Prozess der genutzte Hauptspeicher 2GB beziehungsweise 4GB gesamt nicht übersteigen kann, muss LibOpenCLBench, sofern es nicht entsprechend für eine geringere Speichernutzung modifiziert wurde, als 64-Bit-Anwendung gebaut werden, da die Ausführung einiger Benchmarks mitunter mehr Speicher erfordern kann.

Von der Struktur her besteht LibOpenCLBench aus einer Haupt-C++-Datei, welche die Aufrufe der einzelnen Benchmarks sowie die Initialisierung und Parameterabfragen regelt, einer C++-Header-Datei, einer C++-Datei für die Klasse zur Speicherung der Ergebnisse, je einer C++-Datei für jedes Benchmark sowie den jeweils ein oder zwei Dateien mit OpenCL-Kerneln für die Benchmarks, soforn

diese einen OpenCL-Kernel nutzen. Die Kernel-Dateien haben ebenfalls die Endung einer C++-Header-Datei, damit die Syntax-Hervorhebung in allen gängigen Editoren für den C-artigen Code automatisch funktioniert.

Wichtig ist zu beachten, dass die Kernel-Datei im Ausführungsverzeichnis des Programm liegen, ansonsten können sie zur Laufzeit nicht geladen werden und es wird ein entsprechender Fehler ausgegeben. Das Ausführungsverzeichnis kann z.B. bei Eclipse-Projekten vom Verzeichnis für Quellcode-Dateien abweichen, entsprechend müssen die Kernel-Dateien also entweder auch oder nur dort abgelegt werden.

In der Header-Datei kann gegebenenfalls ein Debug-Flag gesetzt werden, was dazu führt, dass alle erkannten Geräte nach der Initialisierung ausgegeben werden. Die libopenclbench-Klasse stellt nach außen Methoden zum Abruf der Ergebnisse und ausgelesenen Parameter zur Verfügung. Sollte ein Ergebnis für ein Gerät abgerufen werden, für welches das entsprechende Benchmark noch nicht ausgeführt wurde, so wird das Benchmark mit dem Abruf ausgeführt. Möchte man also die Ergebnisse sofort abfragen können, so müssen die Benchmarks schon zu einem vorherigen Zeitpunkt aufgerufen worden sein, wofür es neben der Option dies mit der Initialisierung durchzuführen auch Methoden gibt die öffentlich für jedes Gerät und Benchmark einzeln aufrufbar sind. Intern speichert LibOpenCLBench die Ergebnisse in einer eigenen Ergebnis-Klasse, wovon eine Instanz für jedes verfügbare OpenCL-Gerät erstellt wird.

4.2. Parameter-Abfragen

Zum Beginn werden mit der Initialisierung der Bibliothek automatisch diverse Parameter aller verbauten OpenCL-fähigen Geräte abgefragt und gespeichert. Hierbei handelt es sich vor allem um Informationen, die für die Anpassung von zur von Programmcode an die Eigenheiten der Geräte hilfreich sein könnten, um eine möglichst performante Ausführung zu erreichen. Es wurden aber auch kleinere Erleichterungen implementiert, etwa die Abfrage, ob eine von den bekannten größeren OpenCL-Plattformen jeweils gerade verfügbar ist, und wo sich diese gegebenenfalls befindet. Auch die Abfrage, ob ein Gerät nativ die Ausführung von Instruktionen für Fließkommazahlen doppelter Genauigkeit unterstützt wird aus den der OpenCL-API abgeleitet, so müssen hier beispielsweise die verfügbaren OpenCL-Erweiterungen nicht mehr nach dieser Information durchsucht werden. Die gesamte Liste der OpenCL-Erweiterungen bildet aber auch einen Teil der über LibOpenCLBench möglichen Abfragen. Diese Ansammlung von Erweiterungen kann zum Beispiel auf andere relevante Spezialerweiterungen durchsucht werden, die dabei helfen können Algorithmen mit bestimmten Anforderungen schneller auszuführen. Weiter sind mehrere Abfragen für die jeweiligen Größen verschiedener Speicherarten, sowie der damit zusammenhängenden Cache-Größen in der Bibliothek vorhanden.

Hiermit kann etwa für die dynamische Generierung von OpenCL-Kernel zur Laufzeit entschieden werden, wie die Aufteilung des Speichers unter optimaler Nutzung besonders schneller Ressourcen wie etwa dem lokalen Speicher einer Compute-Unit geschehen soll, um an dieser Stelle einen Flaschenhals für die Ausführungsgeschwindigkeit zu verhindern.

Auch triviale Abfragen wie die des Gerätenamens oder der Gerätetyps sind möglich. Anhand des Gerätetyps lässt sich bereits eine grobe Einteilung der Geräte vornehmen, so kann es z.B. sinnvoll sein nur GPUs für bestimmte Aufgaben zur Auswahl heranzuziehen und diese dann weiter auf ihre Eigenschaften mit LibOpenCLBench zu untersuchen. Möchte man die Benchmarks auf allen Geräten abfragen, so kann über die ebenfalls implementierten Methoden welche die Anzahl der verfügbaren OpenCL-Plattformen angeben, sowie jeweils die Anzahl der auf dieser Plattform verfügbaren Geräte ermitteln, genutzt werden um einfach in einer Schleife alle Geräte durchzugehen.

4.3. Die Microbenchmarks

Der Großteil der Funktionalitäten von LibOpenCLBench besteht allerdings nicht in der Abfrage von Parametern, die normalerweise auch über die OpenCL-API direkt geschehen könnte, sondern aus den Microbenchmarks, auf deren Funktionalität im Folgenden genauer eingegangen werden soll.

4.3.1. Bestimmung der Rechenleistung

Eine der, wenn nicht sogar die wichtigste Eigenschaft ist die maximal mögliche Rechenleistung. Die theoretischer Maximalleistung wird zwar üblicherweise vom Hersteller entsprechender Geräte angegeben, ist allerdings nicht ohne weiteres direkt auslesbar. Außerdem steht im Zweifelsfall nicht fest, ob es sich hierbei auch um Angaben handelt, die praktisch auch mit eigenen Implementierungen erreichbar sind.

Insgesamt enthält das Benchmark zur Bestimmung der Rechenleistung 28 unterschiedliche OpenCL-Kernel. Alle messen den Durchsatz an Gleitkomma-Operationen, was den häufigsten Einsatzzweck und die beste Eignung gerade von GPUs abdecken sollte. Es werden verschiedene Operationen getestet, neben der Addition ist dies die Multiplikation und die kombinierte Addition und Multiplikation. Daneben gibt es eine Unterscheidung nach Vektorbreiten. Es werden alle in OpenCL üblichen Vektorbreiten von einem einzelnen Vektorelement bis zu 16 Vektorelementen pro Operation getestet. Sofern es das Gerät unterstützt wird das Benchmark auch für die Berechnung mit Gleitkommazahlen in doppelter Genauigkeit durchgeführt, wie sie gerade für wissenschaftliche Anwendungen oft erforderlich sind.

Implementierung des Benchmarks zur Bestimmung der Rechenleistung

Wie auch alle anderen Benchmarks führt die dazugehörige Methode, welche in diesem Fall runinstrPerformanceBenchmark heißt, das Benchmark für ein OpenCL-fähiges Gerät aus, welches über die OpenCL-Plattform respektive den dazugehörigen OpenCL-Kontext und die Nummer des Geräts innerhalb der Plattform als Parameter für den Methodenaufruf spezifiziert wird. Die zur Durchführung des Benchmarks erforderlichen OpenCL-Objekte müssen zunächst zugewiesen bzw. erstellt werden. Dies umfasst auch die OpenCL-Buffer, welche mit Daten aus dem dafür angelegten Arrays des Host-Programms gefüllt werden. Im weiteren Schritten werden die Kernel für das entsprechende Gerät kompiliert. Da zwischen einfacher und doppelter Genauigkeit unterschieden wird sind die Kernel hierfür jeweils in zwei verschiedenen Dateien untergebracht. Ob der Kernel für die Berechnung

mit dem Datentyp Double auch kompiliert werden kann, wird anhand der LibOpenCLBench-Methode getDoublePrecisionSupported bestimmt.

