JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

GPGPU

Ausarbeitung

Systemtechnik
5BHIT 2015/16

René Hollander
Paul Kalauner

Inhaltsverzeichnis

1 Einführung

1.1 Grafikkarte

Grafikkarten wurden hauptsächlich für die Beschleunigung von 2D und 3D Anwendungen entwickelt. Die Grafikkarte wird verwendet um eine Szene in ein Bild umzuwandeln. Dabei werden von der CPU Vertexdaten generiert (Punkte im dreidimensionalen Raum) und mithilfe der GPU Transformiert, mit einer Textur überzogen und zuletzt Rasterisiert um am Bildschirm angezeigt zu werden. Grafikkarten wurden auf solche Berechnungen spezifiziert und sind dafür extrem optimiert worden.

Blockschaltdiagramm GPU

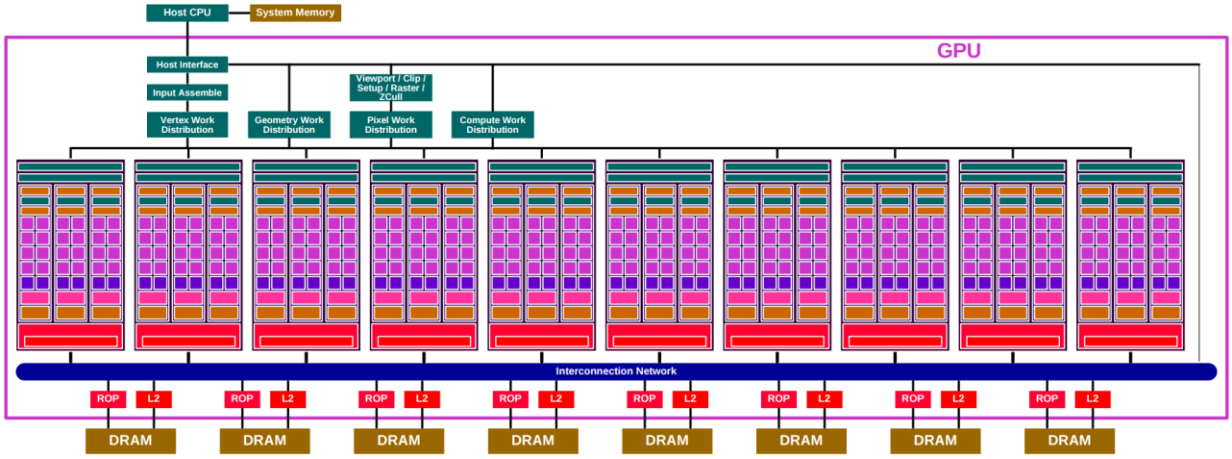

Abbildung 1: Blockschaltdiagramm einer Nvidia GT200 GPU [25]

Eine GPU besteht aus mehreren Streaming Multiprocessors. Jeder dieser Streaming Multiprocessors bekommt eigene Instruktionen und verarbeitet diese. Im DRAM liegen die Texturen und Vertexdaten sowie das resultierende Bild. [25]

Jeder Streaming Multiprocessor hat eine bestimmte Anzahl (In diesem Fall 8) Thread Processors. Jeder dieser Prozessoren führt eine Berechnung durch. In den Special Function Units (SFU) sind Funktionen wie sin, cos, log und exp implementiert. Alle Teile des Streaming Multiprocessor können den gemeinsamen Speicher (Shared Memory) nutzen. [25]

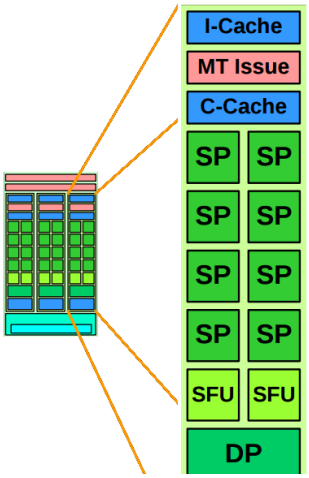

Abbildung 2: Aufbau eines Streaming Multiprocessors (SM) [25]

1.2 Shader

Mithilfe von Shadern ist es möglich Programme auf der GPU auszuführen um das rendern von Objekten zu beeinflussen. Verwendet werden Shader zum Beispiel zur schnellen Berechnung von Licht und Schatten in Videospielen oder ufwendigen Szenen in Animationsfilmen. Aber auch andere Effekte wie Bump Mapping, Normal Mapping oder Light Bloom können mithilfe von Shadern implementiert werden. [4]

Es gibt Vertex, Geometry und Pixel (Fragment) Shader:

Der Vertex Shader führt Transformationen und Berechnungen für die nächsten Shader auf und mithilfe der Vertices aus. Dabei wird jeder Vertex einzeln bearbeitet. [4]

Der Geometry Shader kann basierend auf den vorhanden Vertices neue Vertices erstellen und in die Pipeline einfügen. [4]

Der Pixel (Fragment) Shader mappt die gewünschte Textur auf den zuvor rasterisierten Pixel und färbt diesen bei Bedarf ein. Mit der Färbung kann zum Beispiel Licht simuliert werden. [4]

Aufbau der Grafikpipeline (Logische Ansicht)

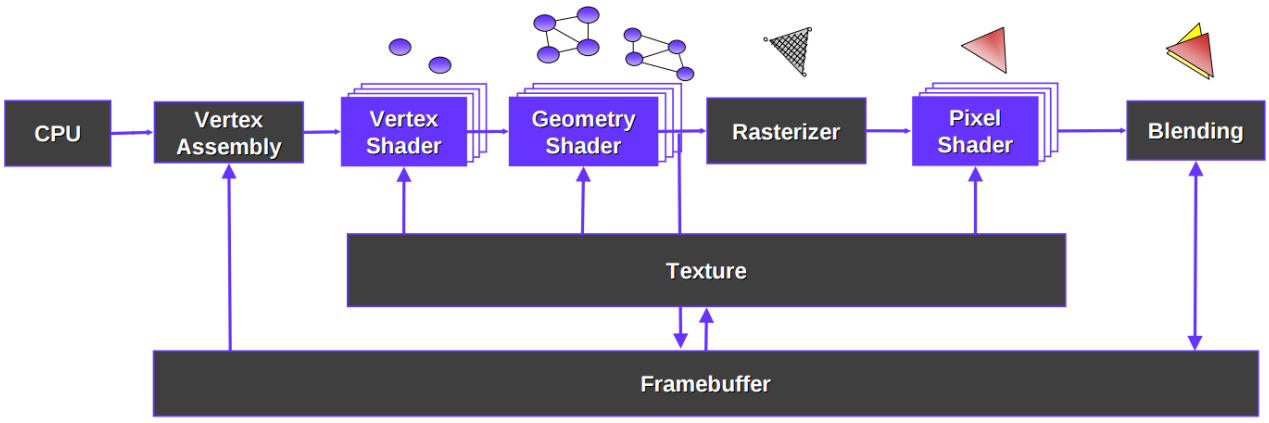

Abbildung 3: Grafikpipeline [25]

Rasterizer: Erzeugt aus den Vertexdaten Pixel
Blending: Legt die Pixel in der gewünschten Reihenfolge übereinander
Texture: Grafik die auf ein Objekt gelegt werden soll
Framebuffer: Hier liegt das Abbild des Bildes das am Monitor angezeigt wird

1.3 Vom Shader zum Kernel

Seit dem Jahr 2007 ist es möglich mithilfe von CUDA von Nvidia aufwendige Berechnungen auf eine GPU auszulagern [16]. Auch AMD hat im selben Jahr mit Close to Metal, heute AMD Firestream, eine API für die eigenen Grafikkarten zur Verfügung gestellt [33]. 2008 wurde OpenCL von der Khronos Group ins Leben gerufen um eine Spezifikation für eine API und Programmiersprache zu definieren die frei von jedermann implementiert werden kann [9].

Eine GPGPU Applikation wird auf einer herkömmlichen Grafikkarte mithilfe der Infrastruktur die für die Shaderprogramme ausgeführt.

1.4 Kritik

Da Grafikkarten zum Rendern vom Grafiken ausgelegt sind, gibt es oftmals schlechten Support für den gängigen IEEE 754 64 Bit Floating Point Datentyp double der von der CPU genutzt werden kann. Spezielle Grafikkarten für den wissenschaftlichen Bereich von Nvidia unterstützen Double Precision Floating Point Zahlen. Auch gibt es Speicher mit Fehlerkorrekturverfahren nur bei den teureren z.B: Nvidia Tesla Modellen. [3]

1.5 GPU vs CPU

Eine CPU besitzt wenige, im Serverbereich bis zu 16, Kerne die für jedmögliche Art von Berechnungen ausgelegt sind. Eine GPU besitzt mehrere Hundert, heute sogar schon bis zu über 1536 Kerne die parallel arbeiten. GPUs sind für mehr oder weniger einfache parallel ablaufende Berechnungen ausgelegt.

