Hochschule für Technik, Wirtschaft und Kultur Leipzig
Fakultät für Informatik, Mathematik und Naturwissenschaften

Masterprojekt

Untersuchung der OpenCL-Programmierplattform zur Nutzung für rechenintensive Algorithmen

B.Sc. Christof Pieloth

25. August 2011

Inhaltsverzeichnis

1 Einleitung

Am Beispiel der Open Computing Language (OpenCL) wird General Purpose Computation on Graphics Processing Unit (GPGPU) untersucht. Mit dieser Technologie kann die Grafikkarte neben ihrer Hauptaufgabe, der Berechnung zur Bildausgabe, für allgemeine Berechnungen, welche zuvor nur von Hauptprozessoren (CPUs) ausgeführt werden konnten, genutzt werden.

Im Vergleich zur Programmierung von CPUs gibt es bei Grafikkarten einige Besonderheiten und Einschränkung, welche in dieser Arbeit beschrieben werden. Neben den Aspekten der Parallelisierung eines Algorithmus unterscheiden sich vorallem das Speicher- und Programmiermodell bei Grafikprozessoren (GPUs) von dem der „klassischen" Modelle.

Außerdem wird die Einrichtung der Entwicklungumgebung von AMD „Accelerated Parallel Processing" (APP) und der allgemeine Ablauf einer OpenCL-Anwendung beschrieben. Eine Matrixmultiplikation dient als Beispiel um das Potenzial von GPU-basierten Algorithmen zu verdeutlichen. Hierbei werden die Laufzeiten der CPU- und GPU-basierten Implementierungen gegenübergestellt und analysiert.

1.1 Relevanz und Anlass der Untersuchung

Den Anlass der Untersuchung gab die fokus GmbH Leipzig. Das Ingenieurbüro bietet Dienstleistungen in den Bereichen der Bauvermessung, Photogrammetrie, Bildverarbeitung und Softwareentwicklung an. Hierzu vertreibt und entwickelt die fokus GmbH firmeneigene Software.

Im Bereich der digitalen Photogrammetrie werden unter Verwendung hochaufgelöster Bilder Objektmaße ausgewertet. Bei der Auswertung von Bild- und Punktdaten kommen rechenintensive Algorithmen zum Einsatz, wie z.B. Matching- und Mustererkennungsalgorithmen, Triangulationsalgorithmen, Algorithmen für Entzerrungen und Orthoprojektionen.

Zum aktuellen Zeitpunkt finden während der Laufzeit solcher Algorithmen die Prozessoren der Grafikkarte keine Verwendung. Diese könnten jedoch zur Berechnung genutzt werden, um die Bearbeitung zu beschleunigen und somit sogar die Nutzung von Algorithmen zu ermöglichen, welche bisher als zu komplex galten.

1.2 Gegenstand der Untersuchung

In dieser Arbeit wird das sogenannte General Purpose Computation on Graphics Processing Unit untersucht. GPGPU kann als „Allgemeine Berechnung auf Grafikprozessoren" übersetzt werden und bezeichnet die Verwendung des Grafikprozessors einer Grafikkarte für Berechnungen außerhalb seines eigentlichen Aufgabenbereichs, die Berechnung der Bildschirmausgabe.

Genauer wird die Technologie OpenCL betrachtet. Die Untersuchung der Technologie beinhaltet verschiedene Faktoren wie z.B. Einschränkungen gegenüber der Programmiersprache C++, die Performance im Vergleich zur CPU, Werkzeugunterstützung und mehr.

1.3 Zielsetzung

Es ist eine Technologierecherche über OpenCL für den Einsatz in der Software der fokus GmbH durchzuführen. Folgende Aspekte sind zu untersuchen:

- Unterstützung der Programmiersprache C++ und der C++ Standard Library
- Programmiersprache der Grafikprozessoren
- Einschränkungen
- Unterschiede zur CPU-Programmierung und Optimierungsansätze
- Unterstützte Entwicklungsumgebungen, Programme und Compiler
- Plattformabhängigkeiten
- Performanceunterschiede zwischen GPU und CPU

Abschließend sind die erarbeiteten Erkenntnisse zusammenzufassen. Weiterhin ist eine Anleitung zur Einrichtung einer Entwicklungsumgebung und ein Beispielprogramm zu erstellen.

1.4 Aufbau der Arbeit

Um die Unterschiede zwischen CPUs und GPUs, sowie deren Programmierung, darzulegen, werden zu Beginn dieser Arbeit das allgemeine Speicher- und Programmiermodell von GPGPU erläutert. Anschließend wird die Technologie OpenCL genauer vorgestellt und abschließend wichtige Punkte zusammengefasst. Darauffolgend wird jeweils eine Beispielimplementierung für die GPU und CPU vorgestellt und mit Hilfe von Laufzeitmessungen verglichen.

Bei der Beschreibung von Quellcode und Installationsanleitungen werden spezielle Schreibweisen verwendet, um die Übersicht zu erhöhen. In *Kursivschrift* markierte Begriffe oder Sätze, sind von Programmen aus Menüpunkten, Auswahlen o.ä. übernommene Inhalte. Befehle, Programmausgaben und Dateiinhalte sind in `Maschinenschrift` dargestellt. Längere Passagen sind vom Fließtext gelöst und grau unterlegt:

```
echo 'hello world' >> hello.txt
cat hello.txt
```

Des Weiteren werden bei der allgemeinen Erklärung von Befehlen Variablen genutzt. Diese sind durch spitze Klammern gekennzeichnet und müssen bei der Anwendung durch einen entsprechenden Wert ersetzt werden. Der Name der Variable beschreibt deren Funktion. Zum Beispiel benötigt der Befehl zum Erstellen eines Ordners den Ordnernamen als Argument:

```
mkdir <directoryname>
```

Um einen Ordner mit dem Namen „foo" zu erstellen muss die Variable `<directoryname>` durch `foo` ersetzt werden:

```
mkdir foo
```

Werte in eckigen Klammern sind mögliche Optionen und können entweder gesetzt oder nicht gesetzt werden, wobei die Klammern bei Verwendung nicht mit angegeben werden.

1.5 Quellen

Alle wichtigen Informationen über GPGPU werden hauptsächlich durch diese drei Quellen bereitgestellt:

- NVIDIAs CUDA Zone [1]
- AMDs OpenCL Zone [2]
- OpenCL-Spezifikation der Khronos Group [3]

Besonders die Entwicklerportale von NVIDIA und AMD bieten viele Dokumente und Quellcodebeispiele. Bei der Suche auf dem Portal von AMD ist darauf zu achten, dass ATI Stream zu AMD APP umbenannt wurde und dadurch einige Informationen noch unter ATI Stream zu finden sind.

2 General Purpose Computation on Graphics Processing Unit

General Purpose Computation on Graphics Processing Unit (GPGPU) bedeutet übersetzt „Allgemeine Berechnung auf Grafikprozessoren". Die vereinfachte Aufgabe einer Grafikkarte ist die Steuerung der Bildschirmausgabe in einem Computer. Der Begriff „allgemeine Berechnung" ist im Zusammenhang mit der Aufgabe einer Grafikkarte in einem Computer zu betrachten. Im Gegensatz zu Hauptprozessoren, die einen möglichst großen Umfang an Funktionen bereitstellen sollen, sind die Funktionen und die damit verbunden Berechnungen eines Grafikprozessors speziell auf dieses Einsatzgebiet zugeschnitten. Aufgrund dieser technischen Einschränkungen war es mit den ersten Grafikkarten nicht annähernd möglich vergleichbare Berechnungen, wie die eines Hauptprozessors, durchzuführen.

Mit der Entwicklung von grafischen Benutzeroberflächen und 3D-Grafik erweiterte sich auch der Funktionsumfang und die Leistung von Grafikkarten. Im Jahr 2000 wurde mit Microsofts Direct3D 8.0 das Konzept von Shadern eingeführt [4]. Diese ermöglichen die hardwareunabhängige Programmierung zur Berechnung von zum Beispiel Beleuchtungseffekten oder Geometriedaten. Durch die Weiterentwicklung dieses Konzepts konnten auf Grafikkarten zunehmend immer aufwendigere Operationen umgesetzt werden.

Da die Berechnungen von Grafikkarten ein hohes Potenzial an Parallelität bieten, verlief die Hardwareentwicklung entscheidend anders als bei Hauptprozessoren. So können zum Beispiel Bildpunkte als Vektoren mit drei Werten für Rot, Grün und Blau dargestellt werden. Soll nun ein Bildpunkt $p_1 =< r_1, g_1, b_1 >$ den Bildpunkt $p_2 =< r_2, g_2, b_2 >$ durch die Funktion $p_1 \cdot p_2 =< r_1 \cdot r_2, g_1 \cdot g_2, b_1 \cdot b_2 >$ modifizieren, kann dies unter Nutzung des SIMD-Prinzips[1] in einem Rechenschritt durchgeführt werden. Diese Eigenschaft nutzten die Hersteller bei der Entwicklung immer leistungstärkeren und hochparallelisierter Grafikkarten massiv aus. Aktuelle Grafikkarten besitzen mehrere hundert Shader-Einheiten [5].

Im Gegensatz dazu fand unter den Hauptprozessorherstellern ein regelrechtes „Megahertz-Rennen" statt. Aufgrund der steigenden Taktfrequenz und der kleiner werdenden Strukturen schien vor allem die hohe Wärmeentwicklung die Leistungsgrenzen der damaligen Prozessoren aufzuzeigen. Als Ausweg aus dieser Entwicklung stellte AMD im Jahr 2004 den ersten Dual-Core-Prozessor mit x86-Befehlssatz vor [6]. Aktuelle Multicore-Prozessoren besitzen bis zu acht physikalische Kerne [7]. Im Vergleich mit aktuellen Grafikkarten ist dies jedoch eine sehr geringe Anzahl.

2.1 Technologien

Der Begriff GPGPU beinhaltet alle Technologien, die es ermöglichen Grafikkarten auch zur Berechnung außerhalb des Anwendungsgebietes der Bildschirmausgabe zu nutzen. Zur Zeit gibt hauptsächlich drei verbreitete Technologien

- CUDA von NVIDIA,
- OpenCL von Apple und der Khronos Group und

[1] Single Instruction Multiple Data

- Direct Compute von Microsoft.