Die Kernel selbst enthalten die Benchmarks sortiert nach der auszuführenden Operation und Vektorlänge. Der erste Benchmark-Kernel ist jeweils das Benchmark zur Messung von Additionen bzw. Subtraktionen. Hierfür wird von einem konstanten Wert ein aus dem Speicher gelesener Wert abgezogen. Dieses Vorgehen wird so in den SHOC-Benchmarks verwendet.[01] Es hat den Vorteil, dass der Wert in den Variablen um Null oszilliert und nicht so schneller besonders große Werte erreicht, was bei ständigem Addieren von zahlen passieren könnte. Dies kann sich wiederum, falls es zu Variableninhalten wie Infinity oder NaN kommen sollte, auch unerwünschter weise auf die Performance auswirken. Die Kernel zum benchmarken der Multiplikationsoperation gehen etwas anders vor, sie multiplizieren die Variablen abwechseln mit einem Wert der kleiner als Eins aber größer als Null ist und einem Wert der etwa den Kehrwert dessen beträgt. So soll ebenfalls erreicht werden, dass es nicht all zu schnell die Grenzen der entsprechenden Wertebereiche erreicht werden. Sollte z.B. hier immer mit dem gleichen Wert multipliziert werden, so würde es sich um eine Exponentialfunktion handeln, die bei zehntausend Ausführungen selbst die Grenzen einer Double-Variablen überschreiten könnte.

Das vorgehen bei den Kerneln die die Ausführungsgeschwindigkeit für eine Multiplikations- und Additionsoperation testen, wird wieder auf eine aus den SHOC-Benchmarks[01] bekannte Vorgehensweise zurückgegriffen. Dabei wird der neue Wert der Variablen bestimmt, indem der alte Wert der Variablen zunächst mit einem festen Wert der etwas kleiner als Eins ist multipliziert und das Ergebnis der Multiplikation dann von einem festen Wert abgezogen wird. Für die Kombination dieser zwei Operationen, der Multiplikation und der anschließenden Addition, haben viele Geräte einen einzelnen Multiply-Accumulate-Befehl, der besonders schnell berechnet werden kann, da solche Berechnungen in vielen oft gebrauchten Algorithmen auftreten. So erreicht man in dieser Konstellation meist die höchste Rechenleistung.

Alle Operationen werden in den einzelnen Kerneln in einer Schleife je 24 mal durchgeführt, die Schleife ist an dieser Stelle also entrollt um eine höhere Ausführungsgeschwindigkeit zu erlangen. Insgesamt wird diese Schleife 10.000 mal durchlaufen, um eine ausreichend große Menge an Instruktionen durchzuführen, was dazu führt, dass andere Faktoren die Teil der gemessenen Ausführungszeit sind, wie etwa die initialen Speicherzugriffe, verhältnismäßig klein gehalten werden. So ist es möglich ein genaueres Ergebnis zu erlangen. In den Schleifen werden alle Operationen in privatem Speicher ausgeführt, wie es OpenCL nennt. Hinter dieser Bezeichnung verbergen sich üblicherweise Register, es sollte also keine Speicherengpässe geben, die das Ergebnis negativ beeinflussen. Am Ende werden die Ergebnisse wieder zurück in den globalen Speicher geschrieben. So wird garantiert, dass der Compiler die Operationen sicher nicht wegoptimiert, da das Ergebnis nicht benötigt würde. Bei Vektorbreiten ab zwei Elementen wird zusätzlich beim initialen Laden der Startwerte aus dem Speicher für jedes weitere Vektorelement zunächst eine kleine Zahl zusätzlich aufaddiert. Dies führt dazu, dass nicht in allen Elementen des Vektors genau die Operation mit den gleichen Werten ausgeführt wird, was zu einer erheblich schnelleren Ausführungsgeschwindigkeit führen kann, wie erste Implementierungen des Benchmarks gezeigt haben, wo diese Maßnahme fehlte. Alle Elemente des Vektors werden nach dem Durchlaufen der Rechenoperationen gespeichert. Das bedeutet, dass die maximale Größe des Buffers entsprechend anhand des breitesten Vektors definiert sein muss, um Platz für das Speichern aller Werte zu haben.

Die Ausführungsdauer des Kernels wird mit einem OpenCL-Event gemessen. Um aus dieser Zeitmessung eine Aussage über die Leistungsfähigkeit des Geräts in FLOPS zu erfahren, ist noch die Anzahl der in dieser Zeitspanne ausgeführten Fließkommaoperationen notwendig. Diese Anzahl variiert auch mit der breite der Vektoren und muss daher für jeden Kernel neu berechnet werden. Die Faktoren für die Anzahl der Fließkommaoperationen pro Work-Item sind die bereits genannte Vektorbreite, die Anzahl der Schleifendurchläufe und die Anzahl der Operationen in der Schleife, also wie stark diese entrollt wurde. Um auf die Gesamtanzahl an ausgeführten Fließkommaoperationen pro Benchmarkausführung zu kommen muss diese Zahl noch mit der Anzahl an insgesamt ausgeführten Work-Items multipliziert werden.

Die Work-Items sollten für eine möglichst hohe Rechenleistung gerade auf GPUs mindestens in Work-Groups der Größe 64 zusammengefasst sein, da es sich hierbei um die Größe einer Wavefront bei AMD-GPUs handelt, die alle gemeinsam ausgeführt werden. Die korrespondierende Größe von Nvidia-GPUs, wo sich diese Einheit Warps nennt, liegt bei 32, größere Work-Groups in der Größenordnung bis 256 schaden dem Durchsatz normalerweise nicht, wobei die maximale Größe für die Dimensionen der Work-Groups bei den entsprechenden Geräten beachtet werden muss. Jeder Kernel wird für das Benchmark zwei mal ausgeführt, wobei immer das erste Ergebnis verworfen wird, durch diese Maßnahme ergaben sich geringere Abweichungen zwischen verschiedenen Ausführungen der Benchmarks. Außerdem wird vor jeder Ausführung der Buffer mit den Daten für die Berechnung neu geschrieben, da er von den vorherigen Benchmarks auch zum Speichern von Ergebnissen genutzt wurde. Hierdurch soll erreicht werden, dass alle Benchmarks mit den gleichen Startvoraussetzungen beginnen. Nachdem jeweils eine Variante zur Bestimmung der Rechenleistung mit einer bestimmten Operation und Vektorbreite durchlaufen wurden werden die Ergebnisse in der Einheit GFLOPS abgespeichert.

4.3.2. Bestimmung von Speicherbandbreiten

Je nach Algorithmus kann die Rechenleistung nicht nur von den maximal möglichen FLOPS abhängen, wenn etwa sehr viele unterschiedliche neue Daten schnell verarbeitet werden sollen. Bei den Benchmarks zur Bestimmung Rechenleistung stellte sich dieses Problem nicht, da immer wieder mit den gleichen Werten gearbeitet wird, die sich in Registern befinden. OpenCL bietet aber neben dem privaten Speicher und dem globalen Speicher auch noch andere Speicherarten, etwa die in der Größe ebenfalls begrenzten lokalen und den konstanten Speicher. Der lokale Speicher, der in OpenCL pro Compute-Unit verfügbar ist, ist etwa noch einmal deutlich schneller als der globale Speicher, dessen Latenz zudem noch recht hoch ist, sofern die Werte nicht in einem Cache liegen. Deshalb ist eine effiziente Nutzung der verschiedenen Speicherarten sowie eine Kenntnis über das Verhalten des Speichers bei verschiedenen Zugriffsabfolgen elementar für die Optimierung von OpenCL-Kerneln. Um diese Optimierung zu ermöglichen wurden für die LibOpenCLBench einige Speicherbenchmarks implementiert. Erfasst werden können damit die Speicherbandbreite des globalen Speichers beim Lesen, jeweils mit und ohne Nutzung des Caches als auch mit zufälligen Zugriffen auf einzelne Speicherbereiche, die Bandbreite beim Schreiben des Globalen Speichers, sowie die Bandbreite beim Lesen des lokalen Speichers.

Implementierung des Benchmarks zur Bestimmung von Speicherbandbreiten

Für die Speicherbenchmarks wird die Methode runmemorybandwidthBenchmark aufgerufen. Anders als im bereits vorgestellten Benchmark ist es hier so, dass die Anzahl an Work-Items unterschiedlich ist. Für den Test des lokalen Speichers werden mehr Work-Items verwendet, da aufgrund der höheren Geschwindigkeit die Messung ansonsten ungenauer würde. Die Daten, die in den Buffer geladen werden haben in diesem Benchmark auch eine Bedeutung. Die Werte sind hierbei jeweils nicht größer als der Speicherbereich, sodass sie als Speicheradressen für den randomisierten Speicherzugriff genutzt werden können, ohne im OpenCL-Kernel Zufallszahlen erzeugen zu müssen.