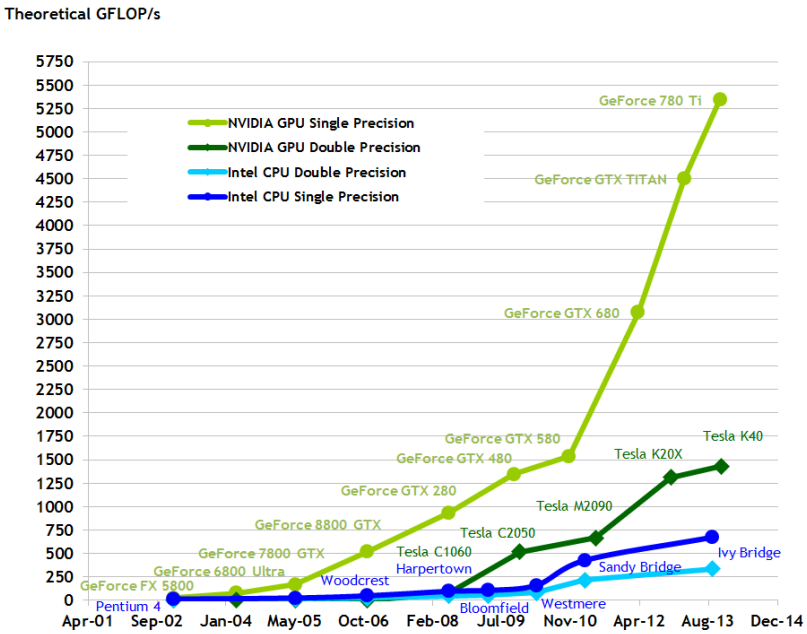

Abbildung 4: GPU vs CPU reine Rechenleistung [40]

Bei der reinen Rechenleistung ist wie auf dem Diagramm ersichtlich die GPU weit vorne. Die Rechenleistung ist in FLOPS (Floating Point Operations), die theoretisch mögliche Spitzenleistung, angegeben.

2 Verwendung

GPGPU wird verwendet um Rechenaufwendige Applikationen zu beschleunigen. Besonders aufwendig sind vor allem die folgenden Anwendungsbereiche:

Simulationswissenschaft

Entwicklung von Modellen, Algorithmen und Software um mithilfe von Computersimulationen Fragestellungen aus Natur und Wissenschaft beantworten zu können. High Performance Computing ist dabei ein wichtiges Werkzeug. [21]

Medizin

GPGPU findet auch Einsatz in der 3D Bildgebung bei Röntgengeräten. [30]

Auch kann die Analyse von Proteinen und die Sequenzierung von DNA beschleunigt werden. [31]

Deep Learning

(Machine Learning, Artificial Intelligence, Neuronale Netzwerke)

Neuronale Netzwerke versuchen das menschliche Gehirn in seiner Funktionsweise zu simulieren. Bei aufwendigen Berechnungen wie Klassifizierung von Bildern (Erkennung von Objekten in Bildern wie Gesichter, Gesichtsausdrücke, Formen oder Gegenstände), sowie Spracherkennung kann GPGPU helfen die Performance zu verbessern um diese Simulationen in Realtime ausführen zu können. [32]

2.1 Vorteile

Grafikkarten sind auf massive Parallelität ausgelegt. Es können je nach Anzahl an programmierbaren Shadereinheiten über 1000 Berechnungen zugleich in einem Takt ausgeführt werden. Applikationen die parallelisierbare Algorithmen (Kapitel Arlgorithmen) verwenden können mit der Verwendung der GPU intensive Berechnungen auslagern und beschleunigen.

2.2 Nachteile

Da eine Grafikkarte parallel arbeitet stellt Sie die Entwickler vor neue Herausforderungen. Algorithmen müssen für den parallelen Einsatz optimiert oder entwickelt werden.

Ein vernünftiges GPGPU Cluster zu betreiben ist sehr teuer. Um auf kurze Sicht Kosten zu sparen können zum Beispiel bei Amazon Webservices GPU Instanzen gemietet werden (siehe Kapitel XaaS)

3 APIs/Frameworks

Im folgenden Abschnitt werden die zwei am meisten verbreiteten GPGPU-Frameworks erläutert. Der Fokus liegt auf dem Open-Source Framework OpenCL. Außerdem wird das proprietäre Framework CUDA von Nvidia beschrieben.

3.1 OpenCL

Einführung

OpenCL (Open Computing Language) ist ein standardisiertes Framework, um Systeme aus einer Kombination von CPUs, GPUs und anderen Prozessoren zu programmieren. OpenCL wurde ursprünglich von Apple entwickelt, um Grafikprozessoren auch für die Ausführung nicht grafischer Anwendungen nützlich zu machen. Mittlerweile ist allerdings die „Khronos Group" für OpenCL verantwortlich. Die OpenCL-Implementierungen variieren von GPU zu GPU. [27]

Heterogene Systeme wurden ein wichtiger Bestandteil von Plattformen und OpenCL war der erste Standard, welcher für solche Systeme nahezu perfekt ausgelegt war. Obwohl OpenCL erst 2009 das erste Mal erschienen ist, besitzt es aus vorher genanntem Grund eine sehr weite Verbreitung. [2, S. 3] [9]

Platform Model

Ein OpenCL-System besteht aus zwei Bestandteilen – einem Host und einem oder mehreren „Compute Devices", auch „OpenCL-Devices" genannt. Ein Compute Device kann eine CPU, GPU oder einen anderen Prozessor darstellen.

Der Host kontrolliert dabei die die Compute Devices. Jedes der Compute Devices besteht aus mehreren „Compute units". Bei einem Mehrkernprozessor sind das die verfügbaren Kerne, welche zusammen die CPU ergeben. Die Compute Units wiederum sind in ein oder mehrere ausführende Elemente, die sogenannten „Processing elements" unterteilt. Die Aufgabe des Hosts ist es, während der Laufzeit die Kernels (Abläufe von Anweisungen) auf die verfügbaren Geräte (Compute Devices) zu verteilen. [2, S. 12] [10]

Die folgende Grafik visualisiert das Plattform Modell:

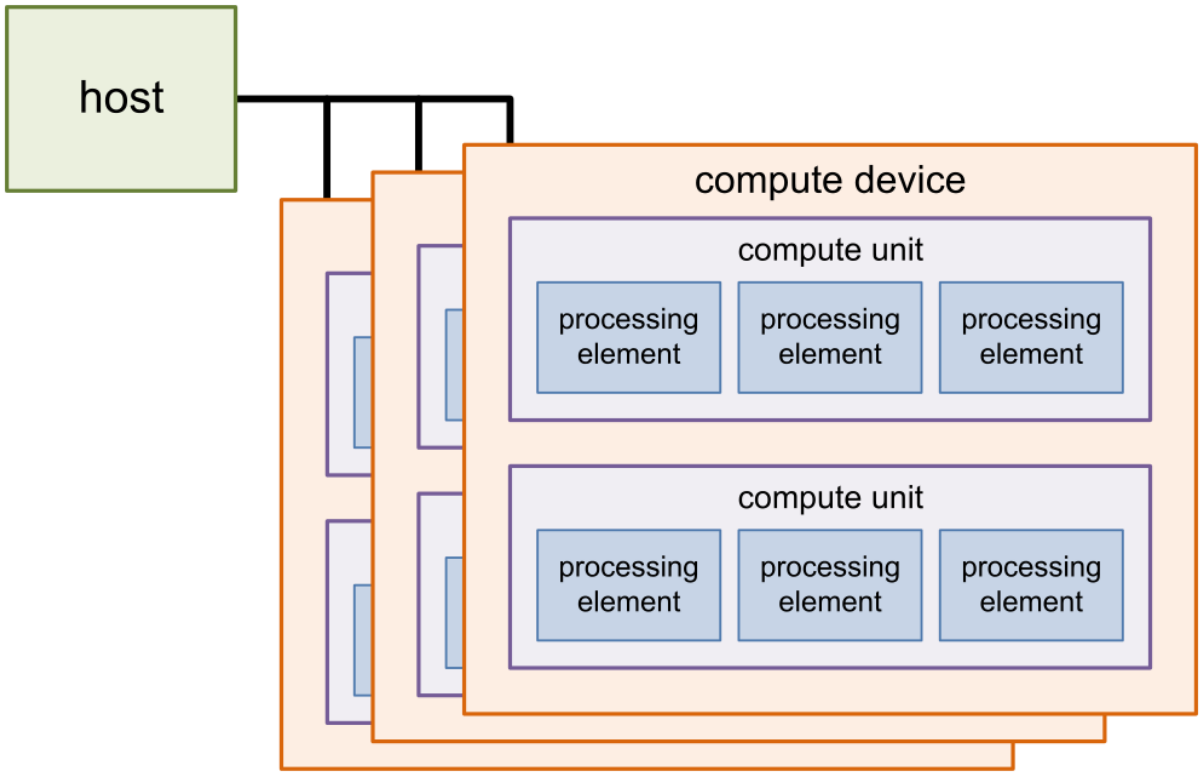

Abbildung 5: OpenCL Plattform Modell [11]