Aufgrund von cleveren Marketing seitens NVIDIA und guten Entwicklungswerkzeugen zur Zeit der Markteinführung ist CUDA bei Entwicklern sehr populär. Aber auch OpenCL ist durch den offenen Standard und der Hardwareunabhängikeit weit verbreitet. Hingegen findet Microsofts Direct Compute beim High-Performance-Computing und in Spezialsoftware kaum Anwendung.

2.2 Programmiermodell

Ein GPGPU-Programm enthält zwei Bestandteile: das Host-Programm und den Kernel. Als Host wird das System bezeichnet, auf dem die Anwendung ausgeführt wird, welche die Grafikkarte nutzt. Der Code, welcher auf einer oder mehreren Grafikkarten ausgeführt wird, heisst Kernel. Ein Kernel ist immer in einer Host-Anwendung eingebettet und kann nicht als eigenständiges Programm ausgeführt werden.

Die Aufgabe des Host ist es, die Ressourcen zu organisieren, indem er die vorhandenen Grafikkarten abfragt und die Aufgaben sowie Daten auf diese verteilt. Nach erfolgreicher Berechnung werden die Daten aus dem Grafikspeicher in dem Hauptspeicher geladen und ggf. weiterverarbeitet.

Bei GPGPU wird hauptsächlich die Datenparallelität ausgenutzt, d.h. der selbe Code wird auf unterschiedlichen, voneinander unabhängigen Daten ausgeführt. Dieser Code wird als Kernel bezeichnet und implementiert den Algorithmus für den Grafikprozessor. Zur Ausführung eines Kernels muss ein Indexraum angegeben werden, um die Anzahl der Kernel-Instanzen festzulegen. Jede Kernel-Instanz wird als Work-item bezeichnet und ist genau einem Index, die Gobal ID, zugewiesen.

Zusätzlich können Work-items in Work-groups organisiert werden, dessen Indexraum die selbe Dimension wie die der Work-items besitzen muss. Jede Work-group besitzt eine Group ID aus ihrem Indexraum. Neben der eindeutigen Global ID, besitzt jedes Work-item eine Local ID. Die Local ID ist innerhalb einer Gruppe eindeutig. Mit Hilfe der Local und der Group ID kann wiederum die eindeutige Global ID eines Work-items berechnet werden.

In Abbildung 2.1 auf der nächsten Seite ist ein globaler 2-dimensionaler Indexraum (G_x, G_y) abgebildet. Die Anzahl der Work-items ergibt sich aus dem Produkt von G_x und G_y. Zusätzlich sind alle Work-items in Work-groups mit dem Indexraum (S_x, S_y) organisiert. Die Anzahl der Work-items einer Work-group ist das Produkt aus S_x und S_y. Die Anzahl der Work-groups pro Dimension kann durch $(W_x, W_y) = (\frac{G_x}{S_x}, \frac{G_y}{S_y})$ berechnet werden.

2.3 Speichermodell

In Abbildung 2.2 auf Seite 9 ist das allgemeine Speichermodell einer Grafikkarte dargestellt. Als „Compute Device" wird die Grafikkarte bezeichnet. Das System, in dem ein oder mehrere Compute Devices arbeiten, heisst „Host"[2]. „Host Memory" bezeichnet den Hauptspeicher eines Computers. Kein Work-item hat direkten Zugriff auf diesen Speicher[3]. Aus dem Host Memory kann der Host Daten in den „Global Memory" und „Constant Memory" kopieren. Alle Work-items können Daten aus dem Gobal memory lesen und schreiben, jedoch keinen Speicher allokieren. Im Constant memory können alle Work-items Speicher allokieren und Daten lesen.

Wie in Abschnitt 2.2 beschrieben können mehrere Work-items in Work-groups zusammengefasst werden. Alle Work-items einer Work-group teilen sich einen gemeinsamen

[2] Diese Terminologie wird genauer in Kapitel 3 auf Seite 11 erklärt.
[3] Ausnahmen gibt es bei OpenCL-Implementierungen.

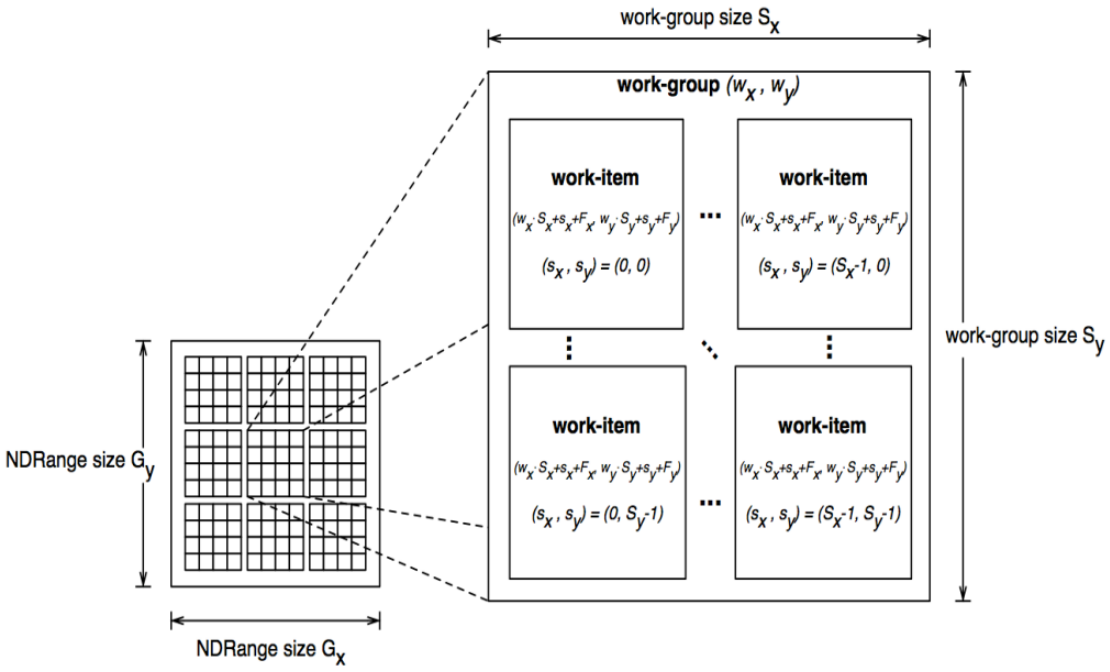

Abbildung 2.1: Aufteilung von Work-items in Work-groups. (Quelle: Khronos Group)

„Local Memory" mit Lese- und Schreibzugriff. Jedes Work-item kann Speicher im Local Memory allokieren, der anschließend von allen Work-items dieser Work-group genutzt werden kann. Jedes Work-item besitzt seinen eigenen „Private Memory" auf den es vollen Zugriff hat.

Der Host hat auf den Local und Private Memory keinen Zugriff. Außerdem sollte beachtet werden, dass der Global bzw. Constant Memory langsamer als der Local Memory ist. Genaue Optimierungsmöglichkeiten werden in Abschnitt 2.4 erklärt.

2.4 Performanceoptimierungen

Aufgrund der speziellen Architektur von Grafikkarten im Vergleich zu Hauptprozessoren und den dazu gehörigen Hauptspeicher können signifikante Laufzeitverbesserung durch Performanceoptimierungen erzielt werden. Diese Optimierungen betreffen überwiegend den Speicherzugriff und den Programmablauf. Folgend werden wichtige und leicht umsetzbare Optimierungen kurz aufgezeigt.

Datentransfer zwischen Host Memory und Global Memory reduzieren

Der Datentransfer zwischen Host Memory und Global Memory ist im Vergleich zum Datentransfer zwischen dem Global Memory und der GPU sehr langsam. Eine NVIDIA GeForce GTX 280 besitzt eine Datentransferrate von 141 GBps, wobei die Verbindung der Grafikkarte über PCIe x16 nur 8 GBps beträgt [8].

Wie bei verteilten Systemen gilt der Leitsatz „Datentransfers so wenig wie möglich und wenn soviel wie möglich". So sollten viele kleine Transfers zu einem großen zusammengefasst werden. Um die zu übertragende Datenmenge zu reduzieren, sollten nur Nutzdaten übertragen werden und temporäre Datenstrukturen erst zu Laufzeit auf der Grafikkarte erstellt werden.

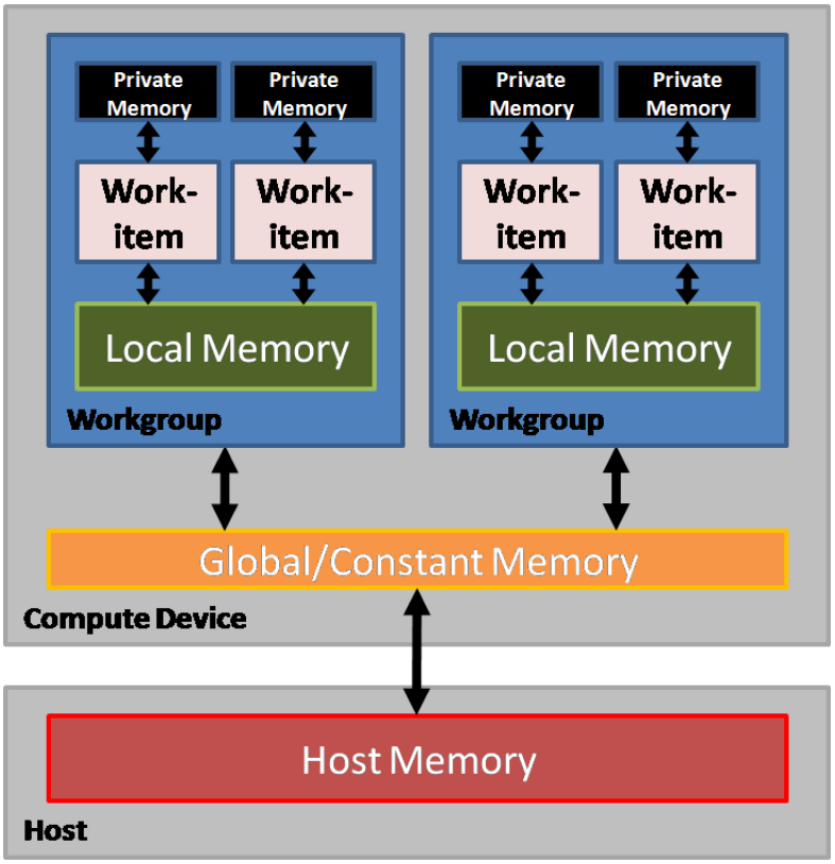

Abbildung 2.2: Allgemeines Speichermodell einer Grafikkarte (Quelle: AMD)

Local Memory anstatt Global Memory nutzen

Der Local Memory ist schneller als der Global Memory und eignet sich daher gut, um die Zugriffe auf den Global Memory zu reduzieren. Er kann sozusagen als Software-Cache genutzt werden.