Die Kernel zur Bestimmung der Speicherbandbreiten basieren jeweils auf den Benchmarks der von AMD mit der OpenCL-Implementierung gelieferten Beispiele.[02] Es gibt bei diesen Benchmarks jeweils zwei verschiedene Kernel, die sich in den Vektorbreiten unterscheiden, um sowohl auf GPUs als auch auf CPUs jeweils gute Ergebnisse erzielen können. Die ersten beiden Kernel dienen zur Bestimmung der Bandbreite des globalen Speichers. Da hier Blockweise gelesen wird, sind relativ hohe Lesegeschwindigkeiten möglich. Außerdem lesen nicht alle Work-Items komplett eigene Bereiche des Speichers, die von der Nummerierung benachbarten Work-Items lesen jeweils nur ein anderes Speicherelement, so kann auch der Cache stark genutzt werden. Jedes gelesene Speicherelement wird für eine Berechnung genutzt, das Ergebnis der Berechnungen wird zurück in den Speicher geschrieben, damit keine Speicherzugriffe durch den Compiler wegoptimiert werden können. Dies geschieht indem zu einer Variablen die im privaten Speicher liegt jeweils der Wert der entsprechenden Speicherzelle abzüglich des vorherigen Werts der Variablen zugewiesen wird. So ist am Ende auch nur eine Variable vorhanden, die in den Speicher geschrieben werden muss, um hierfür nicht unnötig Zeit zu verschwenden.

Die nächsten Kernel versuchen die Nutzung des Caches so gut wie möglich zu verhindern, um die Bandbreite des globalen Speichers selbst zu bestimmen. Hierfür wird innerhalb der Speicherabfragen weit gesprungen, jedoch ist die Startposition immer noch durch die globale Nummer des Work-Items festgelegt, so dass zusammen mit anderen parallel ausgeführten Work-Items eine lineare Abfrage des Speichers möglich ist, was einen unterschied zu den zufälligen Speicherzugriffen bildet. Die zufälligen Zugriffe auf einzelnen Elemente des Speichers folgen in den nächsten Kernel. Hier wird wie schon erwähnt der Inhalt des globalen Speichers, der vom Host-Programm entsprechend präpariert wurde, für die Speicheradressen der jeweiligen Zugriffe genutzt. Hier kann nicht mehr Blockweise aus dem globalen Speicher gelesen werden, wodurch geringere Bandbreiten zu erwarten sind. Dennoch können Kenntnisse über diese Zugriffsart wichtig sein, da gegebenenfalls Algorithmen durchaus nicht immer nur auf lineare Speicherzugriffe angewiesen sein können.

Ein weiteres Paar an Kerneln widmet sich der Messung der Schreibgeschwindigkeit in den globalen Speicher. Hierbei werden jeweils die Nummern der Work-Items in die je 32 von ihnen zu beschreibenden Speicheradressen geschrieben.

Die letzten zwei Kernel für die Speicherbenchmarks beschäftigen sich mit dem lokalen Speicher. Hier werden deutlich mehr Elemente pro Kernel aus dem Speicher gelesen, da die Zugriffe auf den lokalen Speicher deutlich schneller sind und die Messgenauigkeit so höher gehalten werden kann, da andere notwendige Operationen bei der Ausführung des Kernels sonst stärker ins Gewicht fielen. Zusätzlich zum von den Speicherzugriffen auf den globalen Speicher bekannten vorgehen wurde hier außerdem

noch auf die Instruktionslatenzen geachtet und mit zwei Variablen gearbeitet, die nicht voneinander abhängen. Dies geschieht, da wieder alle Elemente auch tatsächlich benutzt werden sollen, um dem Compiler keinen Raum für Optimierungen zu lassen, die die Anzahl an Speicherzugriffen senken würden.

Alle Kernel für die Speicherbenchmarks werden je elf mal ausgeführt, wobei jeweils die erste Messung ignoriert wird. Aus den zehn verbleibenden Messungen wird dann der Durchschnitt gebildet. Um die Bandbreite zu errechnen ist neben der Ausführungsdauer der jeweiligen Kernels auch noch die aus dem Speicher übertragene Datenmenge notwendig. Diese wird durch die Faktoren Anzahl der Work-Items, Vektorbreite, Anzahl an Speicheraufrufen pro Kernel-Instanz und Größe eines Speicherelements ermittelt. Gespeichert wird das Ergebnis aus den Variante mit der Vektorbreite eins und der Vektorbreite acht, das die schnelleren Bandbreite ermitteln konnte. Die Einheit für die gespeicherten Ergebnisse ist GiB/s.

4.3.3. Bestimmung der Bandbreite zwischen Host und OpenCL-Device

Bevor Daten zur Berechnung im Speicher eines Geräts genutzt werden können müssen sie vom Host-Programm dort hin übertragen werden. Praktisch werden dabei zum Beispiel Daten aus dem Arbeitsspeicher des Computers über die PCI-Express-Schnittstelle in den Speicher der Grafikkarte geladen. Diese Übertragung ist meist deutlich langsamer als das lesen von Daten aus dem Speicher direkt. Deshalb kann sich bei einigen Problemen die Frage stellen, ob man Daten für andere Berechnungen aus einem OpenCL-Device ausliest, oder ob es sich lohnt gegebenenfalls diese Berechnungen auch noch auf dem Gerät auszuführen, um sich die Übertragungszeit zu sparen.

Implementierung des Benchmarks zur Bestimmung der Bandbreite zwischen Host und OpenCL-Device

Um in solchen Fragen eine Aussage treffen zu können kann mit der Methode runhostdevicebandwidthBenchmark aus LibOpenCLBench ein entsprechendes Benchmark zu Bestimmung der Bandbreite zwischen Host und OpenCL-Device durchgeführt werden. Dabei handelt es sich um das einzige enthaltene Benchmark, das keinen OpenCL-Kernel benötigt. Die von OpenCL direkt zur Verfügung gestellten Methoden zum Schreiben und Lesen von Daten in und aus einem Buffer reichen hierfür aus. Zunächst wird jedoch die Menge an zu übertragenden Daten festgelegt. Um verlässliche Ergebnisse zu erlangen hat sich eine vergleichsweise große Speichermenge bewährt. Jedoch unterstützen gerade ältere Grafikkarten mit weniger Speicher gegebenenfalls nicht die standardmäßigen 512MiB die übertragen werden sollen. Daher wird zunächst über die OpenCL-API ausgelesen, ob die maximal unterstützte Speichermenge auf einem Gerät kleiner ist, um sie dann entsprechend falls notwendig zu reduzieren.

Die Übertragung der Daten wird mit einem OpenCL-Event in insgesamt 11 Durchläufen gemessen, wobei die erste Messung verworfen wird. Es wird auch eine Referenzmessung für ein einzelnes Speicherelement durchgeführt, das eine vernachlässigbare Größe von 4 Byte haben sollte. Das Ergebnis dieser Referenzmessung wird vom eigentlichen Ergebnis abgezogen, um die reine Transferbandbreite zu ermitteln. Die Messungen für die Übertragung vom Host zum Gerät und anders herum finden jeweils

getrennt statt. Die Werte werden auch getrennt in der Einheit GiB/s gespeichert. Der Grund hierfür ist, dass die Geschwindigkeiten sich für das Schreiben und Lesen vom Gerät merklich unterscheiden können.

4.3.4. Bestimmung des Einflusses von vielen Verzweigungen im Programmcode

Moderne CPUs versuchen unter anderem durch Techniken zur Sprungvorhersage Verzögerungen durch Verzweigungen im Programmcode zu minimieren. Denn sollte ein Sprung im Programmcode stattfinden, der nicht vorhergesehen wurde, so können die entsprechenden Operationen welche tatsächlich ausgeführt werden sollen nicht in die Pipeline der Prozessors geladen werden, was eine Verlangsamung der Ausführung bedingt. Mit diesem Benchmark sollen vor allem verschiedene andere OpenCL-Devices mit CPUs verglichen werden können, was den Umgang mit Verzweigungen anbelangt. Da die vergleichsweise vielen einzelnen Compute Units beispielsweise auf einer GPU eventuell nicht die gleiche komplexe Logik zur Sprungvorhersage besitzen können wie die Kerne einer CPU ist es interessant die Geräte hinsichtlich dieser Eigenschaft zu charakterisieren.