Execution Model

Eine OpenCL Anwendung besteht aus 2 getrennten Teilen – dem Host-Programm und einer Sammlung von einem oder mehreren Kernels. Das Host-Programm läuft – wie der Name bereits verrät – auf dem Host. Die Kernels werden auf den Compute Devices ausgeführt. [2, S.13]

OpenCL definiert zwei Arten von Kernels:

- **OpenCL-Kernel:** Funktionen, welche in der OpenCL C Programmiersprache geschrieben, und mit dem OpenCL-Compiler kompiliert wurden.

- **Native Kernel:** Funktionen, welche außerhalb von OpenCL erstellt wurden. Diese werden mittels function pointer von OpenCL aufgerufen. Native Kernels sind außerdem optional und implementierungsspezifisch.

[2, S. 13]

Ausführung eines Kernels auf einem Compute Device

Das Host-Programm sendet ein Kommando, dass ein Kernel auf einem Compute Device ausgeführt werden soll. Die Berechnungen werden von den „Work-Items" ausgeführt. Als Work-Item wird jede Instanz eines auszuführenden Kernels bezeichnet. Work-Items werden mit einer global ID identifiziert. [2, S. 13]

Das Kommando, welches den Kernel ausführen lässt, erstellt eine Sammlung von Work-Items, wobei jedes Item dieselbe Anweisungssequenz, welche vom auszuführenden Kernel definiert wurde, besitzt. Work-Items einer Work-Group werden nebenläufig auf den Processing Elements einer Compute-Unit ausgeführt.
Es sollte jedoch beachtet werden, dass obwohl die Anweisungssequenzen ident sind, die Verhalten nicht zwingend dieselben sein müssen. Durch beispielsweise selektierte Daten, welche nicht unbedingt gleichbleiben, können die Verhalten variieren. [2, S.13]

Work-Items sind außerdem zu Work-Groups gruppiert, welche auf einen gemeinsamen Speicher zugreifen können. [2, S.13] [12]
Alle Work-Items einer Work-Group, müssen zur selben Zeit durchführbar sein und werden auf einer Compute Unit, also z.B. auf einem Kern, ausgeführt.
Work-Groups werden nicht zwingend parallel ausgeführt. Bei OpenCL werden nur die Work-Items zur selben Zeit durchgeführt, es darf sich aber nicht darauf verlassen werden, dass mehrere Work-Groups parallel ausgeführt werden.
[2, S.14]

Kontext

Die Berechnungen einer OpenCL Applikation finden auf den Compute-Devices statt. Die erste Aufgabe des Hosts ist es, einen Kontext für die OpenCL Applikation zu erstellen. Der Kontext definiert die Umgebung, in welcher die Kernels definiert und ausgeführt werden. Ein Context wird hinsichtlich folgender Ressourcen definiert [2, S.17]:

- **Devices:** Compute-Devices, welche vom Host genutzt werden sollen
- **Kernels:** Funktionen, welche auf den Compute-Devices ausgeführt werden sollen.
- **Program objects:** Quellcode des Programms und die ausführbaren Programme, welche die Kernels implementieren
- **Memory objects:** Eine Sammlung von Objekten im Speicher, worauf die Compute-Devices zugreifen können.

Der Kontext wird vom Host mithilfe von Funktionen der OpenCL-API erstellt und manipuliert.

Angenommen es handelt sich um ein heterogenes System, mit 2 mehrkernigen CPUs und einer GPU.

Devices

Der Host würde die Ressourcen ermitteln und entscheiden, welche Devices genutzt werden. Diese Entscheidung wird von der Art der Berechnung getroffen. Der Host könnte die GPU, eine CPU, bestimmte Kerne der CPU oder Kombinationen daraus wählen. Die Auswahl bestimmt die verwendeten Compute-Devices innerhalb des Kontexts. [2, S.17]

Program objects

Program objects enthalten den Code für die Kernels. Das program object wird vom Host zur Laufzeit erstellt, da im Vorhinein nicht bekannt ist, ob das Programm auf einer CPU, GPU oder auf anderen Prozessoren ausgeführt werden soll. Es ist nur bekannt, dass die Zielplattform mit der OpenCL Spezifikation kompatibel ist. Da der Host die Devices innerhalb des Kontexts definiert, ist danach bekannt, auf welchen Geräten das Programm ausgeführt werden soll. Daher kann der Code nach diesem Schritt kompiliert werden.
[2, S.18] [13, S.195]

Memory objects

Auf heterogenen Systemen existieren meist mehrere Adressräume. Da Geräte verschiedene Speicherarchitekturen besitzen, gibt es in OpenCL die sogenannten memory objects. Diese sind am Host definiert und werden zwischen Host und Compute-Devices „hin und her" verschoben. [2, S.18] [14]

OpenCL definiert 3 Arten von memory objects [13, S.40]:

- **Buffer objects:** Können skalare Datentypen (wie int oder float) oder definierte structs beinhalten. Auf den Buffer kann mithilfe von Pointern zugegriffen werden.

- **Image objects:** Beinhalten Bilder. Das Format von Bild-Elementen ist nicht zwingend das Datenformat, welches im Kernel genutzt wird. Dies hat den Grund, dass eine Optimierung für das jeweilige OpenCL-Device durchgeführt wird. Daher ist sind image objects opak, d.h. es kann nicht direkt auf sie zugegriffen werden. Stattdessen bietet OpenCL Funktionen zur Manipulation von image objects an. Repräsentiert außerdem einen Buffer, welcher als Framebuffer verwendet werden kann.

- **Pipe objects:** Geordnete Sequenz mit 2 Endpunkten. Der eine Endpunkt wird zum Schreiben und Einfügen von neuen Elementen verwendet, der andere zum Lesen und Entfernen von Elementen.

Command-Queues

Die Kommunikation zwischen Host und Compute Devices erfolgt anhand von Commands, welche zur Command-Queue hinzugefügt werden. Command-Queues werden vom Host erstellt und sind an ein Device gebunden.

Folgende Kommandos werden unterstützt [13, S. 27]:

- **Kernel execution commands:** Führt einen Kernel auf den processing elements eines Compute Devices aus.
- **Memory commands:** Überträgt Daten von, zu oder zwischen Memory Objects.
- **Synchronization commands:** Schränkt die Reihenfolge, wie Kommandos ausgeführt werden ein.

Typischerweise definiert der Programmierer den Kontext, die Command-Queues, Memory- und Program-Objects. Memory-Objects werden vom Host auf die Devices verschoben, Kernel-Argumente werden zu Memory-Objects gesetzt und dann zur Command-Queue für die Ausführung hinzugefügt. Sobald der Kernel die Ausführung beendet hat, werden Memory-Objects, welche während der Ausführung erstellt wurden, zurück auf den Host kopiert.

Wenn mehrere Kernels zur Queue hinzugefügt werden, müssen diese eventuell miteinander interagieren. Beispielsweise könnte ein Kernel Memory-Objects erzeugen, die nachfolgende Kernels modifizieren müssen. Für solche Fälle gibt es die synchronization commands, welche eine bestimmte Reihenfolge der Ausführung erzwingen können.

Im Normalfall werden Kommandos immer asynchron zwischen Device und Host ausgeführt, d.h. der Host fügt Kommandos zur Command-Queue hinzu ohne auf die Beendigung der vorherigen Kommandos zu warten. Falls es notwendig ist auf die Beendigung eines Kommandos zu warten, sind synchronization commands notwendig. [2, S. 19]

Die Ausführung der Commands einer Command-Queue erfolgt in einem der zwei Modi:

- **In-order execution:** Kommandos werden in der Reihenfolge ausgeführt, wie sie in der Command-Queue stehen. Ein vorheriges Kommando wird beendet, bevor das nächste ausgeführt wird.