Häufig gelesene Daten sind auf den Local Memory zu kopieren. Zusätzlich sollten Zwischenergebnisse, die von mehreren Work-items gelesen und geändert werden, in den Local Memory geschrieben und abschließend das Endergebnis zurück in den Global Memory kopiert werden.

Aufteilung auf Work-groups

Durch die Aufteilung der Work-items in Work-groups kann die Grafikkarte am besten ausgelastet werden. Eine implizite Aufteilung findet dann statt, wenn nur die Anzahl der Work-items angegeben ist. Eine explizite Aufteilung ist in der Regel performanter, da nur so der Local Memory effizient genutzt werden kann. Die beste Auslastung wird erreicht, wenn die größte, von der Grafikkarte unterstützte,Work-group size gewählt wird. Die Grafikkarten der Hersteller sind auf verschiedene Größen optimiert. Bei NVIDIA sind dies Größen für ein Vielfaches von 32 und bei AMD ein Vielfaches von 64.

Verzweigung im Programmfluss vermeiden

Alle Programmflussbefehle (`if, switch, for, while`) können die Performance negativ beeinflussen. Eine Grafikkarte besteht aus mehreren SIMD-Einheiten, die unabhängig voneinander verschiedene Instruktionen parallel ausführen können. Bei AMD können 64 und bei NVIDIA 32 Daten pro SIMD-Einheit durch eine Instruktion parallel verarbeitet

werden. Die auszuführenden Work-items werden auf diese SIMD-Einheiten aufgeteilt. Falls nun ein Work-item aus einer Einheit einen anderen Ausführungspfad folgt als die restlichen Work-items, wird der Ablauf serialisiert und die Laufzeit verlängert sich, da nicht zwei unterschiedliche Instruktionen innerhalb einer SIMD-Einheit parallel ausgeführt werden können.

Speziellere Mathematikfunktionen allgemeineren vorziehen

Die Bibliotheken von GPGPU-Technologien bieten eine Vielzahl an optimierten Mathematikfunktionen. Wenn möglich sollte eine spezialisiertere Funktion der allgemeineren vorgezogen werden. So ist zum Beispiel anstatt der allgemeineren Funktion `pow(2,x)` die speziellere Funktion `exp2(x)` zu verwenden.

3 Open Computing Language

Open Computing Language (OpenCL) ist eine offener Standard der Khronos Group, der die einheitliche Nutzung von CPUs, GPUs und anderen Prozessoren spezifiziert. Dazu wird die Programmiersprache OpenCL C mit der Syntax von ISO C99 genutzt. OpenCL C selbst ist eine Untermenge von ISO C99 mit Erweiterungen zur Parallelverarbeitung. Die erste Version wurde im Dezember 2008 von der Khronos Group veröffentlicht. Die aktuelle Version 1.1 ist vom Juni 2010.

Anwendungen die OpenCL nutzen, können ohne Änderungen auf allen Geräten ausgeführt werden. Vorausgesetzt auf dem System ist ein OpenCL-Treiber installiert und das Gerät sowie der Treiber unterstützen die verwendete Version. Einschränkungen der Plattformunabhängigkeit gibt es bei der Verwendung von Erweiterungen die laut Standard nicht unterstützt werden müssen und hardwarespezifischen Optimierungen. Erweiterungen unterscheiden sich in optionale Erweiterungen aus dem Standard und die von Herstellern definierten. Letztere sollten nur genutzt werden, wenn die verwendete Hardware sicher eingeschränkt werden kann.

Viele namhafte Hersteller stellen Implementierungen in Form von Treibern und Programmierplattformen für ihre Geräte bereit, wie zum Beispiel:

- NVIDIA (Grafikkarten) [1]
- AMD (Grafikkarten und x86-Prozessoren mit SSE ab Version 2.x) [2]
- Intel (x86-Prozessoren mit SSE ab Version 4.1) [9]
- IBM (POWER6 und Cell Prozessoren) [10]

Somit ist OpenCL nicht nur auf Grafikkarten beschränkt, kann aber dennoch für GPGPU genutzt werden.

In diesem Kapitel wird das Plattformmodell und das allgemeine Programmiermodell aus Abschnitt 2.2 auf Seite 7 in Bezug auf OpenCL genauer erläutert. Als Beispiel einer OpenCL-Implementierung wird AMDs „Accelerated Parallel Processing" vorgestellt.

3.1 Plattformmodell

Das von OpenCL definierte Plattformmodell ist in Abbildung 3.1 auf der nächsten Seite dargestellt. Es besteht aus dem sogenannten Host, einen Computer an den ein oder mehrere Compute Devices angeschlossen sind. Compute Devices sind Geräte die OpenCL unterstützen, wie zum Beispiel Grafikkarten oder Hauptprozessoren im Verbund mit ihren Hauptspeicher. Diese sind weiter unterteilt in ein oder mehrere Compute Units. Bei einer Grafikkarte sind die Compute Units die Shader und bei einem Multicore-Prozessor die einzelnen Prozessorkerne. Eine Compute Unit enthält wiederrum ein oder mehrere Processing Elements. Diese werden in der Regel als SIMD-Einheiten genutzt.

Aufgrund des offenen Standards und der vielen unterstützten Geräte können Geräte und Implementierungen von unterschiedlichen Herstellern in einem Host verfügbar sein. Diese können mit Hilfe der OpenCL-Programmierschnittstelle abgefragt werden. In diesen Fall tritt zwischen den Host und der Compute Devices noch die Plattform in Form eines Treibers. So können in einem Computer z.B. die Compute Devices in Form eines CPUs von AMD und einer GPU von NVIDIA vorhanden sein. In diesem Fall sind die OpenCL-Plattformen bzw. Treiber von AMD und NVIDIA vorhanden.

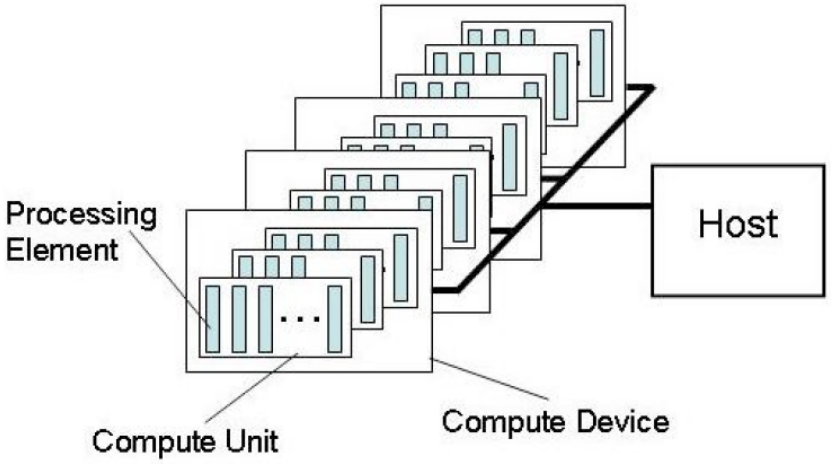

Abbildung 3.1: OpenCL Plattformmodell (Quelle: Khronos Group)

3.2 Programmiermodell

Das Programmiermodell von OpenCL unterscheidet sich kaum von dem in Abschnitt 2.2 auf Seite 7 beschriebenen Modell.

Die Programmierschnittstelle (API) von OpenCL enthält Methoden für die Host-Anwendung und den Kernel (siehe [11]). Die API ist in C implementiert, jedoch bietet die Khronos Group offiziell eine C++ Wrapper API an. Alle Methoden für die Host-Anwendung sind im Namespace `cl` definiert. Darüber können zum Beispiel Eigenschaften über die Devices abgefragt oder Kernel initialisiert und gestartet werden.

OpenCL-Kernel werden in OpenCL C programmiert, dass Methoden, Schlüsselwörter und Datentypen zur Verfügung stellt. Methoden zur Abfrage der jeweiligen IDs, zur Synchronisation und für mathematische Funktionen sind nur einige Beispiele. Folgende Schlüsselwörter können bei der Deklaration von Kernel und Variablen genutzt werden:

__kernel Eine Methode mit dem Prefix `__kernel` kennzeichnet eine Kernel-Methode und kann aus der Host-Anwendung aufgerufen werden.

__global Daten mit dem Prefix `__global` werden im Global Memory gespeichert.

__constant Daten mit dem Prefix `__constant` werden im Constant Memory gespeichert.

__local Daten mit dem Prefix `__local` werden im Local Memory gespeichert. Kernel-Argumente können zwar diesen Prefix benutzen, dürfen dann aber nur den Wert `NULL` übergeben bekommen.

Kernel werden in OpenCL-Kernel und native Kernel unterschieden. OpenCL-Kernel werden erst zur Laufzeit kompiliert. Es ist üblich die Kernel als String in der Host-Anwendung zu definieren oder sie in eine separate Datei mit dem Suffix `.cl` auszulagern. OpenCL-Kernel sind somit bis auf die zuvor beschriebenen Einschränkungen auf allen OpenCL-Implementierungen lauffähig. Native Kernel werden nicht zur Laufzeit kompiliert und sind von der Implementierung abhängig, da die Hersteller unterschiedliche Zwischensprachen verwenden.

Nachfolgend wird anhand einer Vektoraddition der Ablauf einer OpenCL-Anwendung in C++ beschrieben. Der Kernel ist in der Datei `kernel.cl` gespeichert:

```
__kernel void addVec(__global int* vecC, const __global int* vecA, ↩
    const __global int* vecB, const unsigned int size) {
   unsigned int w = get_global_id(0);
   if(w >= size)
      return;
```

```
    vecC[w] = vecA[w] + vecB[w];
}
```

1. Abfrage der vorhandenen OpenCL-Plattformen.

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
```

2. Abfrage aller Plattformen nach Compute Devices vom Typ GPU. Anschließend das erste Device aus der Liste wählen, um auf diesen zu rechnen.