Implementierung des Benchmarks zur Bestimmung des Einflusses von vielen Verzweigungen im Programmcode

Die Methode runbranchingdelayBenchmark führt dazu ein Benchmark aus. Hierzu wird jeweils das Verhalten ganzer Compute-Units überprüft, indem die lokale Größe der Work-Group auf eins gesetzt wird. Dies erfolgt aus dem Grund, dass GPUs oft bei Verzweigungen innerhalb einer Work-Group alle Operationen unabhängig von dem Ablauf der für ein einzelnes Processing-Element vorgesehen ist auf allen Processing-Elementen ausführen müssen, Verzweigungen deshalb also auch nur nacheinander abarbeiten können.[03](Lecture5, Folie 67)

Die Kernel-Datei enthält zwei Kernel, beide führen 32 Operationen aus, die alle dazu benötigten Werte werden zuvor in einem Block in den privaten Speicher geladen, sodass die Ausführungszeit während der Abarbeitung der Operationen selbst nicht von Speicherzugriffen abhängt. Der erste Kernel verwendet die eingelesenen Daten um in Abhängigkeit von deren Inhalt entweder den Wert auf eine Variable hinzu zu addieren oder andernfalls zu subtrahieren. Der Andere Kernel führt einfach am Stück sämtliche Additionen der 32 Elemente aus dem Speicher aus. Das Ergebnis wird zurück in den Speicher geschrieben, damit die Berechnungen vom Compiler nicht entfernt werden, da sie nicht benötigt würden. Als Ergebnis speichert dieses Benchmark das Verhältnis der Ausführungszeit mit Verzweigungen zu der Ausführungszeit der gleichen Anzahl an Operationen ohne Verzweigungen.

4.3.5. Bestimmung von Instruktionslatenzen

Instruktionen werden auf vielen Prozessoren nicht in einem Takt abgeschlossen, sondern müssen mehrere Schritte einer Pipeline durchlaufen. Eine Instruktion hat also auch eine Latenz von einigen Taktzyklen. Wenn nun eine Abhängigkeit zwischen dem Ergebnis einer Instruktion die sich gerade noch in der Pipeline befindet und der nächsten auszuführenden Instruktion ergibt, so muss diese warten, bis die Ausführung der vorherigen Instruktion abgeschlossen ist. Das würde bedeuten, dass

die Pipeline nicht richtig ausgelastet ist. CPUs versuchen diesem Problem deshalb durch Techniken wie Out-Of-Order-Execution entgegenzuwirken. Hierbei werden Instruktionen vorgezogen, die keine Abhängigkeit mit sich derzeit in der Pipeline befindlichen Instruktionen haben. Es kann aber auch Fälle geben, in denen jede Operation auf genau ein oder wenige gemeinsame Register zugreifen, sodass sich die Abhängigkeiten auch durch eine andere Ausführungsreihenfolge nicht aufheben lassen, da immer Abhängigkeiten von einem Operanden vorhanden sind, dessen Wert noch nicht feststeht. In diesem Fall müssten die Instruktionslatenzen zu einem Problem für die Ausführungsgeschwindigkeit werden. Dies gilt insbesondere dann, wenn es sich um komplexere mathematische Funktionen handelt, die nicht in einer einstelligen Anzahl an Taktzyklen abgearbeitet werden können. Um das Verhalten der Hardware in solchen Situationen zu analysieren enthält LibOpenCLBench auch ein Benchmark um Instruktionslatenzen genauer zu untersuchen.

Implementierung des Benchmarks zur Bestimmung von Instruktionslatenzen

Die Methode runinstrlatencyBenchmark ruft das Benchmark zur Messung der Instruktionslatenzen auf. In diesem Fall sollte Parallelisierung keine Rolle spielen, weshalb die Größe der Work-Group und auch der insgesamt ausgeführten Work-Items auf eins gesetzt wird.

```
v0=v0+v1;
v1=v1+v2;
v2=v2+v3;
v3=v3+v4;
v4=v4+v0;
v0=v0+v1;
```

Das grundsätzliche Vorgehen im dazugehörigen Kernel besteht darin Abhängigkeiten zwischen den einzelnen Befehlen so zu erzeugen, dass sie auch durch Umsortieren nicht aufgehoben werden können. Dies wird durch das Benutzen der immer gleichen Variablen, die als sich im privaten Speicher befindlich deklariert werden, also in Registern sein sollten, gesichert. Weiter oben ist ein Ausschnitt aus dem Kernel zu sehen, der bei Instruktionslatenzen von vier oder kleiner ohne Performanceeinbußen ausgeführt werden sollte. Nun kann die Anzahl der Variablen vergrößert werden, und mit ihrer Anzahl auch die Anzahl an Instruktionen die sich zeitgleich in einer Pipeline befinden. Die Ausführungsgeschwindigkeit sollte so lange zunehmen, bis die Abstände zwischen den an Variablen so groß geworden sind, dass die Pipeline gefüllt ist und die maximale Rechenleistung erreicht wird. Optimierungen des Compilers werden versucht mit der Option -cl-opt-disable beim Kompilieren des Kernels abzuschalten. Dieses Vorgehen wurde auch schon in der dazugehörigen Präsentation[04](Folie 9) der Arbeit "Demystifying GPU Microarchitecture through Microbenchmarking"beschrieben. Insgesamt wurden für LibOpenCLBench 29 Funktionen im Kernel mit dieser Funktionalität implementiert, um Instruktionslatenzen von eins bis 30 zu messen. Der Abstand von zwei Abhängigen Instruktionen fängt mit einer Funktion die den Abstand von eins hat an. Das heißt, dass ein Operand der in einer Instruktion genutzt wurde auch in der nächsten wieder genutzt wird. Diese Funktion sollte sich also nur dann mit voller Leistung ausführen lassen, wenn die Instruktionslatenz eins ist. In der nächsten Funktion ist zwischen den zwei abhängigen Instruktionen eine weitere Instruktion, die sich zeitgleich in der Pipeline befinden könnte. Allerdings sind auch diese beiden Instruktionen jeweils über einen Abstand von drei voneinander abhängig, es werden dazu drei Variablen verwendet. So ist sichergestellt, dass auch bei größeren Zahlen an Variablen keine Vertauschungen stattfinden beziehungsweise durch diese keine schnellere Abarbeitung des Programms möglich ist.

Im Host-Programm werden jeweils die Ausführungszeiten der Kernel mit den jeweiligen Abständen durch ein OpenCL-Event gemessen. In diesen Daten würde man sich einen Knick an der Stelle erwarten, an der die Pipeline voll gefüllt ist, da ab dort die Ausführungszeit nicht mehr abnehmen sollte. Die Ergebnisse der Zeitmessungen der einzelnen Ausführungen werden als Rohdaten gespeichert und können aus LibOpenCLBench abgefragt werden. Es wird allerdings auch versucht eine konkrete Instruktionslatenz aus den gesammelten Werten zu ermitteln. Dazu soll der bereits beschriebene Knick in den Daten gefunden werden. Dies geschieht dadurch, dass die Werte jeweils mit dem nächst Kleineren verglichen werden, wenn ab einer Stelle die Ausführungszeit nicht mehr absinkt, was anhand eine Schwellenwerts überprüft wird, wird hier das erreichen der Instruktionslatenz angenommen und gespeichert. Sollte die Ausführung über den gesamten Wertebereich sinken kann keine Instruktionslatenz ermittelt werden, in diesem Fall wird der Wert -1 gespeichert.

4.3.6. Herausforderungen bei der Entwicklung der Benchmarks

Bei der Entwicklung der einzelnen Benchmarks traten insbesondere im Zusammenhang mit OpenCL-Kerneln die auf GPUs ausgeführt werden mitunter zunächst nicht einfach identifizierbare Probleme auf. So führten beispielsweise Index-Variablen wenn sie als int deklariert waren zu enormen Performanceeinbrüchen, die verschwanden wenn ein unsigned int hierfür verwendet wurde.

Auch andere kleine Abweichungen zeigten, dass sie große Auswirkung auf die gemessenen Werte haben konnten. Insbesondere das Benchmark zur Bestimmung der Instruktionslatenzen fiel hierbei auf. Selbst bei mit CodeXL exportiertem Zwischencode, der so aussah, als würde er genau das umsetzen, was beabsichtigt sei, waren die Ergebnisse mitunter kontraintuitiv.

```
v_add_f32  v0, v0, v3                    // 00003460: 06000700
v_add_f32  v1, v1, v0                    // 00003464: 06020101
v_add_f32  v2, v2, v1                    // 00003468: 06040302
v_add_f32  v3, v3, v2                    // 0000346C: 06060503
v_add_f32  v0, v0, v3                    // 00003470: 06000700
```

Dieser Code-Ausschnitt sollte z.B. bei einer bei einer Instruktionslatenz von 4 Taktzyklen schnell ausgeführt werden, tatsächlich war das daraus resultierende Programm aber mit noch geringeren Abständen zwischen voneinander abhängigen Instruktionen schneller. So stieg mit größerem Abstand zwischen zwei voneinander abhängigen Instruktionen die Ausführungszeit mitunter an, anstatt zu sinken und sich ab dem Erreichen einer vollen Pipeline zu stabilisieren. Hier half oft nur das Vorgehen viele verschiedene Kernel mit jeweils kleinen Änderungen zu generieren und den Einfluss der getroffenen Maßnahmen auf das Ergebnis zu überprüfen. CPUs verhielten sich in der Entwicklung meist gutmütiger, und eher wie man es von normalen C-Programmen her gewohnt wäre.