- **Out-of-order execution:** Kommandos werden in der Reihenfolge ausgeführt, wie sie in der Command-Queue stehen. Allerdings wird nicht auf die Beendigung vorheriger Kommandos gewartet.

[2, S.19-20]

Bei der In-order execution ist also eine explizite Synchronisierung nur notwendig, wenn mehrere Compute-Devices auf die selben Daten zugreifen müssen.
Die Out-of-order execution ist aufwändiger zu verwenden, da sie eine explizite Synchronisierung benötigt. Allerdings ermöglicht die asynchrone Ausführung eine effizientere Ausführung.

Memory Model

Das OpenCL Memory Model wird in 5 Abschnitte unterteilt:

- **Host memory**: Dieser Bereich ist nur für den Host zugänglich.
- **Global memory:** Dieser Bereich erlaubt Lese- und Schreibzugriff für alle Work-Items in allen Work-Groups. Work-Items können jedes Element eines memory objects im global memory lesen und modifizieren.
- **Constant memory:** Dieser Bereich bleibt für die Ausführung eines Kernels konstant. Work-Items haben daher nur Leserechte zu memory objects, welche sich im constant memory befinden.
- **Local memory:** Dieser Bereich ist Work-Group spezifisch. Das local memory kann verwendet werden, um Variablen zu allokieren, welche von allen Work-Items innerhalb dieser Work-Group geteilt werden sollen.
- **Private memory:** Dieser Bereich ist Work-Item spezifisch. Variablen, welche im private memory eines Work-Items definiert wurden, sind nicht für andere Work-Items sichtbar.

[2, S.22]

Die folgende Grafik visualisiert das Memory Model:

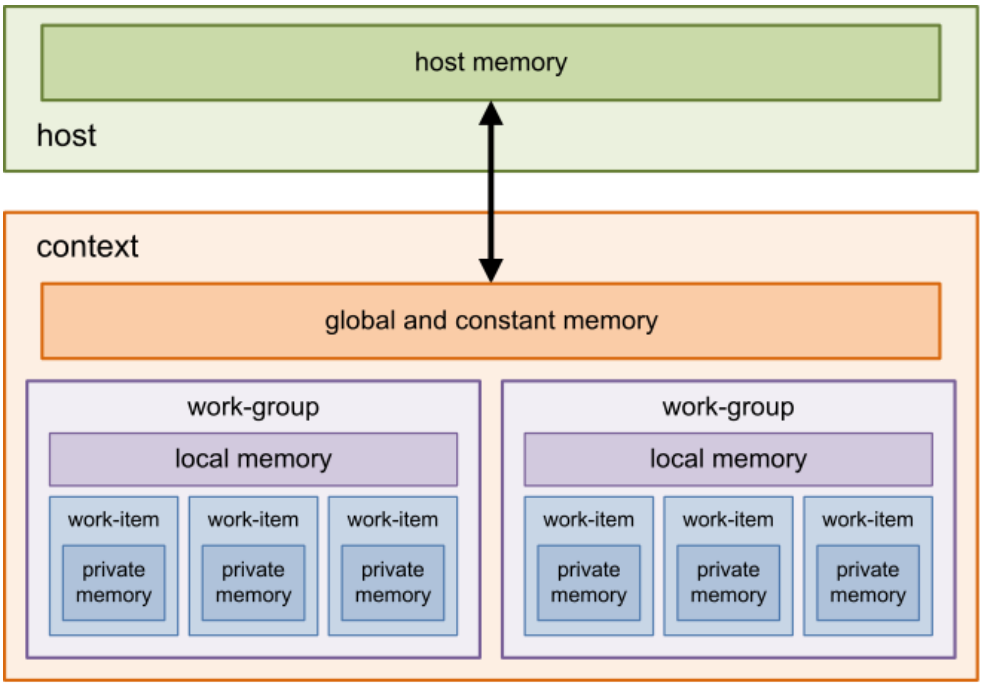

Abbildung 6: OpenCL Memory Model [15]

Die Speichermodelle von Host und Devices sind zum größten Teil unabhängig. Allerdings müssen diese manchmal miteinander interagieren. Dazu können die Daten kopiert werden, oder es kann ein Mapping von Bereichen eines Memory Objects stattfinden.

Um Daten explizit zu kopieren, fügt der Host ein Kommando zur Command-Queue hinzu.
Das Mappen/Unmappen ermöglicht es dem Host einen Bereich eines Memory Objects in seinen eigenen Adressraum zu mappen. Dazu wird ebenfalls ein Kommando zur Command-Queue hinzugefügt.
Sobald ein Bereich gemapped wurde, hat der Host Lese- und Schreibzugriff auf diesen Bereich. Sobald keine weiteren Zugriffe erfolgen, unmapped der Host den Speicherbereich. [2, S.23-24]

Bestandteile von OpenCL

Das OpenCL-Framework ist in folgende Bestandteile unterteilt:

- **OpenCL platform API:** Definiert Funktionen, welche vom Host verwendet werden, um Compute-Devices zu bestimmen und den Context zu erstellen.

- **OpenCL runtime API:** Modifiziert den Kontext um Command-Queues zu erstellen. Funktionen, um Kommandos zu einer Command-Queue hinzuzufügen, stammen ebenfalls von der Runtime API.

- **OpenCL programming language:** Programmiersprache um Code für die Kernels zu schreiben. Sie basiert auf dem ISO C99 Standard.

[2, S.30-31]

Platform API

Plattform bezieht sich in OpenCL auf eine bestimmte Kombination aus Host, OpenCL Devices und dem OpenCL-Framework. Es können mehrere OpenCL Plattformen auf einem heterogenen System existieren. Über die platform API kann nun abgefragt werden, welche Compute-Devices existieren. [2, S.31]

Runtime API

Die Runtime API ist auf Funktionen, welche mit dem Kontext arbeiten, spezialisiert.
Die erste Aufgabe der Runtime API ist es, die Command-Queues zu erstellen. Es kann eine Command-Queue an ein Device gebunden werden, aber es können auch mehrere Command-Queues zur selben Zeit innerhalb eines Kontexts aktiv sein.
Die Runtime API definiert außerdem die Memory und Program Objects und verwaltet diese. [2. S.31-32]

OpenCL programming language

Die OpenCL programming language ist die Programmiersprache, um Kernels zu entwickeln. Sie basiert auf der ISO C99 Sprache. Da dort Funktionen existieren, die nur von CPUs unterstützt werden, wurden manche Funktionen entfernt. Die gelöschten Hauptfunktionen beinhalten:

- Rekursive Funktionen
- Funktionspointer
- Keine Arrays mit variabler Länge

[26, S. 51-53]

Die OpenCL programming language beinhaltet noch weitere Einschränkungen. Diese sind in der OpenCL C -Spezifikation [26] im Abschnitt 6.9 (S.51) zu finden.

Zusammenfassung

Ein Workflow einer OpenCL-Applikation sieht folgendermaßen aus:

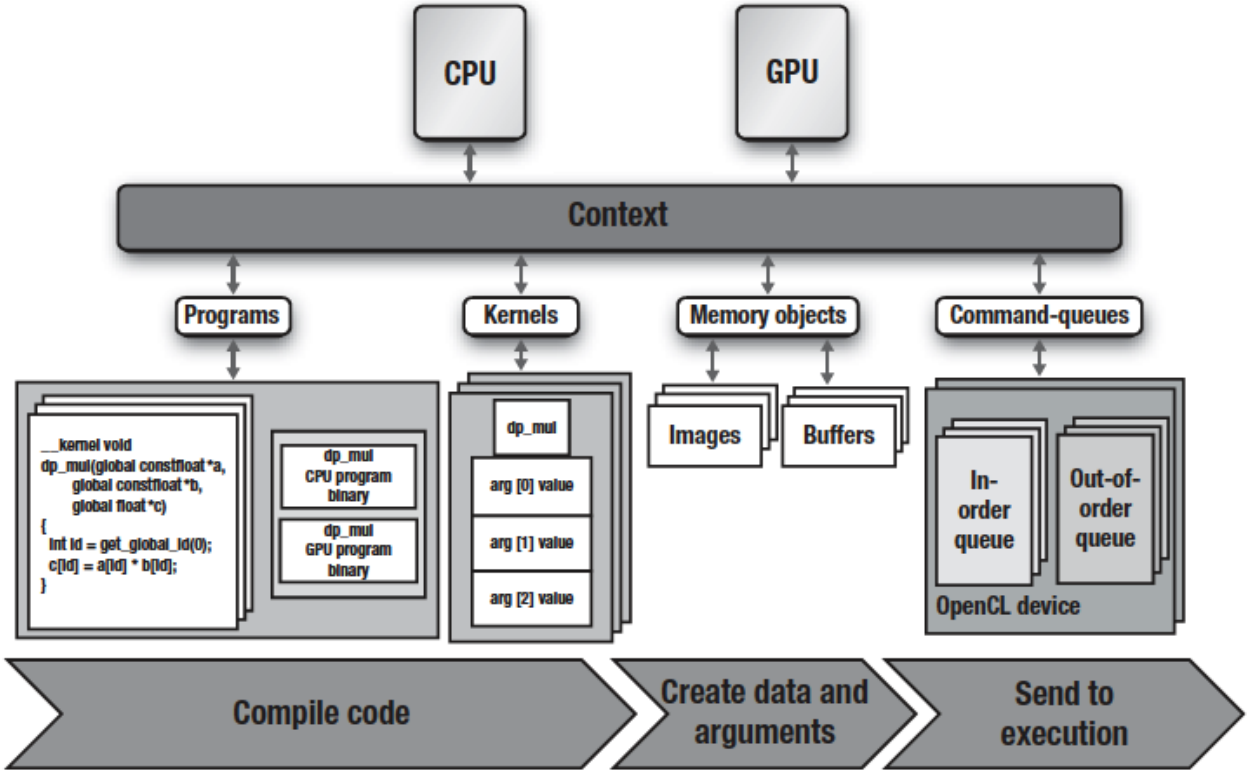

Abbildung 7: OpenCL Workflow [2, S.35]

Als Erstes definiert der Host den Kontext. Dieser beinhaltet in diesem Fall eine CPU und eine GPU. Als Nächstes werden die Command-Queues definiert. In diesem Fall eine in-order queue für die GPU und eine out-of-order queue für die CPU. Danach definiert das Host-Programm ein Program Object, welches kompiliert wird, um Kernels für die beiden Compute-Devices zu generieren. Danach werden vom Host alle benötigten Memory Objects definiert und Argumente der Kernels gesetzt. Abschließend fügt das Host-Programm Kommandos, um die Kernels auszuführen, zur Queue hinzu.

3.2 CUDA

Einführung

CUDA (Compute Unified Device Architecture) ist eine Programmiertechnik, mit der Teile eines Programmes durch die GPU abgearbeitet werden können. CUDA wurde von Nvidia entwickelt und wird daher auch nur von Nvidia Grafikkarten unterstützt. [16]

CUDA verwendet eine C-ähnliche Sprache mit einigen Erweiterungen um GPU-spezifische Funktionen zu verwenden. Wie OpenCL, verwendet CUDA Funktionen, genannt Kernels. [17, S.5]

Architektur

Das Programmierparadigma von CUDA ist eine Kombination aus seriellen und parallelen Ausführungen.

Das nicht-nebenläufige Programm wird auf dem Host (einer CPU) ausgeführt, die nebenläufige Ausführung erfolgt auf Devices. Devices sind Prozessoren, welche auf extreme Parallelisierung ausgelegt sind, beispielsweise GPUs. Dies ist einer der Unterschiede zu OpenCL, wo alle Prozessortypen den Namen Device tragen. Die Anzahl der Threads (Instanzen der Kernels) wird explizit angegeben. [17, S.7]

Programm-Struktur

Ein CUDA-Programm besteht aus mehreren Abschnitten, die entweder am Host, oder auf einem Device ausgeführt werden. Abschnitte, welche keine Nebenläufigkeit erfordern, sind im Host-Code implementiert, während Abschnitte mit viel Parallelisierung im Device-Code implementiert sind.

Code für das Host-Programm ist in ANSI C geschrieben. Kernels werden mit C, welches um einige Funktionen erweitert ist, entwickelt.

Kernels generieren üblicherweise eine große Anzahl an Threads, um die Nebenläufigkeit auszunutzen. CUDA-Threads sind zudem deutlich weniger aufwändig zu generieren und zu schedulen als CPU-Threads. [1, S.41]

Ausführung eines CUDA-Programms

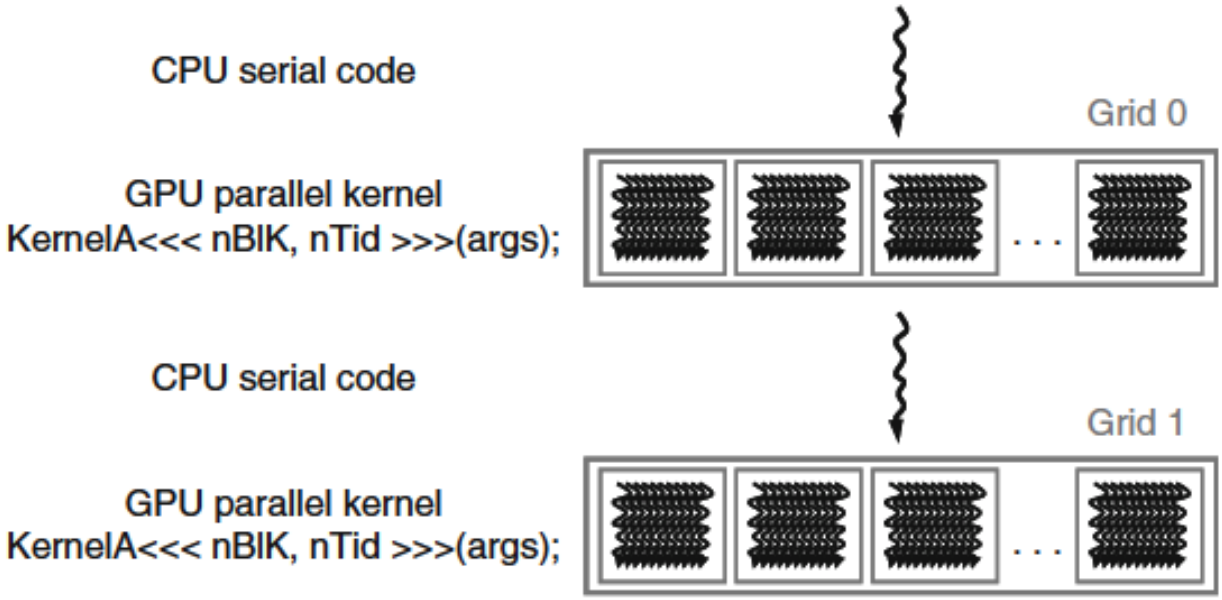

Abbildung 8: Ausführung einer CUDA-Anwendung [1, S.42]

Die Ausführung beginnt mit der Ausführung des Hosts. Sobald ein Kernel aufgerufen wird, wird die Ausführung auf ein Device, also auf eine GPU, verschoben. Dort wird eine gewisse Anzahl von Threads erzeugt. Diese Threads werden zusammengefasst, ähnlich wie eine Work-Group in OpenCL, als *grid* bezeichnet.

Sobald alle Threads eines Kernels ihre Ausführung beendet haben, wird das Grid beendet und die Ausführung wird am Host solange fortgesetzt, bis ein neuer Kernel aufgerufen wird. [1, S.42]

3.3 Sonstige GPGPU-Frameworks

C++ Accelerated Massive Parallelism (Microsoft)

„C++ AMP beschleunigt die Ausführung von C++-Code, indem die Vorteile Daten parallel verarbeitender Hardware, beispielsweise ein Grafikprozessor (Graphics Processing Unit, GPU) auf einer separaten Grafikkarte, genutzt werden. [..]“ – Microsoft Developer Network [28]

Close to Metal, AMD Stream

„Close to Metal“ war die Beta-Version einer GPGPU Programmierschnittstelle von AMD. Die GPGPU Technologie von AMD trägt jetzt den Namen „AMD Stream SDK“. Stream ermöglicht es, die Parallelität von AMD-Grafikkarten für andere Applikationen zu nutzen. [29]

4 Algorithmen

4.1 Gut geeignete Algorithmen für die Ausführung auf der GPU

Hier einige Algorithmen, welche sich gut für die Ausführung auf der GPU eignen.

Map

Eine Map-Operation wendet eine Funktion auf jedes Element einer Sequenz an. Dies kann gut parallel abgearbeitet werden, da keine Kommunikation zwischen den Threads stattfinden muss. Es wird lediglich ein *join* benötigt, um auf die Beendigung aller Threads zu warten. [18, S.3]

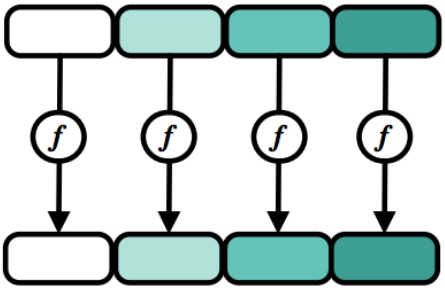

Abbildung 9: Map-Operation [18, S.2]

Die iterative Lösung würde mit einer for-Schleife umgesetzt werden, wobei auf jedes Element nacheinander die gewünschte Operation angewandt wird.

Reduce

Die Reduce-Operation reduziert die Anzahl der Elemente der Eingangssequenz auf weniger oder sogar auf nur ein Element. Beispielsweise könnte ein XOR auf alle angewandt werden. Die Reihenfolge ist irrelevant, weshalb sich eine Parallelisierung anbietet. [18, S.3]

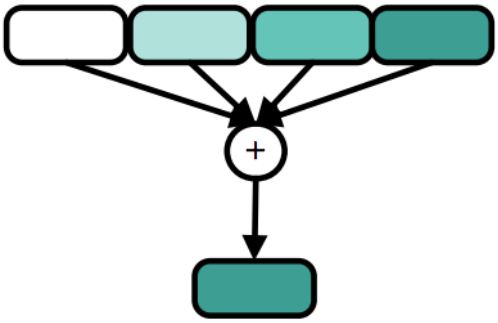

Abbildung 10: Reduce-Operation [18, S.2]

Suche

Eine Suchoperation ermöglicht es, ein bestimmtes Element einer Sequenz zu finden. Bei der Ausführung auf der GPU kann nicht die Suche nach einem Element optimiert werden, aber es können mehrere Suchen gleichzeitig durchgeführt werden.

Sortieren

Eine Sortier-Operation transformiert eine ungeordnete Sammlung an Elementen in eine geordnete Sammlung.

Beispielsweise eignet sich ein Mergesort zur Parallelisierung, da die zerteilten Listen parallel abgearbeitet werden können.

4.2 Benchmarking

Mithilfe von Benchmarks kann die Leistung eines Systems oder Algorithmus anhand von bestimmten Kriterien ermittelt und verglichen werden. [41]

Einiges sollte allerdings beim Durchführen eines Benchmarks beachtet werden. Hier sind einige wichtige Eckpunkte:

Beim Durchführen eines Benchmarks sollte beachtet werden, dass Hintergrundprozesse soweit beendet werden, dass sie die Leistung des Algorithmus nicht beeinflussen.

Ein weiterer beeinflussender Faktor ist die Testumgebung. Falls es sich um einen Software-Benchmark handelt, sollte die Ausführung auf ein und demselben System stattfinden. Des Weiteren sollten mehrere Testläufe durchgeführt werden, und daraus der Mittelwert berechnet werden. Ansonsten könnte durch ein zufällig gutes oder schlechtes Ergebnis ein falscher Eindruck entstehen. [41]

Weiters sollten beim Vergleich zwischen CPU und GPU (in diesem Fall), die beiden Prozessoren ungefähr in derselben Preisklasse liegen, wobei GPUs grundsätzlich teurer sind.

Außerdem sollte beachtet werden, dass beim Vergleich zwischen der Ausführung auf der CPU und auf der GPU der Algorithmus für die Ausführung auf der GPU optimiert worden ist. Dies kann, abhängig vom Algorithmus, relativ aufwändig sein, spiegelt sich aber in einer deutlichen Leistungsoptimierung wieder.

Um einen vertrauenswürdigen Benchmark zu schaffen, sollten bei der Veröffentlichung der Ergebnisse Eckdaten wie CPU, RAM, Mainboard, Betriebssystem und die Anzahl der Testläufe angegeben werden.

Hier ein Vergleich der Performance eines Mergesort-Algorithmus, ausgeführt auf der CPU und auf diversen Nvidia-Grafikkarten:

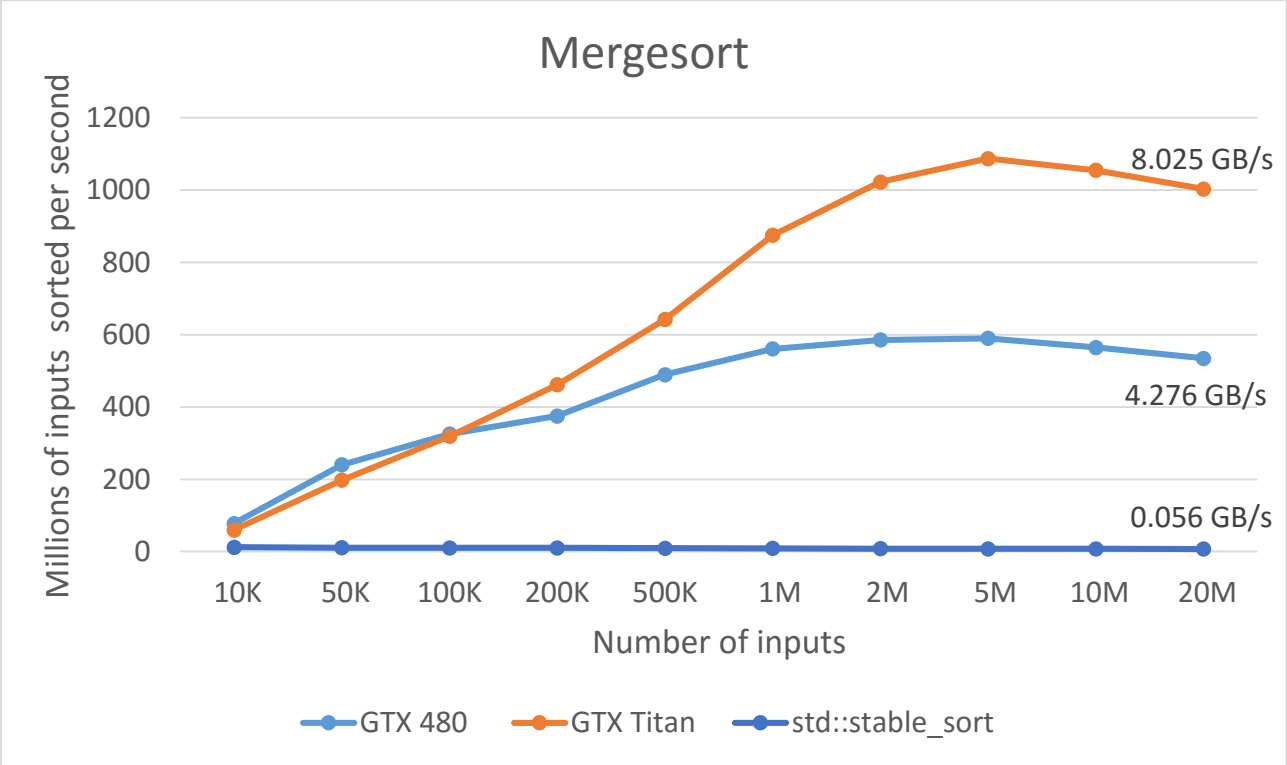

Abbildung 11: Benchmark eines Mergesort-Algorithmus [5]

std::stable_sort ist die Standard Mergesort-Implementierung in C++. Auf den GPUs wurde die Mergesort-Implementierung der Libary moderngpu [5] verwendet.

5 Cloud-Computing

5.1 Allgemein

High Performance Computing

High Performance Computing beschreibt Berechnungen die auf Arbeitsplatzrechnern unmöglich oder unsinnig macht. Dazu gehören vor allem Simulationen von verschiedenen Modellen, Bilderkennung im großen Maße oder Analyse von großen Datenmengen. Um das zu beschleunigen gibt werden mehrere Computer (Nodes) zu einem Cluster zusammengeführt und die Berechnungen auf diesem Cluster gleichmäßig verteilt. So ein Cluster kann auch mit GPUs realisiert werden. [34]

Vorteile

Berechnungen können entweder Realtime oder zumindest in einer annehmbaren Zeitspanne ausgeführt werden. So können zum Beispiel Simulationen nach einer Naturkatastrophe Schäden und weiteren Folgen abwägen und helfen Rettungseinsätze besser planen zu können. [35]

Nachteile

Große Clustersysteme, speziell mit GPUs, sind in der Anschaffung und im Betrieb teuer und können meist nur von großen Unternehmen und Regierungen betrieben werden. [35]

5.2 Umsetzungsmöglichkeiten

VirtualCL

VirtualCL ist ein Framework, dass es ermöglicht OpenCL Applikationen ohne viele Änderungen auf einem heterogenen Cluster von GPUs zu betreiben.