```
std::vector<cl::Device> devices;
std::vector<cl::Device> devTmp;
for (std::vector<cl::Platform>::iterator it = platforms.begin(); ↩
    it != platforms.end(); ++it) {
   it->getDevices(CL_DEVICE_TYPE_GPU, &devTmp);
   devices.insert(devices.end(), devTmp.begin(), devTmp.end());
   devTmp.clear();
}
cl::Device device = devices.front();
```

3. Die Ressourcen zur Ausführung eines Kernels werden in einem `Context` zusammengefasst.

```
cl::Context context = cl::Context(devices);
```

4. Eine `CommandQueue` koordiniert die Ausführung der Kernel.

```
cl::CommandQueue cmdQ = cl::CommandQueue(context, device, ↩
    CL_QUEUE_PROFILING_ENABLE);
```

5. Laden des Programmes aus der Datei `kernel.cl` und Erstellen des Kernels `addVector`.

```
std::string src = readFile("kernel.cl");
cl::Program::Sources source;
source.push_back(std::make_pair(src.data(), src.length()));
cl::Program program = cl::Program(context, source);
try {
   program.build(devices);
} catch(cl::Error& err) {
   return EXIT_FAILURE;
}
cl::Kernel kernel = cl::Kernel(program, "addVector");
```

6. Erstellen der OpenCL-Datenstrukturen und kopieren der Daten aus der Host-Anwendung auf den Grafikkartenspeicher. Neben den atomaren Datentypen unterstützt OpenCL die Datentypen `Buffer`, um 1-dimensionale Felder oder Vektoren zu speichern, und `Image`, um 2- oder 3-dimensionale Texturen, Frame-Buffer oder Bilder zu speichern.

```
cl_int status = CL_SUCCESS;   // Fehlercode
cl::Buffer aBuffer(context, CL_MEM_READ_ONLY | ↩
    CL_MEM_COPY_HOST_PTR, sizeof(int) * vecA.size(), &vecA[0], ↩
    &status);
cl::Buffer bBuffer(context, CL_MEM_READ_ONLY | ↩
    CL_MEM_COPY_HOST_PTR, sizeof(int) * vecB.size(), &vecB[0], ↩
    &status);
cl::Buffer cBuffer(context, CL_MEM_WRITE_ONLY, sizeof(int) * ↩
    vecC.size(), NULL, &status);
```

7. Argumente für den Kernel setzen.

```
status = kernel.setArg(0, cBuffer);
status = kernel.setArg(1, aBuffer);
status = kernel.setArg(2, bBuffer);
status = kernel.setArg(3, vecC.size());
```

8. Kernel zur Ausführung an die `CommandQueue` senden.

```
cl::KernelFunctor func = kernel.bind(cmdQ, ↩
    cl::NDRange(vecC.size()), cl::NDRange(1));
cl::Event event = func();
event.wait();
cmdQ.finish();
```

9. Ergebnis aus dem Grafikspeicher zurück in den Hauptspeicher kopieren.

```
status = cmdQ.enqueueReadBuffer(cBuffer, true, 0, sizeof(int) * ↩
    vecC.size(), &vecC[0]);
```

3.3 AMD Accelerated Parallel Processing

Accelerated Parallel Processing (APP) ist Software Development Kit (SDK) für OpenCL von AMD. In ihr sind ein OpenCL-Treiber für x86-Prozessoren, Header-Dateien für C und C++, Beispielprogramme und Analysewerkzeuge enthalten. Der Intel C Compiler, Microsoft Visual Studio 2008/2010, die GNU Compiler Collection sowie Windows (ab XP) und verschiedene Linux-Distributionen werden bisher unterstützt.

Aufgrund der Unterstützung von CPUs bietet sich APP auch zur OpenCL-Entwicklung ohne eine geeignete Grafikkarte an.

3.3.1 Einrichtung des Software Development Kit

Die folgende Anleitung ist für Windows 7 (64 Bit) und das AMD APP SDK in der Version 2.3. Zum Kompilieren unter Windows werden die Compiler von MinGW[1] zusammen mit dem Tool `make` verwendet.

- Download des AMD APP SDK von `http://developer.amd.com/gpu/AMDAPPSDK/downloads/Pages/default.aspx`
- Entpacken der Installationsdatei durch ausführen der Datei:

```
ati-stream-sdk-v2.3-vista-win7-64.exe
```

- Anschließend sollte automatisch das Installationsprogramm gestartet werden. Wurden die Dateien in das Standardverzeichnis entpackt, kann die Installation manuell durch Ausführen dieser Datei gestartet werden:

```
C:\ATI\SUPPORT\streamsdk_2-3_win764\Setup.exe
```

- Folgende Pakete installieren:
  - ATI Stream SDK v2 Developer
  - ATI Stream SDK v2 Samples
  - ATI Stream Profiler 2.1

[1] `http://www.mingw.org`

  - Stream KernelAnalyzer 1.7

- Setzen der Umgebungsvariablen z.B. mit der Eingabeaufforderung `cmd.exe`

```
setx ATISTREAMSDKROOT "C:\Program Files(x86)\ATI Stream
setx ATISTREAMSDKLIB "%ATISTREAMSDKROOT%\lib\x86
setx ATISTREAMSDKSAMPLESROOT "C:\Users\<username>\Documents\ATI ↩
    Stream
setx PATH "%PATH%;%ATISTREAMSDKROOT%\bin\x86
```

  - `setx` setzt die Umgebungsvariable nur für den angemeldeten Benutzer
  - da MinGW aktuell nur 32 Bit unterstützt, werden die Pfade für 32 Bit verwendet

- Hinzufügen der Pfade für MinGW zur Umgebungsvariable `PATH`[2] z.B. mit der Eingabeaufforderung `cmd.exe`

```
setx PATH "%PATH%;<directory to MinGW>\bin
setx PATH "%PATH%;<directory to MinGW>\msys\1.0\bin
```

- Kontrollieren der Umgebungsvariablen z.B. mit der Eingabeaufforderung `cmd.exe`

```
echo %ATISTREAMSDKROOT%
echo %ATISTREAMSDKLIB%
echo %ATISTREAMSDKSAMPLESROOT%
echo %PATH%
```

- Erstellen einer Makefile z.B.

```
CXX=g++
CXXFLAGS=-O2 -g -Wall -Werror
all : <program>.exe
%.o : %.cpp
   $(CXX) $(CXXFLAGS) -c -I"$(ATISTREAMSDKROOT)\include" $< -o $@