5. Testergebnisse der Microbenchmarks und Vergleich mit Referenzen

Um die Funktion der Benchmarks zu zeigen wurden sie auf verschiedenen Geräten ausgeführt. Dazu zählt eine CPU vom Typ Intel Xeon E5-2620, die über 6 Kerne und 12 Threads verfügt und standardmäßig mit 2GHz taktet.[14] Weiter wurde eine CPU von AMD verwendet, ein A10 5800K auf 4.4GHz übertaktet der über 4 Kerne verfügt.[15] Auch eine Grafikkarte von Nvidia, die GeForce GTX 680 mit 1536 Streamprozessoren[16] und eine Beschleunigerkarte vom Typ Nvidia Tesla K20X, welche 2688 Streamprozessoren besitzt[17] wurden genutzt. Am häufigsten wurden die Benchmarks auf einer Grafikkarte von AMD, der R9 280X, getestet. Diese GPU verfügt über 2048 Streamprozessoren[18]. Besonders an dieser Karte für eine GPU aus dem Consumer-Bereich ist die hohe Leistung bei Berechnungen mit doppelter Genauigkeit, sie erreicht hier bis zu einem Teraflop. Für diese Karte befinden sich auch alle Ergebnisse der LibOpenCLBench als Anhang am Ende der Arbeit. Die Testergebnisse, gerade auf Grafikkarten, schwankten auch bei anderen Benchmarks als LibOpenCLBench zwischen verschiedenen Ausführungen mitunter etwas. Um möglichst exakte Ergebnisse zu erhalten empfiehlt es sich deshalb sofern es möglich ist den Computer frisch zu booten und keine anderen Programme neben den Benchmarks auszuführen.

5.1. Ergebnisse der Benchmarks zur Bestimmung der Rechenleistung

Die Herstellerangaben zur Rechenleistung mit Fließkommazahlen haben sich hier als sehr gute Referenz gezeigt, sowohl LibOpenCLBench als auch andere Benchmarks konnten die Werte reproduzieren. Die höchsten Leistungen wurden meist in den Teilen des Benchmarks erreicht, in denen die kombinierte Multiplikation und Addition berechnet wurde. LibOpenCLBench konnte hier beispielsweise 3945,6GFLOPS mit der AMD R9 280X und Berechnungen einfacher Genauigkeit erreichen, das SHOC-Benchmark MaxFlops erreichte 4021,6GFLOPS, die Herstellerangabe liegt hier bei 4096GFLOPS. Auch die Beschleunigerkarte und GPU von Nvidia konnten an das theoretische Maximum heranreichen.

Ähnlich sieht es bei den Berechnungen mit doppelter Genauigkeit aus. LibOpenCLBench kam für die AMD R9 280X auf eine Geschwindigkeit von 928,6GFLOPS, das SHOC-Benchmark auf 949,77GFLOPS bei einer Herstellerangabe von 1024GFLOPS. In diesem Benchmark war die Nvidia Tesla K20X die schnellste Karte mit 1305,6GFLOPS, und damit nur 5 GFLOPS unter der angegebenen Leistung. Die GeForceGTX680 brach im Vergleich zu den Berechnungen mit einfacher Genauigkeit stark ein, dies ist allerdings auch vom Hersteller so spezifiziert. Die CPUs konnten dagegen auch bei doppelter Genauigkeit fast die gleiche Anzahl an Berechnungen durchführen wie bei einfacher Genauigkeit, insbesondere die Intel CPU tat sich hier hervor. Die CPUs müssen sich aber in der

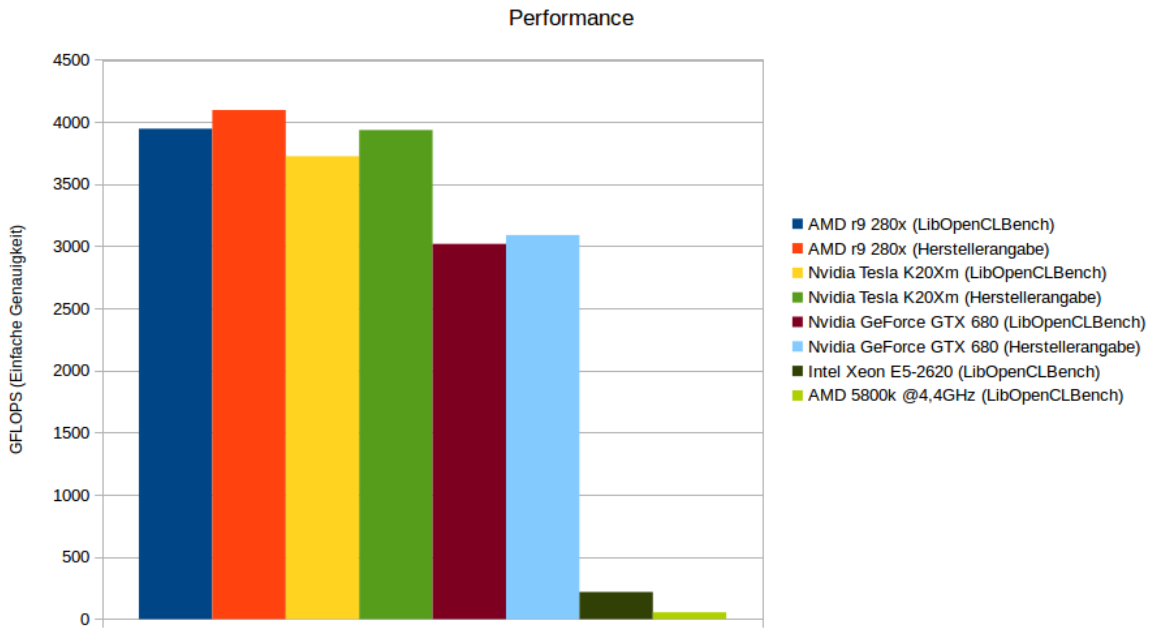

Abbildung 5.1.: Ergebnisse bei einfacher Genauigkeit und Herstellerangaben zum Vergleich

Gesamtrechenleistung den ebenfalls getesteten GPUs deutlich geschlagen geben, bei Berechnungen mit einfacher Genauigkeit ist z.B. die AMD-CPU um zwei Größenordnungen langsamer als die AMD-Grafikkarte.

Damit die AMD-CPU allerdings überhaupt eine hohe Ausführungsgeschwindigkeit erreicht muss sich der Programmierer selbst um die Vektorisierung der Werte kümmern, wie die Ergebnisse mit verschiedenen Vektorbreiten zeigen wird dies nämlich nicht vom Compiler automatisch erledigt. Der Einfluss der Vektorbreite ist zwar auch bei GPUs bemerkbar, allerdings lange nicht so deutlich. Die R9 280x erreichte in bei einer Vektorbreite von eins immer noch rund 85% der maximalen Rechenleistung bei der MAD-Operation.