Runtime Model

In jedem Cluster gibt es einen Hosting Node und mehrere Backend Nodes.

Der Hosting Node übernimmt die Administrierung des Clusters. Auf dem Hosting Node wird auch das Programm das auf dem Cluster ausgeführt werden soll mithilfe von vclrun gestartet. Auf dem Hosting Node kann auch ein Backend Node zum Testen oder auch während dem Produktivbetrieb verwendet werden. [20]

Die Backend Nodes lagern dann die Berechnungen die sie ausführen sollen auf die lokalen GPUs aus. Die Backend Nodes können von unterschiedlichen physischen Geräten betrieben werden. [20]

Betriebsmodi

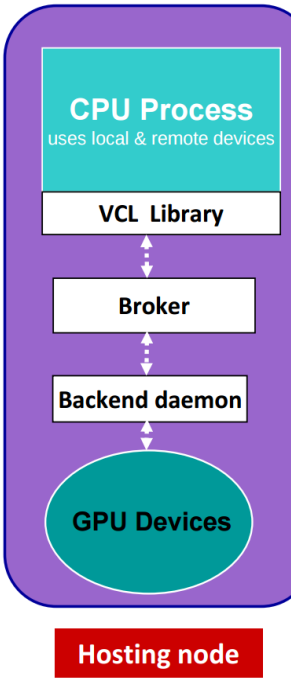

Abbildung 14: Hosting Node u. Backend Node auf einem physischen Gerät [20]

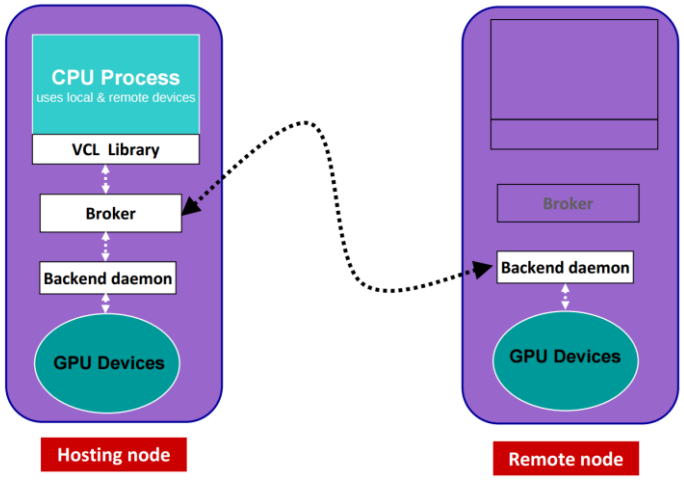

Abbildung 12: Hosting Node mit Backend Node u. externem Backend Node[20]

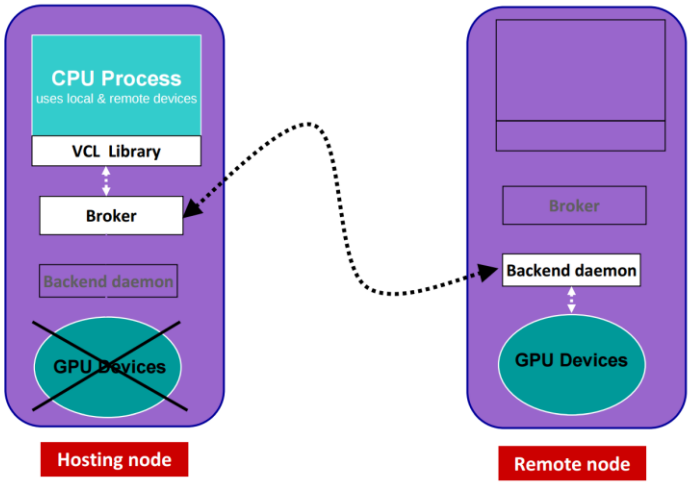

Abbildung 13: Hosting Node und Backend Node getrennt [20]

Vorteile

OpenCL Applikationen können ohne viel Aufwand und Veränderung der Code Base auf einem verteiltem System ausgeführt werden. So stehen mehr Ressourcen in Form von Speichern und Berechnungszeit zur Verfügung. [20]

Nachteile

Die Netzwerk Latenz spielt eine große Rolle. VirtualCL versucht durch diverse Optimierungen den Einfluss auf die Ausführungszeit zu vermindern. [20]

VirtualCL Overhead

Buffer Size	Native time (ms)	VCL Overhead on Local device	VCL Overhead on Remote Device
4KB	96	(96)+35	(96)+113
16KB	100	(100)+35	(100)+ 111
64KB	105	(105)+35	(105)+ 106
256KB	113	(113)+36	(113)+ 105
1MB	111	(111)+34	(111)+ 114
4MB	171	(171)+ 36	(171)+ 114
16MB	400	(400)+36	(400)+ 113
64MB	1,354	(1,354)+33	(1,354)+ 112
256MB	4,993	(4,993)+37	(4,993)+ 111
Average Overhead		**Δ = 35µs**	**Δ = 111µs**

Tabelle 1: VirtualCL Overhead [20]

Installation

VirtualCL wird von http://www.mosix.org/txt_vcl.html/ runtergeladen und entpackt. Danach wird vcl.install ausgeführt und im Setup lokaler Backend Node sowie Host Node erstellt. Das Setup Skript wird gegen Ende ein paar Fehler werfen. [19]

Verwendung

Da das Setup Skript nicht für Debian ausgelegt war musste ich den Host und das Backend manuell starten:

```
sudo /sbin/broker -llocalhost

sudo /sbin/opencld
```

Nun kann ein beliebiges Programm das OpenCL verwendet ausgeführt werden:

```
vclrun ./opencl_helloworld
```

[19]

5.3 rCuda

Mit rCuda gibt es ein Framework wie VirtualCL für CUDA. Es erlaubt ein GPU Cluster zu erstellen und die dadurch bereitgestellten Ressourcen für verschiedene Applikationen zu nutzen. Wert wurde dabei vor allem auf den Stromverbrauch gelegt. Durch den Support für InfiniBand neben TCP, eine serielle Schnittstelle zur Hochgeschwindigkeitsdatenübertragung, wird die Latenzzeit niedrig und der Datendurchsatz hoch gehalten. [23]

6 GPGPU as a service

Bei GPGPU as a service geht es darum, die Hardware nutzerfreundlich in der Cloud zur Verfügung zu stellen. Dabei können Server gemietet oder selber gehostet werden und mit verschiedenen Tools zu Clustern zusammengefasst werden. [37]

6.1 Dienste

Amazon Web Services

Amazon Web Services bieten die Möglichkeit GPU Instanzen zu mieten:

- g2.2xlarge: 0.722$/Stunde, 1 Nvidia GPU
- g2.8xlarge: 3.008$/Stunde, 4 Nvidia GPUs

Als Betriebssystem wird entweder Windows Linux verwendet, unterstützt werden folgende Schnittstellen zur GPU: OpenGL, DirectX, CUDA, OpenCL und das GRID SDK. [32]

NIMBIX

NIMBIX bietet ähnlich wie AWS GPU Computing und HPC Clusters die stündlich abgerechnet werden. [38]

6.2 Frameworks

OpenStack

OpenStack bietet momentan noch keine Unterstützung von GPU Instanzen. Es gibt aber ein Projekte das Support für solche Instanzen implementiert hat, aber inzwischen wieder inaktiv ist.

Das Projekt hat sich zum Ziel Heterogene Systeme zu nutzen um möglichst viele Einsatzbereiche abdecken zu können. So kann der Nutzer im Gegensatz zu zum Beispiel AWS eine Virtuelle Maschine mit der benötigten Zahl an Prozessoren und GPUs starten. [22]

Status

Der Sourcecode ist verfügbar unter https://github.com/usc-isi/nova und hat Support für GPU-enabled LXC Instanzen.

Hoopoe

Hoopoe ist eine GPGPU Infrastruktur die laut Webseite Performance schneller als Realtime bereitstellen kann. Unterstützt wird CUDA und OpenCL und lauffähig ist das Framework auf Windows, Linux, MacOSX und Solaris. [23]

Weitere Informationen zum Produkt kann man nur per E-Mail bekommen.

7 Abbildungsverzeichnis

8 Referenzen

[1] David B. Kirk: *Programming Massively Parallel Processors: A Hands-on Approach [Paperback]*. Morgan Kaufmann, 2010, ISBN 978-0-12-381472-2.
Online verfügbar unter: http://analog.nik.uni-obuda.hu/ParhuzamosProgramozasuHardver/02_GPGPU-Irodalom/02_GPGPU-Irodalom_MagyarBalint/Programming%20Massively%20Parallel%20Processors.pdf
[zuletzt abgerufen: 15.10.2015]

[2] Aaftab Munshi: *OpenCL programming guide*. Addison-Wesley, 2012, ISBN 0-321-74964-2

[3] Matt Pharr: *GPU gems2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley, 2005, ISBN 0-321-74964-2

[4] Hubert Nguyen: *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005, ISBN 0321515269

[5] Sean Baxter. Modern gpu, 2013. Abrufbar unter: http://nvlabs.github.io/moderngpu/
[zuletzt abgerufen: 15.10.2015]

[6] Kai Hwang; Geoffrey C. Fox; Jack J. Dongarra: Distributed and cloud computing: from parallel processing to the internet of things. Elsevier, Morgan Kaufmann, 2012, ISBN 978-0-12-385880-1

[7] Khaled M. Diabt, M. Mustafa Rafique, Mohamed Hefeeda: Dynamic Sharing of GPUs in Cloud Systems. Abrufbar unter: http://qcri.org.qa/app/media/1776 [zuletzt abgerufen: 23.10.2015]

[8] Brian Schott, John Paul WaltersUSC Information Sciences Institute: HeterogeneousGpuAcceleratorSupport. Abrufbar unter: https://wiki.openstack.org/wiki/HeterogeneousGpuAcceleratorSupport [zuletzt abgerufen: 04.11.2015]

[9] Khronos Group: OpenCL. Abrufbar unter:
https://www.khronos.org/opencl/
[zuletzt abgerufen: 29.11.2015]

[10] Jonathan Tomposon, Kristofer Schlachter: An Introduction to the OpenCL Programming Model. Abrufbar unter:
http://www.cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf
[zuletzt abgerufen: 29.11.2015]

[11] „Platform architecture 2009-11-08" von Björn König - self-made. Lizenziert unter CC BY-SA 3.0 über Wikipedia -
https://de.wikipedia.org/wiki/Datei:Platform_architecture_2009-11-08.svg#/media/File:Platform_architecture_2009-11-08.svg

[12] Wikipedia, Die freie Enzyklopädie: OpenCL. Abrufbar unter:
https://de.wikipedia.org/w/index.php?title=OpenCL&oldid=146958162
[zuletzt abgerufen: 29.11.2015]

[13] Khronos Group: The OpenCL Specification 2.1. Abrufbar unter:
https://www.khronos.org/registry/cl/specs/opencl-2.1.pdf
[zuletzt abgerufen: 05.12.2015]

[14] Mohamed Zahran: Graphics Processing Units (GPUs): Architecture and Programming, OpenCL. Abrufbar unter:
http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture11.pdf
[zuletzt abgerufen: 05.12.2015]

[15] „OpenCL Memory model" von Björn König - self-made. Lizenziert unter CC BY-SA 3.0 über Wikipedia -
https://de.wikipedia.org/wiki/Datei:OpenCL_Memory_model.svg#/media/File:OpenCL_Memory_model.svg

[16] Nvidia Cuda. Abrufbar unter:
http://www.nvidia.com/object/cuda_home_new.html
[zuletzt abgerufen: 07.12.2015]

[17] Rafia Inam: An Introduction to GPGPU Programming – CUDA Architecture. Abrufbar unter:
http://www.diva-portal.org/smash/get/diva2:447977/FULLTEXT01.pdf
[zuletzt abgerufen: 07.12.2015]

[18] Samadi, Mehrzad; Jamshidi, Davoud Anoushe; Lee, Janghaeng; Mahlke, Scott (2014). Paraprox: Pattern-based approximation for data parallel applications.
Abrufbar unter: http://cccp.eecs.umich.edu/papers/samadi-asplos14.pdf
[zuletzt abgerufen: 07.12.2015]

[19] Barak; Shiloh (July 2013). VirtualCL (VCL) Cluster Platform Programmer and Administrator Guide and Manuals
Abrufbar unter: http://www.mosix.cs.huji.ac.il/vcl/VCL_Guide.pdf
[zuletzt abgerufen: 08.12.2015]

[20] Barak; Shiloh. The VirtualCL (VCL) Cluster Platform
Abrufbar unter: http://www.mosix.cs.huji.ac.il/vcl/VCL_presentation.pdf
[zuletzt abgerufen: 08.12.2015

[21] Strategische Weiterentwicklung des Hoch- und Höchstleistungsrechnens in Deutschland (2014)
Abrufbar unter: http://www.wissenschaftsrat.de/download/archiv/1838-12.pdf
[zuletzt abgerufen: 08.12.2015]

[22] John Paul Walters. CUDA in the Cloud – Enabling HPC Workloads in OpenStack
Abrufbar unter: http://on-demand.gputechconf.com/gtc/2013/presentations/S3214-Enabling-HPC-Workloads-OpenStack.pdf [zuletzt abgerufen: 08.12.2015]

[23] Hoopoe™ – GPU Cloud Computing
Abrufbar unter: http://www.cass-hpc.com/solutions/hoopoe/
[zuletzt abgerufen: 08.12.2015]

[24] Federico Silla. Virtualización remota de GPUs
Abrufbar unter: http://www.rcuda.net/pub/rCUDA_cordoba_15.pdf
[zuletzt abgerufen: 08.12.2015]

[25] Ashu Rege. An Introduction to Modern GPU Architecture
Abrufbar unter:
ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf
[zuletzt abgerufen: 08.12.2015]

[26] Khronos Group: The OpenCL C Specification 2.0. Abrufbar unter:
https://www.khronos.org/registry/cl/specs/opencl-2.0-openclc.pdf
[zuletzt abgerufen: 07.12.2015]

[27] Khronos Launches Heterogeneous Computing Initiative. Abrufbar unter: https://www.khronos.org/news/press/khronos_launches_heterogeneous_computing_initiative [zuletzt abgerufen: 11.12.2015]

[28] Microsoft Developer Network: C++ AMP. Abrufbar unter: https://msdn.microsoft.com/de-de/library/hh265137.aspx [zuletzt abgerufen: 11.12.2015]

[29] AMD: AMD Stream Technology. Abrufbar unter: http://www.amd.com/en-us/innovations/software-technologies/firepro-graphics/stream [zuletzt abgerufen: 11.12.2015]

[30] NVIDIA-Grafikprozessoren beschleunigen medizinische Bildverarbeitung Abrufbar unter: http://www.nvidia.de/object/ziehm-imaging-de.html [zuletzt abgerufen 14.12.2015]

[31] Bioinformatics and Life Sciences ttp://www.nvidia.com/object/bio_info_life_sciences.html [zuletzt abgerufen: 14.12.2015]

[32] Simon Haykin. Neural Networks and Learning Machines http://www.mif.vu.lt/~valdas/DNT/Literatura/Haykin09/Haykin09.pdf [zuletzt abgerufen: 14.12.2015]

[33] Close to Metal Abrufbar unter: https://en.wikipedia.org/wiki/Close_to_Metal [zuletzt abgerufen: 14.12.2015]

[34] Hoffman, Allan R.; et al. (1990). Supercomputers: directions in technology and applications. National Academies. ISBN 0-309-04088-4.

[35] Parallel computing for real-time signal processing and control by M. O. Tokhi, Mohammad Alamgir Hossain 2003. ISBN 978-1-85233-599-1.

[36] Amazon AWS HPC Abrufbar unter: https://aws.amazon.com/de/hpc/ [zuletzt abgerufen: 14.12.2015]

[37] The NIST Definition of Cloud Computing http://www.slideshare.net/crossgov/nist-definition-of-cloud-computing-v15 [zuletzt abgerufen: 14.12.2015]

[38] NIMBIX Website
http://www.nimbix.net/
[zuletzt abgerufen: 14.12.2015]

[39] Lecture 0: CPUs and GPUs. Prof. Mike Giles, Oxford University Mathematical Institute.
Abrufbar unter: http://people.maths.ox.ac.uk/gilesm/cuda/lecs/lec0.pdf
[zuletzt abgerufen: 14.12.2015]

[40] Cuda Toolkit Documentation
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture
[zuletzt abgerufen: 14.12.2015]

[41] Benchmarking. William M. Lankford, State University of West Georgia.
Abrufbar unter:
https://www.coastal.edu/business/cbj/pdfs/benchmark.pdf
[zuletzt abgerufen: 14.12.2015]