%.exe : %.o
   $(CXX) $(CXXFLAGS) $< -L"$(ATISTREAMSDKLIB)" -lOpenCL -o $@

clean:
   $(RM) *.o *.so *.a <program>.exe
```

- Bei Verwendung einer Entwicklungsumgebung müssen folgende Optionen konfiguriert werden:
  - Include-Pfad: `$(ATISTREAMSDKROOT)\include`
  - Pfad für zusätzliche Bibliotheken: `$(ATISTREAMSDKLIB)`
  - zusätzliche Bibliothek: `OpenCL` oder `OpenCL.lib`

3.3.2 AMD APP Profiler

Der AMD APP Profiler ist ein Laufzeit-Profiler für Microsoft Visual Studio (ab 2008) und die Kommandozeile. Mit ihm können Laufzeiten von OpenCL und DirectCompute Kernel, Datentransfers und mehr gemessen werden. In Abbildung 3.2 auf der nächsten Seite ist der integrierte Profiler in Visual Studio 2010 zu sehen.

Vor der Benutzung in Visual Studio ist unter *Hilfe* > *Info über Microsoft Visual Studio* zu prüfen, ob der Profiler verfügbar ist. Nach erfolgreicher Installation kann unter *Ansicht* > *Weitere Fenster* das *Stream Session List* Fenster geöffnet werden, über dies das Profiling gestarten werden kann.

[2] GNU `make` für Windows stürzt ab, wenn in der Umgebungsvariable `PATH` Leer- & Sonderzeichen wie Klammern vorkommen. Ein Workaround ist im Anhang beschrieben.

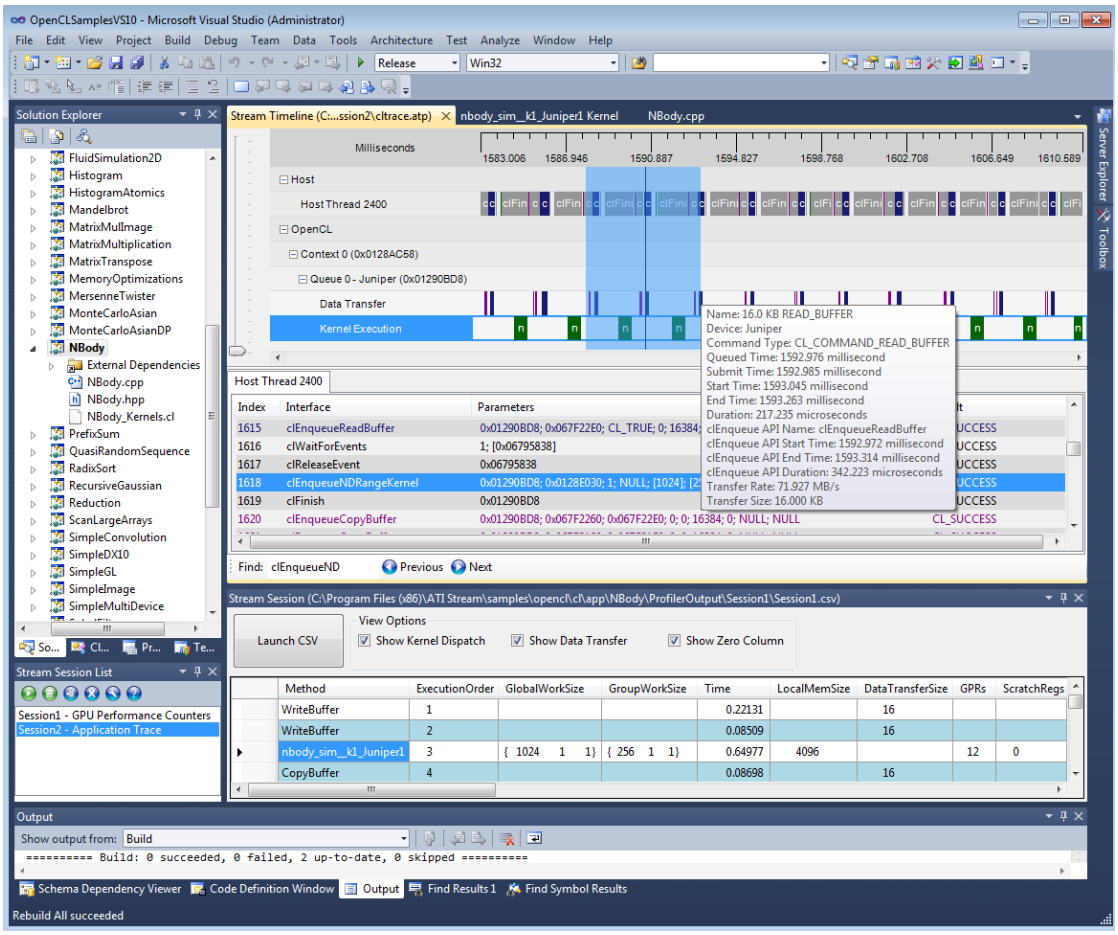

Abbildung 3.2: AMD APP Profiler in Visual Studio 2010 (Quelle: AMD)

3.3.3 AMD APP KernelAnalyzer

Der AMD APP KernelAnalyzer ermöglicht das Kompilieren und Analysieren eines Kernels. Für eine große Auswahl von AMD Grafikkarten stellt er Abschätzungen für das Verhältnis von ALU- zu Fetch-Operationen und einen möglichen Flaschenhals an. Der Compiler vom KernelAnalyzer ermöglicht es, einen Kernel unabhängig von der Host-Anwendung zu kompilieren. Dadurch eignet er sich sehr gut zur Entwicklung mit OpenCL C.

In Abbildung 3.3 auf der nächsten Seite ist der KernelAnalyzer dargestellt. Auf der linken Seite des Programms ist der geladene Kernel zu sehen. Im unteren Teil werden die Compiler-Statistiken und Compiler-Meldungen dargestellt. Zusätzlich kann auf der rechten Seite der Assembler-Code von AMD Grafikkarten angezeigt werden.

3.4 Debugging

Das Debuggen unterteilt sich in die Host-Anwendung und den Kernel. Da eine Host-Anwendung in C oder C++ geschrieben ist und auf dem Hauptprozessor ausgeführt wird, kann das Debugging wie bei jedem anderen Programm durchgeführt werden. Das Debuggen eines Kernels ist im Standard nicht definiert. Dennoch bieten einige Implementierungen Möglichkeiten an, um zumindest Kernel auf einem CPU zu debuggen.

AMD und Intel stellen über die Erweiterung `cl_amd_printf` bzw. `cl_intel_printf` die Systemfunktion `printf()` zur Laufzeit eines Kernels zur Verfügung. Diese kann zum Beispiel so genutzt werden:

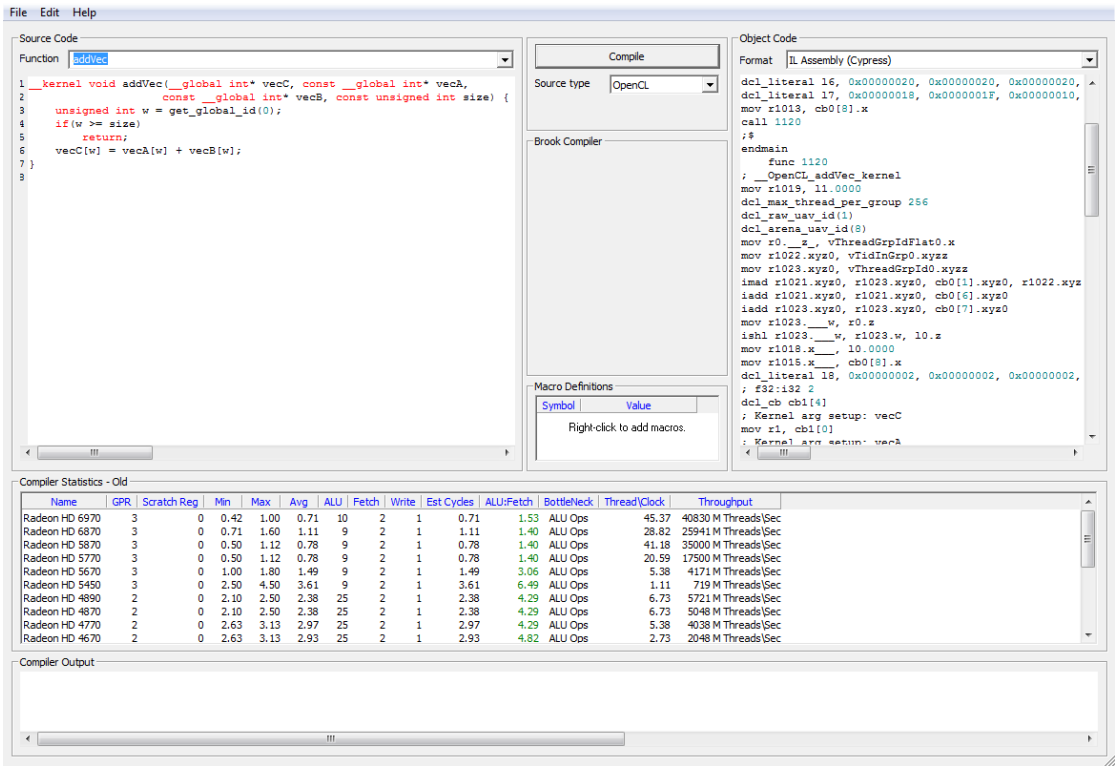

Abbildung 3.3: AMD APP KernelAnalyzer (Quelle: AMD)

```
#ifdef DEBUG_ON
    #ifdef cl_amd_printf
        #pragma OPENCL EXTENSION cl_amd_printf : enable
        #define DEBUG(output) printf(output);
    #else
        #define DEBUG(output) /* no debug */
    #endif
#else
    #define DEBUG(output) /* no debug */
#endif

__kernel void addVec(__global int* vecC, const __global int* vecA, ↩
     const __global int* vecB, const unsigned int size) {
    DEBUG("addVec called\n")
    unsigned int w = get_global_id(0);
    if(w >= size)
        return;
    vecC[w] = vecA[w] + vecB[w];
}
```

Die Host-Anwendung übergibt zur Laufzeit eine Compiler-Option, um die Ausgabe einzuschalten:

```
program.build(devices, "-DDEBUG_ON");
```

Außerdem unterstützt AMD APP unter Linux das Debugging mit dem GNU Debugger (GDB) auf CPUs[3]. Wie beim Kompilieren der Host-Anwendung mit GCC, muss auch die Compiler-Option `-g` zur Laufzeit in der Host-Anwendung übergeben werden:

```
program.build(devices, "-g");
```

[3] Unter Windows kommt es zu einem Segmentation Fault. Jedoch wäre das Debuggen prinzipiell wie beschrieben möglich.

Um den Kernel nur auf dem Hauptprozessor auszuführen, dürfen nur Devices vom Typ `CL_DEVICE_TYPE_CPU` ausgewählt werden. Zusätzlich darf nur ein Prozessorkern genutzt werden, was mit einer Umgebungsvariable eingestellt werden kann:

```
CPU_MAX_COMPUTE_UNITS=1
```

Unter Berücksichtigung dieser Einstellungen kann GDB wie folgt genutzt werden:

- Start von GDB:

  ```
  gdb <program>
  ```

- Starten der Anwendung:

  ```
  run [arguments]
  ```

- Breakpoint in der Host-Anwendung setzen:

  ```
  b <linenumber | functionname>
  ```

- Kernel-Funktionen sind erst sichtbar, nachdem der Kernel geladen wurde:

  ```
  program.build(devices, "-g");
  ```

- Breakpoints für Kernel-Funktionen werden durch einen Prefix und Suffix umklammert

  ```
  b __OpenCL_<kernelfunction>_kernel
  ```

  - als Beispiel wird der Breakpoint für `addVec` gesetzt:

    ```
    b __OpenCL_addvec_kernel
    ```

- Auflisten aller Kernel-Funktionen:

  ```
  info functions __OpenCL
  ```

- Schrittweises Durchgehen: `s`
- Fortfahren der Anwendung: `c`
- GDB beenden: `quit`

3.5 Zusammenfassung

Aktuelle Version	1.1 (Juni 2010)
Plattformabhängigkeit	keine, wenn OpenCL-Treiber vorhanden (CPU, GPU u.m.)
Herstellerunterstützung	NVIDIA, AMD, Intel, IBM
Programmiersprache der Kernel	OpenCL C
Programmiersprache des Host	C und C++
Unterstützung der C++ Standard Library	nur in der Host-Anwendung
Debugging	eingeschränkt
Einschränkungen (Kernel)	• keine Pointer auf Funktionen • keine Rekursion • keine dynamische Speicherallokierung • u.m. (siehe [11])
Einbettung in Host-Anwendung	• Kernel als OpenCL-Kernel oder native Kernel • als String oder externe Datei • Initialisierung der OpenCL-Klassen und Kernel • Start des Kernels
Programmablauf	• Abfrage der vorhandenen OpenCL-Plattformen • Selektion der Compute Devices • Erstellen eines Context und einer CommandQueue • Laden und Initialisierung des Kernel • Initialisierung der Datenstrukturen und Datentransfer auf Device • Setzen der Kernel-Argumente • Kernel starten • Export der Daten
Datentransfer auf Device	• `cl::Buffer` für Vektortypen • `cl::Image2D`, `cl::Image3D` für Bilder und Texturen

4 Laufzeitmessung

Anhand einer Matrixmultiplikation wird die Laufzeit der Implementierungen für die GPU in OpenCL und die CPU in C++ verglichen. Die Matrixmultiplikationen werden mit den Datentypen float und double getestet. Die Grafikkarte des Testsystems unterstützt nicht die offizielle Erweiterung `cl_khr_fp64` des OpenCL-Standards für doppelt genaue Gleitkommazahlen. Stattdessen wird vom Hersteller die Erweiterung `cl_amd_fp64` bereitgestellt [12]. Diese implementiert eine Teilmenge der offiziellen Erweiterung und wird im OpenCL-Kernel verwendet.

Die Laufzeit beinhaltet die Dauer für das Kopieren der Daten auf und von dem Grafikkartenspeicher. Das Initialisieren der OpenCL-Hardware sowie das Einlesen und Kompilieren des OpenCL-Kernel wird nicht gemessen, da dies bei einer Anwendung nur einmalig ausgeführt werden müsste. Auf dem Testsystem benötigte dies ungefähr 400ms.

Als Testsystem dient ein PC mit den folgenden Leistungsdaten:

CPU AMD Phenom II X4 955 (4x 3,2 GHz)

RAM 4 GB DDR3 (1333 MHz)

Betriebssystem Windows 7 Professional 64 Bit

GPU ATI Radeon HD 5850 mit 1 GB GDDR5

Treiber AMD Catalyst 11.3

Compiler GNU Compiler Collection 4.5.0

4.1 Quellcode

Für den Datentyp float werden kurz die Implementierungen aufgelistet.

OpenCL-Kernel