5.2. Ergebnisse der Benchmarks zur Bestimmung Speicherbandbreiten

Mit 253,2GB/s konnte die AMD r9 280 beim Test mit LibOpenCLBench lesen, angegeben sind hier vom Hersteller 288GB/s. Das Benchmark von AMD war an dieser Stelle allerdings etwas langsamer und erreichte 220,2GB/s. Auch die anderen getosten Geräte konnten im Benchmark nicht ganz die Leistungen erreichen die die Hersteller versprochen hatten. Die GeForce GTX 680 erreichte 147,7GB/s, theoretisch ist sie mit 192GB/s angegeben, die Tesla K20X erreichte 220,9GB/s, hier sind 250GB/s der angegebene Wert. Diese Werte wurden alle so ermittelt, dass viele Cache-Hits vermieden werden. Mit der Nutzung des Caches und dem häufigen Lesen der selben Speicherstellen konnte die AMD r9 280X

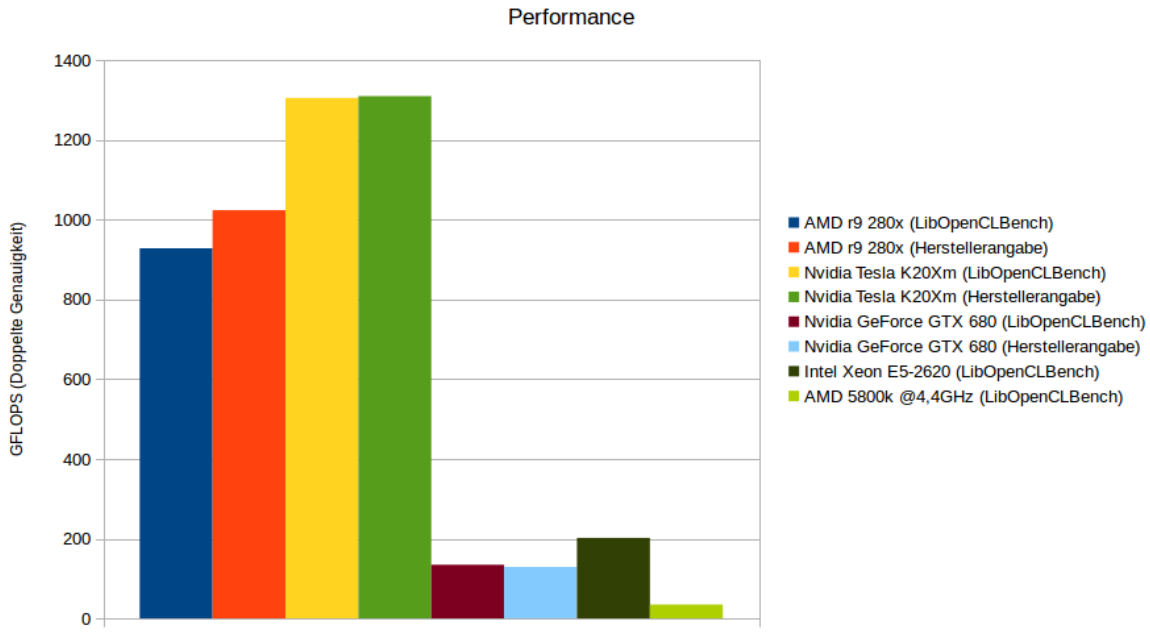

Abbildung 5.2.: Ergebnisse bei doppelter Genauigkeit und Herstellerangaben zum Vergleich

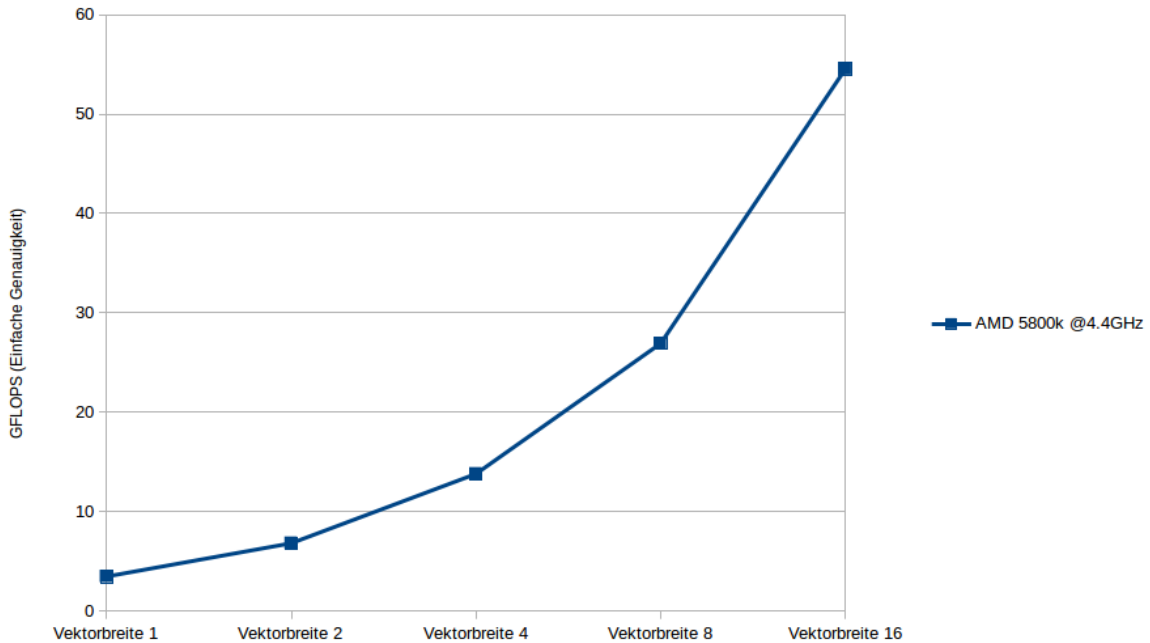

Abbildung 5.3.: Geschwindigkeitszuwachs durch manuelle Vektorisierung bei AMD-CPU

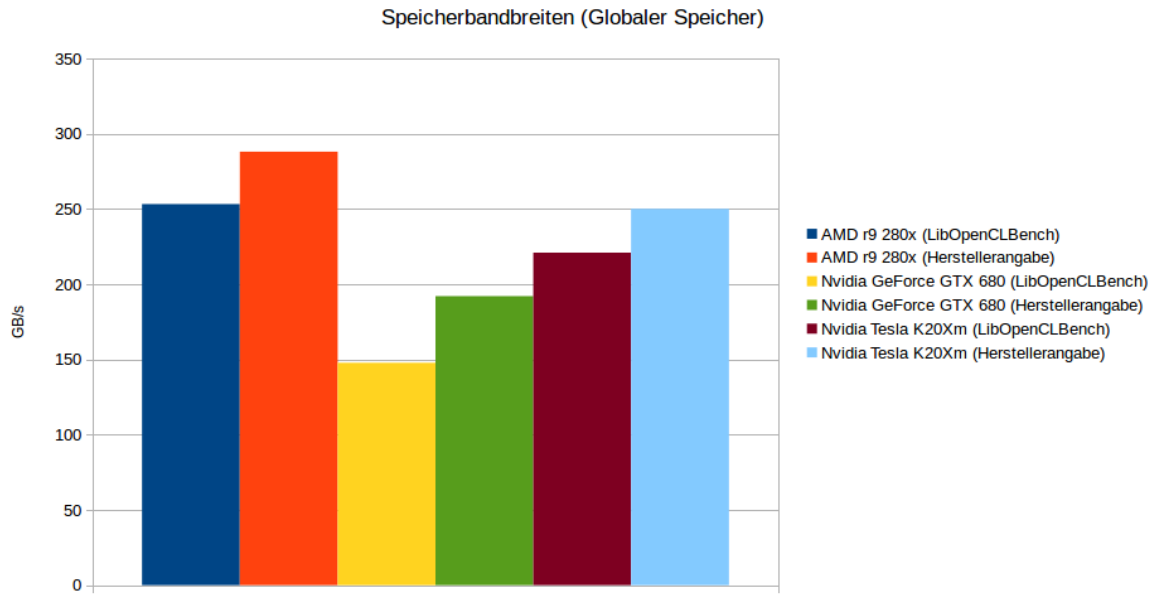

Abbildung 5.4.: Speicherbandbreiten von GPUs und Beschleunigerkarte sowie Herstellerangaben zum Vergleich

ihre Lesegeschwindigkeit auf 1,581TB/s steigern. Noch höhere Bandbreiten erreichte nur der lokale Speicher, hier waren 3,8TB/s möglich.

Wie wichtig das Coalescing bei Speicherzugriffen der GPU ist zeigten die randomisierten Speicherzugriffe. Hier brach die Bandbreite in LibOpenCLBench mit der r9 280X auf 58,8GB/s ein, das Benchmark von AMD konnte bei zufällig verteilten Speicherzugriffen hier noch 56,0GB/s zählen. Dies zeigt, dass man bei der Entwicklung von OpenCL-Kerneln und der Nutzung des globalen Speichers dringend darauf achten sollte, dass die Werte in einer geeigneten Reihenfolge im Speicher liegen, damit solche Einbrüche bei den Leseraten nicht auftreten. Dieser Effekt war allerdings bei CPUs nicht zu beobachten, der Durchsatz konnte hier mit randomisierten Zugriffen ähnlich hoch oder sogar höher sein.

5.3. Ergebnisse des Benchmarks zur Bandbreite zwischen Host und OpenCL-Device

Daten vom Host zum Device zu verschieben ist langsamer als sie wieder von diesem zu lesen, das war das Resultat der Ausführung dieses Benchmarks. Die maximale Bandbreite der PCI-Express-2.0-Schnittstelle mit 16 Lanes, über welche die AMD-Grafikkarte angebunden ist, beträgt 8GB/s. Tatsächlich erreicht wurden hier von LibOpenCLBench 6556MB/s beim Lesen vom Device zurück in den Speicher des Hosts und 5390MB/s beim Schreiben von Daten in den Speicher der Grafikkarte.

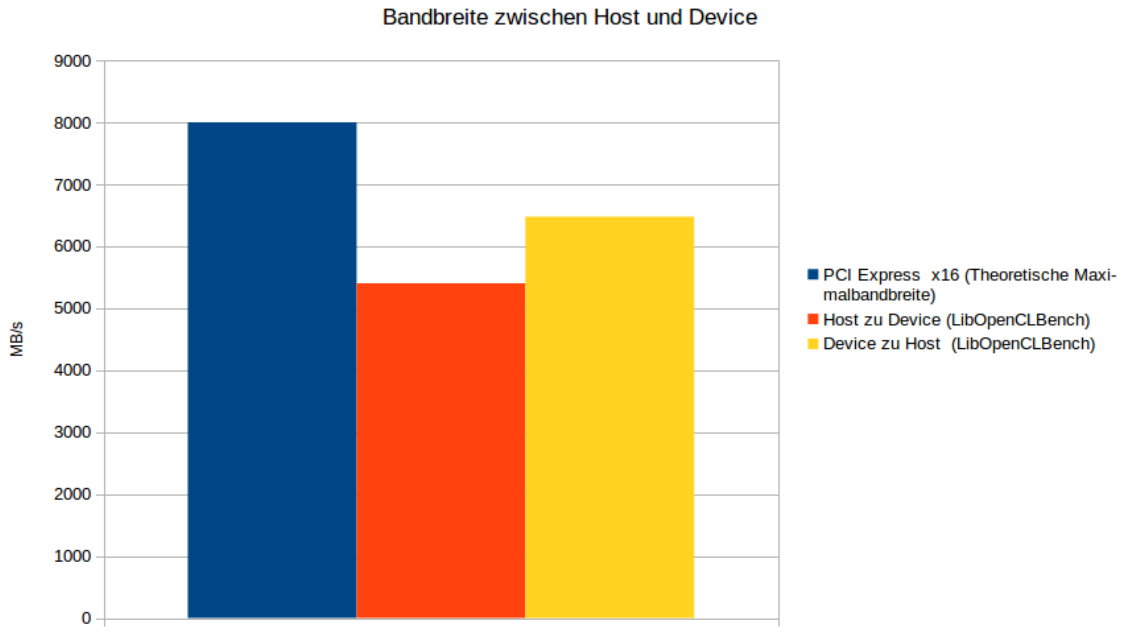

Abbildung 5.5.: Erreichte Bandbreiten und maximale Bandbreite der PCI-Express-Schnittstelle zum Vergleich

Die SHOC-Benchmarks konnten hier 6898MB/s respektive 5893MB/s für übertragene Speicherelemente der gleichen Größe erreichen. Diese Werte sind deutlich unter denen der Speicherbandbreiten innerhalb von Grafikkarten und Beschleunigerkarten, man sollte unnötiges Verschieben von Daten zwischen Host und Device also nach Möglichkeit vermeiden.

5.4. Ergebnisse des Benchmarks zur Prüfung des Einflusses von vielen Verzweigungen im Programmcode

Wie erwartet zeigte sich bei diesem Benchmark, dass die Logik der CPUs zur Sprungvorhersage einwandfrei funktioniert. So konnte mit der AMD-CPU nur eine Verlangsamung um den Faktor 1,00685 gemessen werden. Auf Grafikkarten kann aber eine merkliche Verlangsamung gesehen werden, die AMD R9 280X benötigte etwa 40% mehr Zeit um das Microbenchmark auszuführen.

5.5. Ergebnisse des Benchmarks zu Bestimmung von Instruktionslatenzen

Die Ausführungszeiten dieses Microbenchmarks zeigen deutlich, dass die Instruktionslatenz, hier am Beispiel der Additionsoperation, auf GPUs wie der AMD R9 280X mit knapp 20 Taktzyklen deutlich höher ist als etwa die Instruktionslatenz der AMD-CPU, welche sich im höheren einstelligen Bereich an

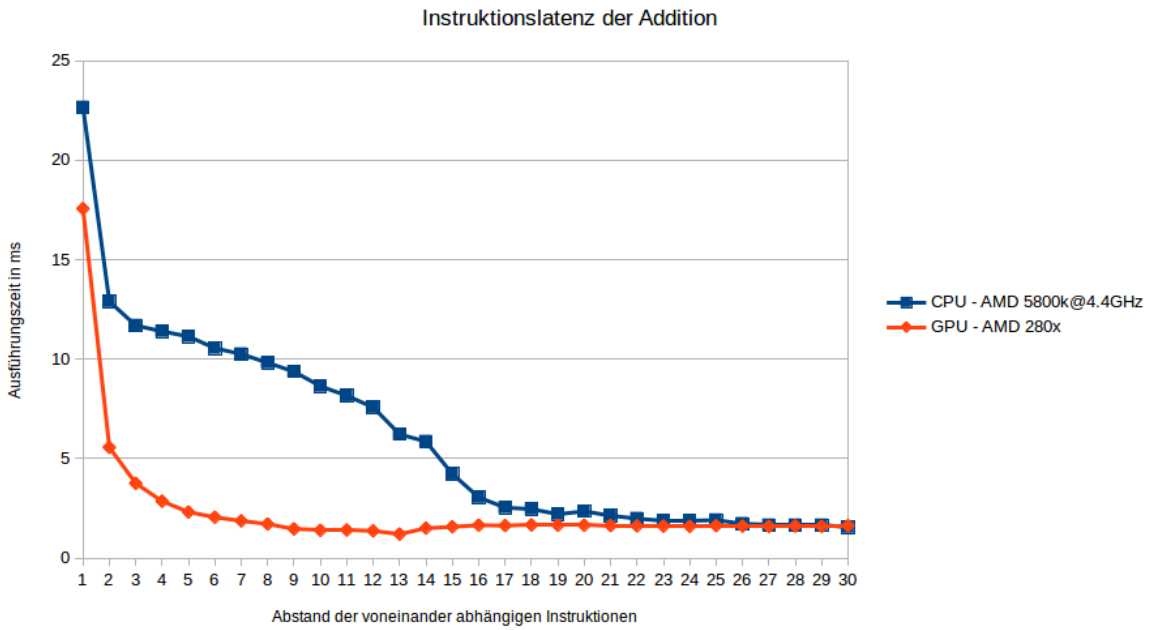

Abbildung 5.6.: Verlauf der Ausführungsdauer hinsichtlich des Abstands untereinander abhängiger Instruktionen

Taktzyklen bewegt. Dieses Benchmark ist leider etwas störanfällig was die genaue Bestimmung einer Instruktionslatenz anbelangt, wie man an den Daten sehen kann. Die grundlegenden Zusammenhänge von Instruktionslatenzen sollten allerdings erkennbar sein. Damit kann man auch für die Erstellung von eigenen OpenCL-Kernels Konsequenzen ziehen und darauf achten, dass Abhängigkeiten zwischen einzelnen Operationen nach Möglichkeit zu vermeiden sind, da ansonsten gerade bei Abständen von nur einer oder zwei Instruktionen zwischen den untereinander abhängigen Ausführungen mit starken Verlangsamungen der Ausführungsgeschwindigkeit zu rechnen ist.

6. Fazit

Im Rahmen dieser Arbeit wurde darauf eingegangen, wie mit OpenCL Informationen über die Stärken und Schwächen unterschiedlicher Gerätetypen erlangt werden kann. Die über die ausgelesenen Parameter oder über die Softwarebibliothek LibOpenCLBench bereitgestellten Microbenchmarks können fortan genutzt werden, um die Performance von OpenCL-Kerneln schneller an gerätespezifische Eigenschaften anpassen zu können. So kann die Stärke von OpenCL, auf vielen unterschiedlichen Plattformen lauffähig zu sein, mit hoher Leistungsfähigkeit verbunden werden.