```
__constant unsigned int X_DIM = 0;
__constant unsigned int Y_DIM = 1;

__kernel void matMulSimple2D(__global float* matC,
      const __global float* matA, const unsigned int rowA, const ↩
          unsigned int colA,
      const __global float* matB, const unsigned int rowB, const ↩
          unsigned int colB)
{
   const unsigned int W = get_global_id(X_DIM) * get_global_id(Y_DIM);

   if(W >= rowA * colB)
      return;

   const unsigned int ROW = get_global_id(Y_DIM);
   const unsigned int COL = get_global_id(X_DIM);

   float sum = 0;

   for (size_t i = 0; i < colA; ++i)
```

```
    {
      sum += matA[ROW * colA + i] * matB[i * colB + COL];
   }

   matC[ROW * colB + COL] = sum;
}
```

C++

```
double
matMul(MatrixFloat& result, MatrixFloat& left, MatrixFloat& right)
{
   timeUtils::Clock timer;

   float tmp;
   const float* rowLeft;
   float* rowResult;

   timer.start();

   for (size_t r = 0; r < left.rows; ++r)
   {
      rowResult = &result.elements[r * right.cols];
      for (size_t c = 0; c < right.cols; ++c)
        {
         tmp = 0;
         rowLeft = &left.elements[r * left.cols];
         for (size_t j = 0; j < left.cols; ++j)
         {
            tmp += rowLeft[j] * right.elements[j * right.cols + c];
         }
         rowResult[c] = tmp;
        }
    }

   timer.stop();

   return timer.getTimeInSeconds();
}
```

4.2 Auswertung

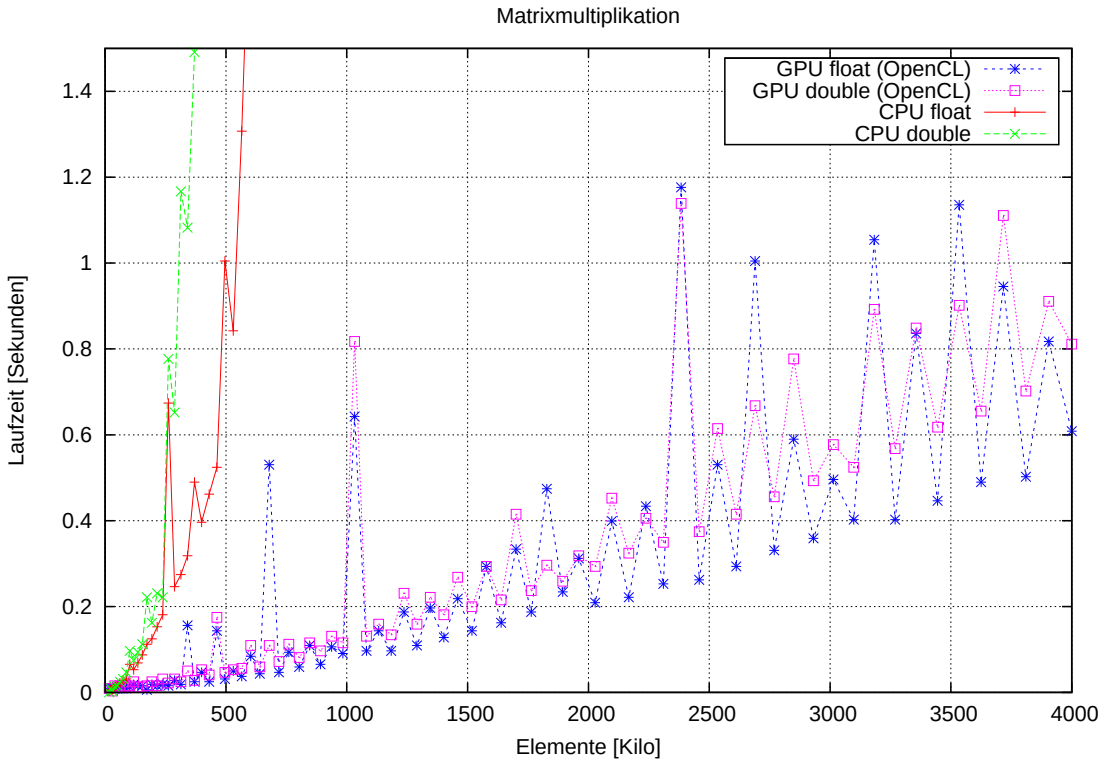

Abbildung 4.1: Laufzeiten der Matrixmultiplikation für die Datentypen float und double.

In Abbildung 4.1 sind die Laufzeiten der Matrixmultiplikation für die Datentypen float und double dargestellt. Diese wurden in C++ für die CPU und in OpenCL für die GPU implementiert. Die Laufzeiten der GPU sind deutlich niedriger als die der CPU. Bei den OpenCL-Implementierungen sind Schwankungen in der Laufzeit zu erkennen, welche mit steigenden Elementen zunehmen. Da die Aufteilung in Work-groups von der Anzahl der Elemente abhängig ist, wird die Grafikkarte unterschiedlich stark ausgelastet und die Laufzeiten schwanken. Bei einfachen Datenstrukturen und Operationen kann die Problemgröße so erweitert werden, dass eine optimale Aufteilung auf die Work-groups gewährleistet ist. Dabei müssen jedoch neutrale Daten hinzugefügt werden, um das Ergebnis nicht zu beeinflussen.

Es ist zuerkennen, dass die Graphen der Datentypen float und double zueinander leicht verschoben sind. So sind im Mittel die Laufzeiten für float geringer als für double.

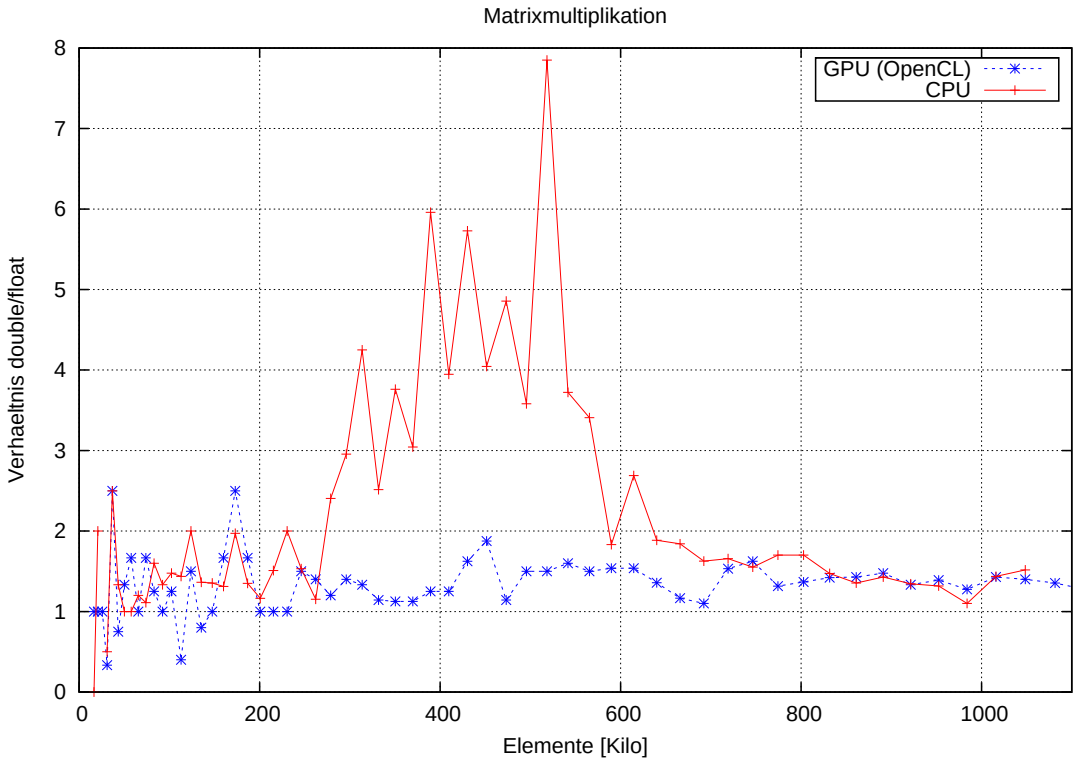

Abbildung 4.2: Verhältnis der Laufzeit für die Matrixmultiplikation von double zu float.

In Abbildung 4.2 ist das Verhältnis von double zu float genauer dargestellt. Der hohe Faktor der CPU zwischen 200.000 und 600.000 Elementen ist vermutlich in Caching-Algorithmen und Cache-Größen begründet. Bei detailierter Betrachtung der Laufzeiten ist zu erkennen, dass die Graphen von float und double auf dem CPU bis ca. 250.000 Elementen den selben Anstieg haben und somit der Faktor nahezu konstant ist. Ab 250.000 Elementen erhöht sich der Anstieg des Graphen für double, wohin gegen der Anstieg von float bis ca. 550.000 Elementen konstant bleibt. Ab 550.000 Elementen erhöht sich auch der Anstieg des Graphen für float und somit gleichen sich beide Graphen wieder an. Dies ist auch im Diagramm 4.2 gut zuerkennen.

Auf der GPU ist die Matrixmultiplikation mit dem Datentyp double im Durchschnitt um den Faktor 1,2 mal langsamer als mit dem Datentyp float. Der Datentyp float ist 4 Byte und double 8 Byte groß. Somit müssen beim Kopieren der Daten auf und aus dem Grafikkartenspeicher doppelt soviele Daten übertragen werden. Zusätzlich ist die Berechnung von 32 Bit Daten (float) laut Herstellerangaben 5 mal schneller als die von 64 Bit Daten (double) [13].

Trotz dieser doppelten Datenmenge und der langsameren Performance ist der Faktor von rund 1,2 sehr gering. Bei quadratischen Matrizen mit m Zeilen und m Spalten beträgt die asymptotische Laufzeit $\mathcal{O}(m) = \frac{1}{p} \cdot m^3$, wobei die Anzahl der maximal möglichen parallel auszuführenden Kernel durch p definiert ist. Da das Kopieren der Daten linear ist, wirkt sich dies nur sehr gering auf die kubische Laufzeit aus. Auch die Operationen vom Datentyp double, beeinflussen die Laufzeit kaum, da ein Kernel von den rund $7 \cdot m$ arithmetischen Operationen für eine Matrixmultiplikation nur $2 \cdot m$ Operationen vom Datentyp double ausführt. Somit beträgt der Rechenanteil für den langsameren Datentyp double weniger als 30 Prozent. Diese Tatsachen erklären den nur geringen Performanceunterschied zwischen double und float bei der Matrixmultiplikation.

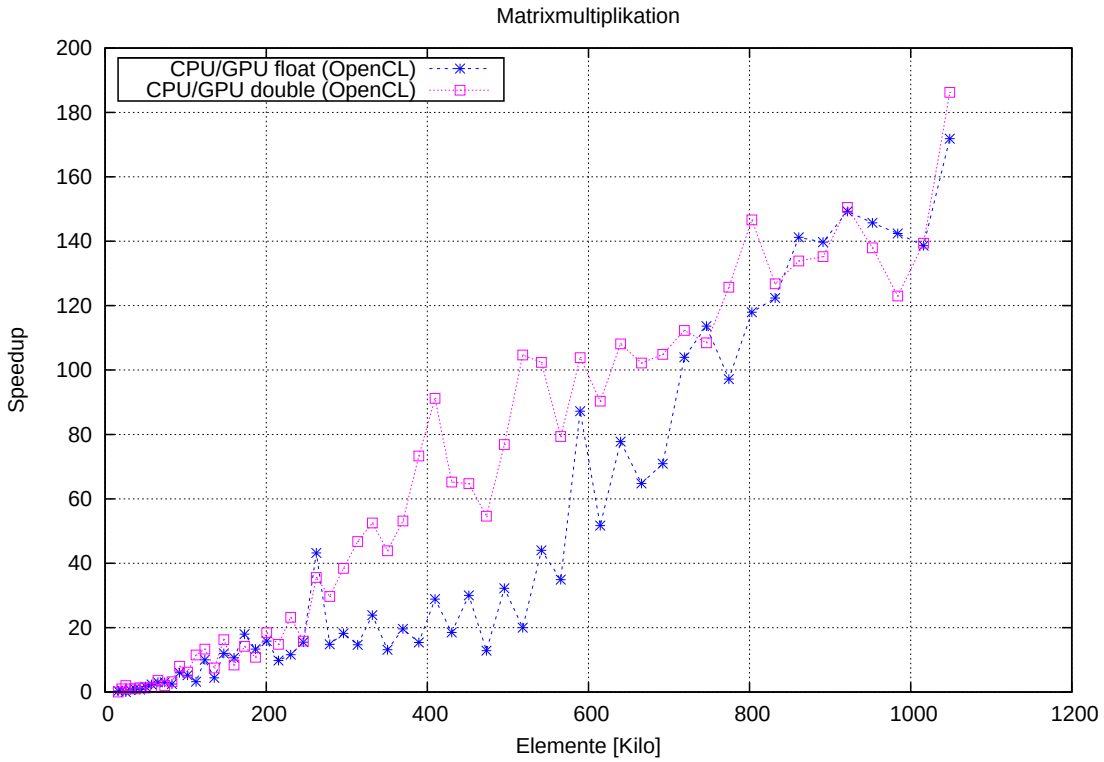

Abbildung 4.3: Speedup der Matrixmultiplikation von float und double.

Der relative Speedup einer GPU im Bezug auf eine CPU ist in Abbildung 4.3 dargestellt. Aufgrund der sehr langen Laufzeiten auf der CPU ist der Speedup nur bis zu Matrizen einer Größe von 1024×1024 berechnet.

Mit steigender Größe der Matrizen steigt auch der Speedup bis auf 180 an, da die Grafikkarte besser ausgelastet wird und der Rechenaufwand in Bezug zum Kommuniktationsaufwand stark ansteigt. Auch in diesem Diagramm sind die Differenzen zwischen 250.000 und 550.000 Elementen erkennbar. Aufgrund der geringen Laufzeitunterschiede zwischen float und double auf der CPU sowie GPU ist der Speedup beider Datentypen ähnlich.

Obwohl die Grafikkarte 1440 Stream-Prozessoreinheiten besitzt [13], beträgt der maximale Speedup auf dem vorhandenen Testsystem nicht 1440. Da die GPU mit 725 MHz und die CPU mit 3,2 GHz unterschiedliche Taktraten besitzen, verringert sich der hypothetische Speedup von 1440, unter Berücksichtigung des Systemtakts als einziges Leistungskriterium, um den Faktor 4,4 auf 326,3. Dieser Wert ist selbst mit stark optimierten Algorithmen schwer erreichbar, da einerseits alle Daten immer auf und von den Grafikkartenspeicher kopiert werden müssen und sich andererseits nicht alle Optimierungen aus Abschnitt 2.4 auf Seite 8 für alle Algorithmen umsetzen lassen. Des Weiteren kommen hardwareunabhängige Einschränkungen hinzu, wie z.B. der Anteil sequenzieller und paralleler Operationen.

5 Fazit

5.1 Zusammenfassung

Das Kapitel 2 beschreibt die allgemeinen Grundlagen des GPGPU, die in ähnlicher Form bei allen Technologien vorzufinden sind. GPGPU ist der Name von Technologien die es ermöglichen allgemeine Berechnungen, die bisher nur auf einer CPU ausgeführt werden konnten, auf Grafikprozessoren auszuführen. Die drei zur Zeit verbreitesten Technologien sind: CUDA, OpenCL und DirectCompute.

Soll eine Anwendung Operationen auf der Grafikkarte berechnen, unterteilt sich die Anwendung in den Host- und den Kernel-Teil. Der Host-Code wird auf der CPU ausgeführt und unterliegt keinen Einschränkungen. Dieser organisiert die Verteilung der Daten und die Zuweisung der Kernel auf die Grafikkarten. Der Algorithmus für die Grafikkarte wird als Kernel bezeichnet und wird in einer speziellen Programmiersprache implementiert. Aufgrund der speziellen Hardwarearchitektur der Grafikkarten unterliegt dieser Code einigen Einschränken, welche sich von Technologie zu Technologie leicht unterscheiden. Aus jedem Kernel wird eine Kernel-Instanz erstellt, welche als Work-item bezeichnet wird. Mehrere Work-items können in Gruppen organisiert werden und werden automatisch den physischen Ausführungseinheiten zugewiesen.

Die spezielle Speicherhierarchie einer Grafikkarte beeinflusst die Art und Weise der Programmierung einer GPU im Vergleich zu einer CPU am stärksten. Der größte, aber zugleich langsamste, Speicherbereich einer Grafikkarte ist der Global memory. In diesem Bereich kann die Host-Anwendung die Daten auf der Grafikkarte lesen und schreiben. Hierarchisch betrachtet, befindet sich unter dem Global memory der Local memory. Dieser ist kleiner, aber schneller, als der Global memory. Der Local memory unterteilt sich in mehrere Speicherbereiche, die nur von einer Work-group und deren Work-items genutzt werden können. Der unterste und zugleich schnellste, aber kleinste, Speicherbereich ist der Private memory. Jedes Work-item hat seinen eigenen Private memory und exklusiven Zugriff auf diesen. Die effiziente Ausnutzung dieser Speicherhierarchie bietet großes Potenzial für Performanceoptimierungen.

Als einer der drei GPGPU-Technologien wird in Kapitel 3 OpenCL vorgestellt. OpenCL ist ein offener Standard der Khronos Group und wurde ursprünglich von Apple entwickelt. Mit OpenCL können verschiedene Prozessorarten, wie CPUs und GPUs, mit einer einheitlichen Programmiersprache programmiert werden. Hierzu definiert der Standard eine API in C für den Host-Teil einer Anwendung und die Programmiersprache OpenCL C für die Programmierung der Prozessoren, dem Kernel-Teil. OpenCL C ist eine Untermenge von ISO C99 mit Erweiterungen zur Parallelverarbeitung.

Um OpenCL-fähige Hardware nutzen zu können, wird ein OpenCL-Treiber, der den offenen Standard implementiert, benötigt. Diese müssen von den Hersteller bereitgestellt werden. Zur Zeit bieten folgende Hersteller für einige ihrer Hardware OpenCL-Treiber an: NVIDIA, AMD, Intel und IBM.

Für die Entwicklung von OpenCL-Anwendungen stellen einige Hersteller SDKs bereit. Diese beinhalten Beispielcode, Analysewerkzeuge, Compiler, Dokumentationen und mehr. Als Beispiel wurde AMD APP in wenigen Worten vorgestellt. Der AMD APP Profiler ist ein Laufzeit-Profiler für Visual Studio und die Kommandozeile. Mit ihm können Laufzeiten von OpenCL und DirectCompute Kernel, Datentransfers und mehr gemessen werden. Der AMD APP KernelAnalyzer ermöglicht das Kompilieren und Analysieren eines Kernels. Für eine große Auswahl von AMD Grafikkkarten stellt er Abschätzungen

für das Verhältnis von ALU- zu Fetch-Operationen und einen möglichen Flaschenhals an. Zusätzlich beinhaltet das SDK einen OpenCL-Treiber für x86-Prozessoren mit Hilfe dessen das Debug-Werkzeug GDB für OpenCL-Kernel (auf dem CPU) genutzt werden kann.

Das Debugging einer OpenCL-Anwendung unterteilt sich in die Host-Anwendung und den Kernel. Die Host-Anwendung unterliegt beim Debugging keinen Einschränkungen. Das Debugging eines OpenCL-Kernels ist bisher nur möglich, wenn dieser auf einer CPU ausgeführt wird. Dazu bieten einige Hersteller Erweiterungen an, mit denen die Standardausgabe genutzt werden kann. Außerdem ist es mit manchen OpenCL-Treibern möglich, Werkzeuge wie GDB zu nutzen.

In Kapitel 4 wird am Beispiel einer Matrixmultiplikation der Performancegewinn einer OpenCL-Anwendung unter Nutzung der Grafikkarte untersucht. Auf dem Testsystem ist die OpenCL-Implementierung bei rund 1 Million Elementen um den Faktor 180 schneller als die sequenzielle Implementierung für die CPU. Des Weiteren ist zuerkennen, dass die Laufzeiten auf der GPU schwanken. Eine optimale Auslastung wird erreicht, wenn die Anzahl der Work-items ein Vielfaches der physischen Ausführungseinheiten ist. Da dies jedoch nicht für jede Matrixgröße zutrifft, schwanken die Laufzeiten und somit auch der Speedup.

Aus diesem Beispiel kann jedoch nicht allgemein abgeleitet werden, dass jeder Algorithmus auf einer Grafikkarte bedeutend schneller als sein CPU-basiertes Pendant ist. Die Matrixmultiplikation zählt mit einer asymptotischen Laufzeit von $\mathcal{O}(m) = m^3$ zu den rechenintensiven Algorithmen und besitzt mit 2-dimensionalen Feldern eine einfache Datenstruktur. Im Vergleich zu dem Rechenaufwand ist der Aufwand für das Kopieren der Daten auf und von dem Grafikkartenspeicher sehr klein. Vor allem bei Problemen mit linearer Zeitkomplexität ist der Kopieraufwand oft größer als der eigentliche Rechenaufwand und die Portierung auf eine Grafikkarte lohnt sich somit nicht.

Bei geeigneter Auswahl der Algorithmen kann durch GPGPU ein hoher Performancegewinn erzielt werden. Die Programmierung von GPU-basierten Anwendungen ist im Vergleich jedoch aufwendiger als die für CPU-basierte Anwendungen.

Um eine optimale Performance zu erreichen, müssen die Speicherhierarchie, der Kopieraufwand und das Abfragen der OpenCL-API berücksichtigt werden. Zusätzlich muss damit gerechnet werden, dass der Grafikkartenspeicher nicht ausreicht. Hierfür müssen für jeden Algorithmus Lösungen erarbeitet werden, wie die Daten in Blöcke unterteilt, diese verarbeitet und anschließend wieder zu einem Endergebnis zusammenführt werden können. Besonders wenn die Datengröße unbekannt ist oder die Hardwareanforderungen für einen Endanwender nicht zu hoch sein sollen, sind Lösungen für die Datenaufteilung erforderlich. Des Weiteren muss berücksichtigt werden, dass viele OpenCL-fähige Grafikkarten noch nicht die Erweiterung `cl_khr_fp64` für doppelte genaue Gleitkommazahlen unterstützen.

5.2 Ausblick

Die Entwicklungen im Bereich von GPGPU zeigen, dass großes Interesse an der Technologie besteht. So verkündete Amazon im November 2010 [14], dass es GPU-Instanzen auf der eigenen Cloud-Plattform EC2 bereitstellt. Im Gegensatz zu den normalen Computern des EC2-Services enthalten die gemieteten Computer der GPU-Instanz NVIDIA Tesla Grafikkarten. Diese Grafikkartenmodelle von NVIDIA wurden speziell für das GPGPU entwickelt.

Mit der Tesla-Serie bietet bisher nur NVIDIA spezielle Grafikkarten zur Parallelverarbeitung an. Vermutlich ist deshalb NVIDIAs CUDA im High Performance Computing am stärksten verbreitet. Da CUDA nur auf Grafikkarten von NVIDIA genutzt werden kann, ist es für Consumer-Software weniger geeignet. Dennoch wird auch hier überwie-

gend CUDA genutzt. Die aktuelle Entwicklung der CPUs zeigt, dass diese in Zukunft immer mehr Prozessorkerne besitzen werden. Aufgrund der Unterstützung von Multicore-Prozessoren könnte OpenCL in Zukunft an Popularität gewinnen.

Um Multicore-Prozessoren einfach und effizient nutzen zu können, sind neue Programmiersprachen erforderlich. Die gegenwärtige Programmierung mit Threads ist für Probleme mit Datenparallelität nur bedingt geeignet, da das Erstellen und Verwalten von Threads durch das Betriebssystem mit hohen Aufwand verbunden ist. Dies ist erst bei langlebigen Threads, wie sie bei der Task-Parallelität auftreten, effizient. Mit sogenannten Thread Pools wird versucht den hohen Kosten der Threads entgegenzuwirken. Wenn Prozessorhersteller Treiber für ihre Hardware bereitstellen, könnte dieser Missstand durch OpenCL gelöst werden. Die Veröffentlichungen von IBM und Intel zeigen, dass dies durchaus denkbar ist.

Aufgrund des offenen Standards empfiehlt der Autor der fokus GmbH OpenCL als GPGPU-Technologie zuwählen. In Ingenieurbüros werden oft spezielle Grafikkarten für CAD-Anwendungen[1] oder für Multi-Head-Lösungen[2] verwendet. Da die Hardware- und Treiberunterstützung bei den Hersteller zunimmt, können viele Kunden von der neuen Technologie profitieren, ohne auf ihre Spezialhardware verzichten zu müssen.

Bei der Integration von OpenCL in die Software der fokus GmbH sollten Abstraktionsschichten entwickelt werden, die zum Beispiel bereits abgefragte OpenCL-Hardware und Kernel zwischenspeichern und die Datenaufteilung organisieren. Die Wahl der Implementierung für die CPU und die GPU kann durch das Entwurfmuster Strategie vereinfacht werden. Durch geeignete Abstraktionen und Entwurfsmuster kann die Komplexität der GPU-Programmierung vor Entwicklern aus anderen Bereichen der Software versteckt werden.

Die Entwicklung von OpenCL-Kernel sollte schrittweise von einer einfachen, verständlichen und oft langsamen Implementierung hin zu einer optimierten Implementierung erfolgen. Da Performanceoptimierungen ein großes Fehlerpotenzial besitzen, sollten zu Beginn für jeden Algorithmus Testfälle entwurfen werden.

[1] Zum Beispiel die NVIDIA Quadro- oder AMD FirePro-Serie.
[2] Zum Beispiel die Matrox M-Serie.

Literaturverzeichnis

[1] NVIDIA Corporation. NVIDIA GPU Computing Developer Home Page. URL: http://developer.nvidia.com/object/gpucomputing.html. Abgerufen: 15.03.2011.

[2] Advanced Micro Devices Inc. OpenCL Zone. URL: http://developer.amd.com/zones/OpenCLZone/. Abgerufen: 15.03.2011.

[3] Khronos Group. OpenCL Overview. URL: http://www.khronos.org/opencl/. Abgerufen: 15.03.2011.

[4] Wikipedia. Microsoft Direct3D. URL: http://en.wikipedia.org/wiki/Direct3d. Abgerufen: 16.03.2011.

[5] NVIDIA Corporation. Quadro 6000. URL: http://www.nvidia.de/object/product-quadro-6000-de.html. Abgerufen: 16.03.2011.

[6] Simon Hülsbömer. Der x86-Prozessor wird 30 - wie Intel dank IBM alle Gipfel stürmte. *Computerwoche*, Juni 2008. URL: http://www.computerwoche.de/hardware/notebook-pc/1866928/index8.html. Abgerufen: 16.03.2011.

[7] Intel Corporation. Intel Xeon 7000er-Prozessoren. URL: http://www.intel.com/cd/products/services/emea/deu/processors/xeon7000/overview/344515.htm. Abgerufen: 16.03.2011.

[8] NVIDIA Corporation. *CUDA C Best Practices Guide*, 3.2 edition, August 2010. URL: http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide.pdf. Abgerufen: 21.03.2011.

[9] Intel Corporation. Intel OpenCL SDK. URL: http://software.intel.com/en-us/articles/intel-opencl-sdk/. Abgerufen: 16.03.2011.

[10] International Business Machines Corp. OpenCL Development Kit for Linux on Power. URL: http://www.alphaworks.ibm.com/tech/opencl, Oktober 2009. Abgerufen: 16.03.2011.

[11] Aaftab Munshi. *The OpenCL Specification - Version: 1.1*. Khronos OpenCL Working Group, September 2010.

[12] Benedict Gaster, Michael Houston, Brian Sumner, Micah Villmow, and Bixia Zheng. OpenCL Extension cl_amd_fp64. URL: http://www.khronos.org/registry/cl/extensions/amd/cl_amd_fp64.txt. Abgerufen: 16.03.2011.

[13] Advanced Micro Devices Inc. ATI Radeon HD 5850 Grafikkarten. URL: http://www.amd.com/de/products/desktop/graphics/ati-radeon-hd-5000/hd-5850/Pages/ati-radeon-hd-5850-overview.aspx. Abgerufen: 16.03.2011.

[14] Amazon Corp. Announcing cluster gpu instances for amazon ec2. URL: http://aws.amazon.com/de/about-aws/whats-new/2010/11/15/announcing-cluster-gpu-instances-for-amazon-ec2//190-1231653-1355408. Abgerufen: 16.03.2011.

Abkürzungsverzeichnis

API	Programmierschnittstelle (englisch: Application Programming Interface)
APP	AMD Accelerated Parallel Processing
CAD	Computer-aided Design
CPU	Hauptprozessor (englisch: Central Processing Unit)
CUDA	Compute Unified Device Architecture
EC2	Elastic Compute Cloud
GDB	GNU Debugger
GPGPU	General Purpose Computation on Graphics Processing Unit
GPU	Grafikprozessor (englisch: Graphics Processing Unit)
OpenCL	Open Computing Language
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data

Abbildungsverzeichnis

Workaround für make-Fehler

- Problem: GNU `make` für Windows stürzt ab, wenn die Umgebungsvariable `PATH` Leer- und Sonderzeichen wie Klammern enthält.
- Lösung:
  - Verschieben der Programmdateien in einen Ordner ohne Leer- und Sonderzeichen wie Klammern, oder
  - Relevante Pfade im „8-Punkt-3"-Format angeben
  - Shells wie die Eingabeaufforderung und die PowerShell laden alle Umgebungsvariablen für eine Sitzung beim Start und aktualisieren diese nicht, somit können zwei Shells mit unterschiedlichen Umgebungsvariablen genutzt werden
- Vorgehen:
  1. Umgebungsvariable `PATH` zwischenspeichern
  2. Relevante Pfade im „8-Punkt-3"-Format angeben, z.B. mit `dir /X` ermitteln, oder `make` verschieben und Umgebungsvariable `PATH` ändern, z.B:
     `C:\PROGRA~2\MinGW\bin;C:\PROGRA~2\MinGW\msys\1.0\bin;%SystemRoot%;%SystemRoot%\system32`
  3. Shell zum Kompilieren starten und offen lassen
  4. Zwischengespeicherte Umgebungsvariable `PATH` wiederherstellen
  5. Shell zum Ausführen und Testen starten