7. Ausblick

Als zukünftigen Aufgabe könnte es sich anbieten die verschiedenen vorgestellten Benchmark-Sammlungen quasi zu vereinen, und eine Implementierung so zu erweitern, dass sie alle relevanten Faktoren der Hardware erfassen kann. LibOpenCLBench könnte etwa noch um die Fähigkeit zur Messung von Speicherlatenzen erweitert werden. Auch eine Verbesserung der Benchmarks für Instruktionslatenzen wären wünschenswert.

Die direkt Erfassung von Parametern ohne Benchmarks könnte z.B. um eine Datenbank an Parametern für bekannte Geräte erweitert werden, wo auch Informationen Platz finden, die nicht über die OpenCL-API auslesbar sind. Um bei diesen Daten nicht das Problem zu haben, dass sie an eine Softwareversion gebunden sind, die mit der Veröffentlichung einer neuen Hardwaregeneration nicht mehr aktuell ist, wäre es auch denkbar eine öffentliche Datenbank zu erstellen, die über das Internet abgefragt werden kann und jeweils auch die Informationen zu den neusten Geräten enthält.

Von weiterem Interesse könnten Benchmarks sein, die auf weitere Eigenheiten der Geräte eingehen, welche beispielsweise physischer Natur sind, also etwa die Kühlung betreffen. So kommt es gerade bei Grafikkarten mittlerweile vor, dass die Kühllösungen nicht für den Dauerbetrieb unter maximaler Last dimensioniert sind, und sie sich entsprechend nach einiger Zeit heruntertakten. Manche Geräte bieten im Gegenteil dazu etwa die Nutzung von Boosts an, mit dem die Taktfrequenz für eine gewisse Zeit erhöht wird. Eine genaue Erfassung dieser Eigenschaften ließe gegebenenfalls eine Steigerung der maximalen Leistungsfähigkeit über einen längeren Zeitraum gesehen zu.

Denkbar wäre als Anwendung der durch die Benchmarks und anderweitig erlangten Daten, dass diese genutzt werden um automatisch das passende OpenCL-Device für einen Algorithmus auszuwählen und die Ausführung auf diesem zu starten. Neben dieser angepassten Wahl des Geräts nach dem Algorithmus wäre natürlich aber auch anders herum eine automatische Anpassung von Algorithmen an die Eigenschaften der OpenCL-Devices interessant. So könnte eine hohe Performance-Portabilität erreicht werden, wobei der Programmierer nicht im Detail alle Eigenschaften der Hardware kennen muss. Dies würde es ermöglichen mehr Zeit in die Lösung eines Problems auf höherer Ebene zu investieren.

A. Anhang

A.0.1. Benchmarkergebnisse für die AMD R9 280X

Kernelname	Performance in GFLOPS
Add1	1860.15GFLOPS
Add2	1845.05GFLOPS
Add4	1779.92GFLOPS
Add8	1909.96GFLOPS
Add16	1693.76GFLOPS
Mul1	1861.16GFLOPS
Mul2	1805.72GFLOPS
Mul4	1657.3GFLOPS
Mul8	1893.5GFLOPS
Mul16	1712.36GFLOPS
MulAdd1	3371.84GFLOPS
MulAdd2	3598.01GFLOPS
MulAdd4	3872.53GFLOPS
MulAdd8	3712.09GFLOPS
MulAdd16	3945.65GFLOPS

Tabelle A.1.: Rechenleistung bei einfacher Genauigkeit

Kernelname	Performance in GFLOPS
Add1	839.003GFLOPS
Add2	913.158GFLOPS
Add4	855.536GFLOPS
Add8	930.283GFLOPS
Add16	891.458GFLOPS
Mul1	419.287GFLOPS
Mul2	414.547GFLOPS
Mul4	403.001GFLOPS
Mul8	466.747GFLOPS
Mul16	406.102GFLOPS
MulAdd1	911.165GFLOPS
MulAdd2	876.611GFLOPS
MulAdd4	829.229GFLOPS
MulAdd8	928.558GFLOPS
MulAdd16	884.622GFLOPS

Tabelle A.2.: Rechenleistung bei doppelter Genauigkeit

Kernelname	Bandbreite in MB/s
resultGlobalReadLin	253.226 MB/s
resultGlobalReadwCache	1581.25 MB/s
resultGlobalReadRand	58.794 MB/s
resultGlobalWriteLin	1640.25 MB/s
resultLocalReadLin	3801.85 MB/s

Tabelle A.3.: Gemessene Speicherbandbreiten

Host zu Device	Device zu Host
5389.87MB/s	6556.08MB/s

Tabelle A.4.: Geschwindigkeit der Datenübertragung zwischen Host und Device

AMD R9 280X	AMD A10 5800K (CPU)
1.40336	1.00685

Tabelle A.5.: Faktor der Ausführungsdauer bei vielen Verzweigungen

Abstand zwischen zwei voneinander abhängigen Instruktionen	Ausführungsdauer
1	22,5622
2	12,899
3	11,6902
4	11,3957
5	11,1388
6	10,5501
7	10,2561
8	9,81496
9	9,69452
10	8,64904
11	8,74548
12	7,25111
13	6,21378
14	6,07644
15	4,22978
16	3,05348
17	2,53926
18	2,46667
19	2,90415
20	2,24607
21	2,13452
22	1,98815
23	1,87748
24	2,336
25	1,9163
26	1,73185
27	1,65674
28	1,65704
29	1,6563
30	1,54667

Tabelle A.6.: Rohdaten des Benchmarks zur Bestimmung von Instruktionslatenzen

Literaturverzeichnis

[01] The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite https://github.com/vetter/shoc
[02] OpenCL-Benchmarks aus den AMD-Beispielen die dem AMD APP SDK beiliegen http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/
[03] M. Bannerman, S. Strobl, T. Pöschel. Supercomputing on Graphics Cards. 2011 https://faui36a.informatik.uni-erlangen.de/trac/puppet/wiki/OpenCL (Links zu den Folien zum derzeit nicht mehr erreichbar)
[04] H.Wong. Demystifying GPU Microarchitecture through Microbenchmarking (Präsentation), 2010 http://www.stuffedcow.net/files/gpuarch_presentation.pdf
[05] P. Thoman, K. Kofler, H. Studt, J. Thomson, T. Fahringer. Automatic OpenCL Device Characterization: Guiding Optimized Kernel Design. In Euro-Par 2011 Parallel ProcessingLecture Notes in Computer Science Volume 6853, 2011, Seiten 438-452
[06] uCLbench http://www.insieme-compiler.org/uclbench.html
[07] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In Performance Analysis of Systems and Software (ISPASS), 2010 IEEE International Symposium, 2010, Seiten 235-246
[08] X. Yan, X. Shi, Q. Sun. An OpenCL Micro-Benchmark Suite for GPUs and CPUs. In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2012, Seiten 53-58
[09] G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, J.S. Vetter. The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite. In GPGPU '10 Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, 2010, Seiten 63-74
[10] Y. Zhang, M. Sinclair, A.A. Chien Improving performance portability in OpenCL programs Y Zhang, M Sinclair II, AA Chien - Supercomputing, 28th International Supercomputing Conference, ISC 2013, Leipzig, Germany, June 16-20, 2013. Proceedings, 2013, Seiten 136-150
[11] MPBenchmarks http://www.bealto.com/gpu-benchmarks.html
[12] AIDA64 Extreme http://www.aida64.com/downloads
[13] SiSoftware Sandra http://www.sisoftware.co.uk/?d=qa&f=gpgpu_gpu_perf
[14] Eigenschaften der CPU Intel Xeon E5-2620 http://ark.intel.com/de/products/64594/Intel-Xeon-Processor-E5-2620-15M-Cache-2_00-GHz-7_20-GTs-Intel-QPI
[15] Eigenschaften der CPU AMD A10 5800K https://de.wikipedia.org/wiki/AMD_Fusion#Modelle_f.C3.BCr_den_Desktop_2
[16] Eigenschaften der GPU Nvidia GeForce GTX 680 http://www.nvidia.de/object/geforce-gtx-680-de.html#pdpContent=2
[17] Eigenschaften der Beschleunigerkarte Nvidia Tesla K20X http://www.nvidia.de/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf
[18] Eigenschaften der GPU AMD R9 280X https://de.wikipedia.org/wiki/AMD-Radeon-R200-Serie#Modelldaten

[19] The OpenCL Specification Version 1.2 https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
Verwendete LaTeX-Vorlage für dieses Dokument https://github.com/latextemplates/uni-stuttgart-compu

Alle URLs wurden zuletzt am 21. 06. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift