Seminar Multiprocesser on Chip
Seminararbeit

General-purpose computing on GPUs

Institut für Informatik
Universität Potsdam

eingereicht von:
Michael Winkelmann (738901)

Sommersemester 2010

Zusammenfassung

Als General Purpose Computation on Graphics Processing Unit, kurz GPGPU, bezeichnet man die Verwendung des Grafikprozessors für Berechnungen außerhalb der Grafikberechnungen, seinem eigentlichen Aufgabenbereich. Die Anwendungspalette von GPGPU reicht von Datenbankberechnungen, geometrische Berechnungen (wie Physik-Berechnungen in Computerspielen) über bildgebende Verfahren in der Medizin oder auch in der Bioinformatik.

In meiner Arbeit werde ich zunächst die Grafikpipeline und die Funktionsweise einer GPU näher beschreiben. Anschließend werde ich die Unterschiede zwischen CPUs und GPUs in puncto Hardware-Aufbau, Daten und Kontrollfluss, die Eignung verschiedener Algorithmen näher beleuchten. Danach werde ich genauer auf die Programmierung von Grafikkarten eingehen und abschließend werde ich die genannten Fakten zu diskutieren und versuchen, einen Ausblick auf künftige Entwicklungen zu geben.

Inhaltsverzeichnis

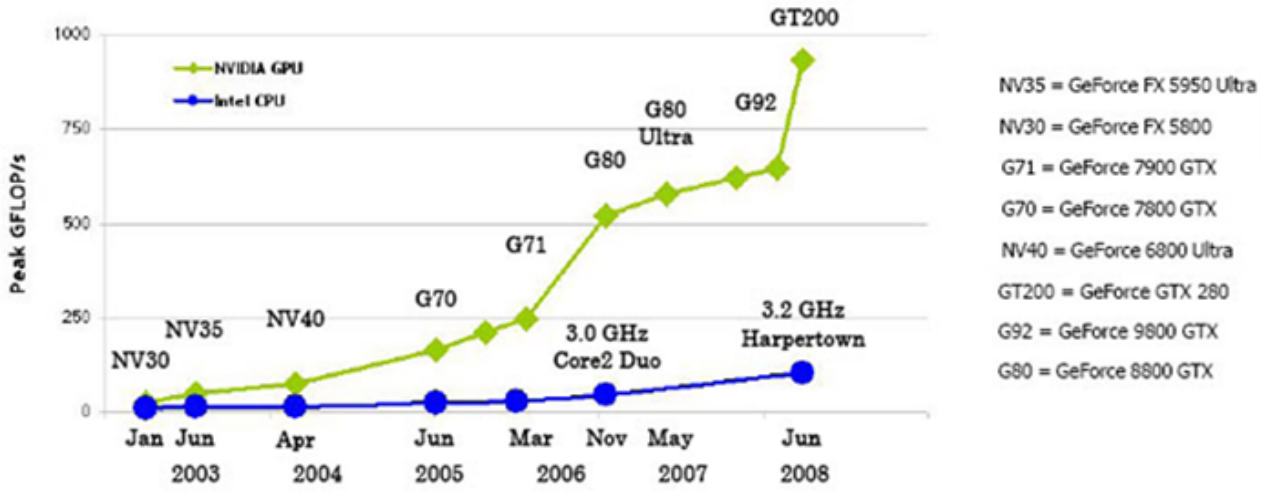

Abbildung 1: Die Rechenleistung von GPUs ist gegenüber den CPUs in den letzten Jahren viel schneller angestiegen. Die Werte in GFLOPS/s sind theoretische Maximalwerte. *Quelle: (Kir06)*

1 Einführung

In den frühen 90er Jahren war interaktives und realistisches 3D-Rendering nur Supercomputern vorbehalten. Es bestand jedoch auch eine hohe Nachfrage an Anwendungen mit interaktiven 3D-Inhalten, meistens Computerspiele. Jedoch enthielt bereits zum Ende dieses Jahrzehnts jeder neuere Computer eine separate Graphics Processing Unit (GPU), die interaktive 3D-Grafik bei hohen Bildraten möglich machte.

Allerdings waren diese Grafikkarten früher statisch und somit nicht programmierbar. Sie konnten lediglich Daten nach einem festen Schema verarbeiten. Mit der Einführung von programmierbaren Shader-Einheiten mit DirectX 8 im November 2000 erhielten Grafikkarten erstmals Möglichkeiten, abweichend von dieser statischen Berechnungsweise kleinere Programme (die sogenannten Shader) auszuführen zu können. Von nun an war es prinzipiell möglich, beliebige, nicht unbedingte grafische Daten zu mit Hilfe von Grafikkarten verarbeiten. Es gab zwei Arten von DirectX-8-Shadern: Die Vertex-Shader, welche sich um Geometrie-Berechnungen kümmern und die Pixelshader, welche sich um die Berechnung von einzelnen Pixeln kümmern.

Mit der Einführung von Shadern begann auch die Verwendung von Grafikkarten für grafik-fremde Berechnungen, das General Purpose Computing on GPUs (GPGPU), mit dem sich die vorliegende Arbeit auseinandersetzt.Für GPGPU wurden meist die Pixelshader eingesetzt. Trotzdem war deren Flexibilität noch zu eingeschränkt, um größere Berechnungen durchführen zu können.

Durch die Nachfrage an rechenaufwändigen visuellen Effekten entwickelt sich die Leistung von GPUs seit mehreren Jahren schneller als nach dem Mooreschen Gesetz vorgesehen. Die Leistung hat sich 2006 innerhalb von 18 Monaten nicht verdoppelt, sondern verfünffacht. Heute ist die Rechenleistung von GPUs bei bestimmten Anwendung denen der CPU weit überlegen, denn Grafikprozessoren bieten eine hohe Parallelleistung, die mit heutigen CPU unmöglich zu erreichen ist. Weil man diese Rechenleistung nicht nur für Grafikeffekte nutzen kann, stieg das Interesse an GPGPU enorm.

Mit der Einführung von DirectX im Jahr 2007 wurden Vertex- und Pixelshader zum Unified Shader Model vereinigt. Erst damit wurde GPGPU wirklich anwendungstauglich. Mit der Vereinheitli-

chung des Shadermodells haben sich die GPUs zu einer festfunktionalen 3D-Grafik-Pipeline hin zu einem flexiblen Allzweck-Parallelprozessor entwickelt. GPUs stellen somit die ersten weitreichend verfügbaren Desktop-Parallel-Rechner dar. Durch das Bereitstellen von speziellen Entwicklerwerkzeugen wurde die Entwicklung von GPGPU-Programm zudem vereinfacht. Für wissenschaftliche Zwecke sind außerdem höhere Genauigkeiten und weniger Rechenfehler wichtig, woraufhin Fehlererkennungsalgorithmen in die Grafikkarten implementiert wurden und sie mit auch mit doppeltgenauen Gleitkommazahlen umgehen können.

Die Anwendungspalette von GPGPU reicht von Datenbankberechnungen, geometrische Berechnungen (wie Physik-Berechnungen in Computerspielen) über bildgebende Verfahren in der Medizin oder auch in der Bioinformatik. Es wurde auch schon Ray-Tracing und Global Illumination sowie für die Kollisionserkennung eingesetzt.

In meiner Arbeit werde ich zunächst die Grafikpipeline näher beschreiben. Anschließend werde ich die Unterschiede zwischen CPUs und GPUs in puncto Hardware-Aufbau, Daten und Kontrollfluss, die Eignung verschiedener Algorithmen näher beleuchten. Danach werde ich genauer auf die Programmierung von Grafikkarten eingehen. Abschließend werde ich die genannten Fakten zu diskutieren und versuchen, einen Ausblick auf künftige Entwicklungen zu geben.

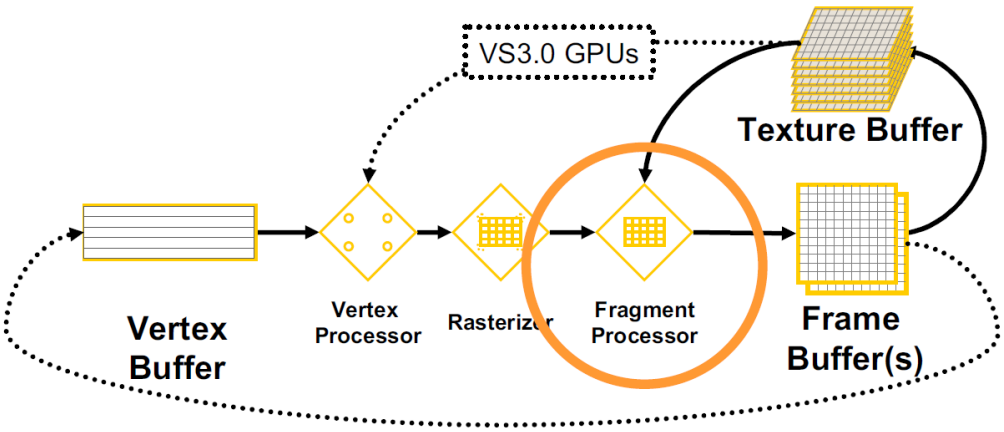

Abbildung 2: Die Grafikpipeline. Mit Einführung des Unified Shader Models konnten auch die Vertex-Prozessoren auf den Texturspeicher zugreifen. *Quelle: (Kic07)*

2 Die Grafikpipeline

Die Grafik-Pipeline ist ein Modell, welches beschreibt, welche Schritte ein Grafiksystem zum Rendern eines Bildes ausführen muss. OpenGL und Direct3D sind die Praxis am häufigsten anzutreffenden Grafikpipeline-Modelle. Die Pipeline besteht im Wesentlichen aus drei Stufen (DMZ09).

Die erste Stufe ist die Berechnung der Vertices, also der eigentlichen dreidimensionalen Geometrie. Dabei werden die dreidimensionalen Vertex-Koordinaten in zweidimensionale Bildschirmkoordinaten umgerechnet. Bei der Rasterisierung werden die Vertices in Fragmente konvertiert, die aus den Bildschirmpixeln bestehen. Anschließend erfolgt das Fragment-Processing, bei dem für jeden Pixel durch Lesen der Texel aus dem Textur-Speicher ein Farbwert berechnet wird.

Die Grafikpipeline ist für den Rendering-Prozess speziell angepasst und erlaubt es der GPU, wie ein Stream-Prozessor zu arbeiten, denn alle Vertices und Fragmente können unabhängig voneinander betrachtet werden (Per06). Durch das unabhängige Betrachten der zu verarbeitenden Daten können diese auch parallel abgearbeitet werden.

Bei den ersten Grafikchips waren die Berechnungsabläufe der einzelnen Schritte direkt in die Hardware implementiert. Mit der Einführung programmierbarer Grafikchips 2001 konnten sowohl Vertex- als auch Fragment-Prozessor programmiert werden. Die Berechnungen auf verschiedenen Vertices und Fragmenten erfolgt durch das Benutzen vieler gleichartiger Prozessoren parallel. Beispielsweise besitzt die nVidia 7900 GTX 8 Vertex- und 24 Fragmentprozessoren. Für GPGPU werden normalerweise nur die Fragment-Programme für die Berechnungen eingesetzt.

Mit dem Unified Shader Model (Shader Model 3.0) wurden Vertex- und Fragment-Prozessor vereinigt, so dass eine flexiblere Programmierung möglich wurde. Alle Recheneinheiten der Grafikkarte haben Zugriff auf den Hauptspeicher der Grafikkarte und können gemeinsam die gleichen Berechnungen ausführen. Für GPGPU ist dies besonders von Vorteil, da die früher nicht-benötigten Vertex-Prozessoren nun nicht mehr brachliegen (LH07).

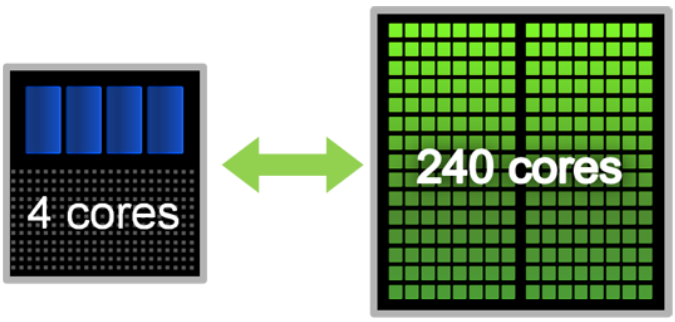

Abbildung 3: Schematischer Vergleich zwischen CPU und GPU. *Quelle: (Zib10)*

	Rechenleistung	Speicherbus-Datenrate
ATI RV770	1200 GFlops	n/a
NVIDIA GeForce GT200	1063 GFlops	n/a
NVIDIA GeForce 6800	60 GFlops	18 GByte/s
Intel Core 2 Quad Q6600	21,4 GFlops	n/a
Intel Pentium 4	14,4 GFlops	5 GByte/s

Abbildung 4: Vergleich der Rechenleistung und Speicherbandbreite zwischen aktuellen Grafikchips und CPUs. *Quelle: (Owe06)*

3 Unterschiede zwischen CPU und GPU

Die Funktionalität der GPU ist im Gegensatz zur CPU sehr eingeschränkt. Jedoch kann eine GPU im Gegensatz zu einer CPU hochgradig parallelisierbare Aufgaben ausführen. Eine GPU besitzt viele kleine spezielle Recheneinheiten, während eine CPU über wenige große Recheneinheiten verfügt, die einen gemeinsamen Cache besitzen (siehe Abbildung oben).

3.1 Speicher und Cache

Wenn man auf der GPU ein Programm ausführen würde, welches nur einen Thread und wenig Datenparallelität besitzt, so wie es bei aktuellen Anwendungsprogrammen üblich ist, dann wären die vielen kleinen Rechenkerne der GPU nicht ausgelastet. Hohe Ausführungsgeschwindigkeiten werden also hauptsächlich durch den hohen Grad an Parallelität der Rechenoperationen des Grafikprozessors erreicht.

Die GPU hat kaum Logiken für Steuerungsaufgaben und Caching implementiert, dafür jedoch wird der Großteil der Transistoren für Rechenoperationen verwendet, wodurch eine höhere Rechenleistung erzielt werden kann.

CPUs besitzen einen sehr schnellen und großen Cache (bei aktuellen CPUs ca. 8 MB), wodurch Daten für spätere Berechnungen wiederverwendet werden können. Für den Cache wird allerdings auch ein Drittel der Transistoren der CPU benötigt. GPUs besitzen einen deutlichen kleineren Cache, dafür eine sehr schnelle Anbindung an den Hauptspeicher, der sich meist direkt on-board befindet. Der GPU-Speicher ist für das Verarbeiten von Datenströmen ausgelegt. Für die Verringerung von Latenzen findet ein Prefetching der Daten statt, wodurch die Zahl der Taktzyklen für den Zugriff auf den Speicher auf ein Minimum reduziert werden kann. Durch diese schnelle Anbindung eignen sich GPUs auch für die Verarbeitung großer Datensätze (Hou05).

Die relativ kleinen Caches in der GPU würden bei nicht parallelisierten Programmen zu größeren Latenzen in der Programmausführung führen, die nicht durch gleichzeitiges Abarbeiten vieler

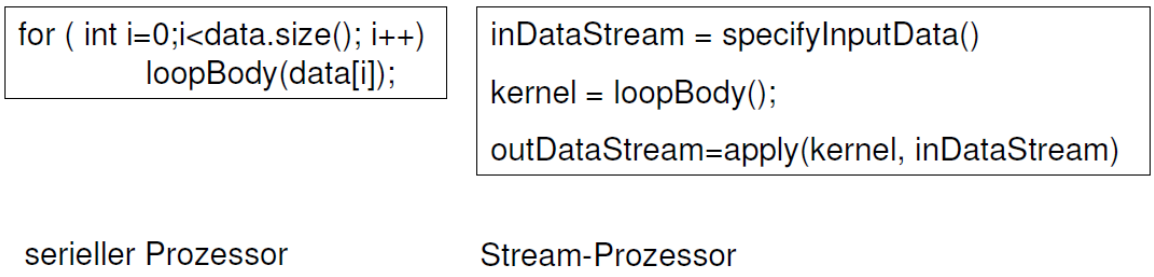

Abbildung 5: Vergleich von zwei Pseudocodes für Streamprozessoren und für serielle Prozessoren, den CPUs.*Quelle: (Kir06)*

Aufgaben ausgeglichen werden könnten. Bei sequentiellen Aufgaben ist die CPU daher schneller, sie sind für eine hohe Performance auf einem einzelnen Thread ausgelegt. Außerdem können CPUs viel besser als GPUs mit Verzweigungen umgehen und mehrere verschiedene Threads, die verschiedene Aufgaben ausführen, gleichzeitig behandeln (LMW07).

Dadurch sind GPUs für solche Anwendungen besonders gut geeignet, die eine hohe arithmetische Dichte aufweisen. Dies bedeutet, dass viele Rechenoperationen, aber nur wenige Lese- und Schreibbefehle ausgeführt werden müssen. Die einzelnen Recheneinheiten auf der GPU können untereinander keine Daten austauschen und nicht untereinander kommunizieren. Eine GPU ist daher besonders gut für solche Einsatzgebiete geeignet, bei denen die Daten keine Abhängigkeiten untereinander aufweisen. Die ist besonders in der Bildverarbeitung der Fall, da man ein Bild beispielsweise aufteilen und jedem Rechenkern der GPU auf einem Bildausschnitt arbeiten lassen kann (Buc05).

Jedoch ist der Hauptspeicher von Grafikkarten üblicherweise kleiner als bei CPUs (beispielsweise 1GB bei einer aktuellen GPU gegenüber 8GB bei einer aktuellen CPU), was man jedoch durch spezielle GPGPU-Grafikkarten mit mehr Speicher, aber dafür ohne Bildschirmausgabe zu kompensieren versucht (z. B. Nvidia Tesla, AMD FireStream). Der Speicher dieser Grafikkarten verfügt auch über spezielle Fehlerkorrekturverfahren , denn die Speicherbausteine von Grafikkarten für Endverbraucher sind meist von schlechterer Qualität, so dass für Berechnungen entsprechende Maßnahmen ergriffen werden müssen, damit sichergestellt werden kann, dass die Ergebnisse korrekt sind (DMZ09). Dies ist besonders bei wissenschaftlichen Anwendung von Bedeutung.

Somit besitzt eine ideale GPGPU-Anwendung einen großen Datensatz, hohen Parallelismus und eine minimale Abhängigkeit zwischen den Daten. Ein Vergleich zwischen den Speicherbandbreiten und Rechenleistung ist in Abbildung 4 dargestellt.

3.2 Preise und Kompabilität

Ein weiterer Vorteil ist der geringe Preis im Vergleich zu ähnlich schnellen anderen Lösungen und die Tatsache, dass Grafikkarten heute in nahezu jedem PC zu finden sind. Beispielsweise kann erreicht eine ATI Radeon R800 eine Rechenleistung von bis zu 2 TeraFlops bei einem Preis von 500, während ein aktueller Intel Quad-Core für den gleichen Preis auf lediglich 30-40 GFlops kommt (Zib10).

Jedoch sind viele von der GPU gelösten Aufgaben sind nicht einheitlich spezifiziert und die Unterschiede zwischen den Herstellern sind viel größer als die bei CPUs, da die GPU-Hersteller keinen einheitlichen Befehlssatz (wie z.B. x86 bei AMD und Intel) verwenden. Dadurch erhöhen sich der Entwicklungsaufwand und Kosten und eine Portierbarkeit der GPGPU-Programme ist problematisch.

3.3 Kontroll- und Datenfluss

Eine GPU ist ein Stream-Prozessor. Das bedeutet, dass die einströmenden Daten keine Abhängigkeiten aufweisen, sich sehr ähnlich sind und deshalb parallel verarbeitet werden können. Das Stream-Modell hat den Vorteil, dass es weniger komplex als die traditionelle parallele Programmierung ist. Stream-Programme lassen sich als Graphen darstellen. Die Knoten des Graphen sind dabei die Kernels, welche die Fragment-Programme darstellen, die die eigentlichen Berechnungen ausführen. Die Kernels bieten sowohl Instruktionsparallelismus und Datenparallelismus (durch SIMD-Befehle realisiert). Die Verbindungen zwischen den Graphenknoten ist ist der Datenstrom, der Stream. Die Daten befinden sich im Texturpuffer. Der Datenfluss ist bei einer GPU also auf höchstmögliche Parallelität ausgelegt. Ein Programmierbeispiel zwischen seriellen und Streamprozessoren ist in der Abbildung 5 dargestellt.

Der Kontrollfluss bei einer GPU ist hingegen weniger flexibel als bei einer CPU. Zwar können alle aktuellen Grafikkarte mit Verzweigung, Schleifen und Unterprogrammen umgehen, jedoch geht bei deren häufigem Einsatz der Geschwindigkeitsvorteil gegenüber der CPU verloren. Verzweigung sollten besonders bei inneren Schleifen vermieden werden.

3.4 Kommunikation zwischen CPU und GPU

Die GPUs sind mit den restlichen Komponenten des Computers über die PCI-Express-Schnittstelle angebunden. PCI-Express wurde entwickelt um die alten PCI- und AGP-Schnittstellen zu abzulösen. Im Unterschied zu PCI ist PCI-Express kein geteiltes Bus-System mit paralleler Datenübertragung sondern bietet serielle Punkt-zu-Punkt-Verbindungen, den *Lanes*. Durch PCI-Express 2.0 konnte die Datenübertragungsrate von einer im Vergleich zur Vorgängerversion auf 500MB/s in beide Richtungen verdoppelt werden.

Für eine zusätzliche Beschleunigung kann ein Gerät auch mehrere Lanes gleichzeitig benutzen. Für den Grafikkarten-Steckplatz lassen sich so beispielsweise 16 Lanes verwenden, wodurch die Bandbreite erhöht wird (Kic07).

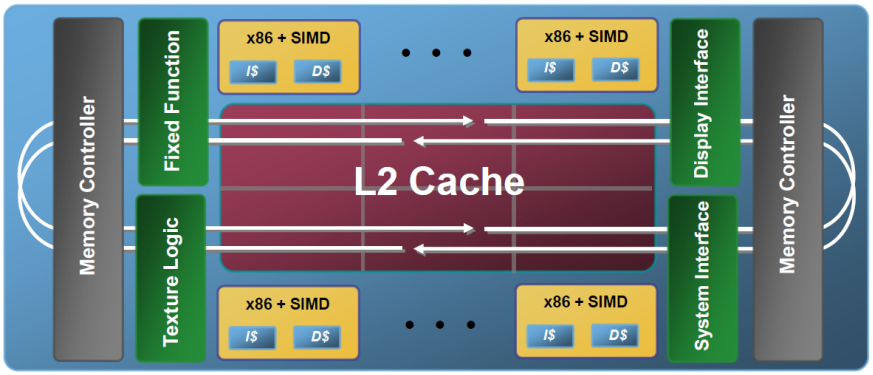

Abbildung 6: Die einzelnen Recheneinheiten des Larrabee-Chip unterstützen x86 und sind durch ein Ringsystem miteinander verbunden. *Quelle: (wik10)*

4 Hardware-Architekturen

Der Grafikkartenmarkt lässt sich im Wesentlichen in drei Teile aufteilen. Die Hälfte des Marktes gehören Intels on-board-Grafikchips. Die andere Hälfte ist widerum zu gleichen unter nVidia und ATI aufgeteilt. ATI wurde 2006 von AMD aufgekauft und seither auch Produkte unter dem Namen von AMD vertreibt. Sowohl nVidia als auch ATI stellen Grafikkarten für anspruchsvollere 3D-Anwendungen und neuerdings auch GPGPU-Anwendungen her. nVidia GeForce und ATI Radeon sind ursprünglich für den nicht-professionellen Bereich vorgesehene Grafikchips, die hauptsächlich für Computerspiele ausgelegt waren. Dadurch, dass diese frei programmierbar wurden, war GPGPU auf nahezu jedem Heimcomputer verfügbar.

Mit der gestiegenen Verbreitung bieten sowohl ATI als auch nVidia mit Firestream bzw. Tesla Chips an, die nicht primär auf Grafikberechnung ausgelegt sind. In den neusten Generationen erreichen die Chips mehr als 1000 GFlops bei einfacher Genauigkeit und bis zu 250 GFlops bei doppelter Genauigkeit. Die ATI RV770 Chips können dabei mehr als 16000 Threads parallel verarbeiten. Beide Systeme sind auch mit dem IEEE754-Standard kompatibel.

Neuere und experimentellere Ansätze wie Cell und Larrabee hingegen versuchen, die beiden Welten CPU und GPU zu vereinigen. Die Ansätze sind dabei vollkommen unterschiedlich. Der CELL-Prozessor ist aus ALUs mit vierfachem SIMD bestückt, die zusammenarbeiten und dabei bis zu 200 GFlops bei einfacher Genauigkeit erreichen. Intels Larrabee ist dagegen auch dem weitverbreiteten x86-Befehlsatz kompatibel, so dass auf dem Grafikchip eine Vielzahl von Anwendungen lauffähig wäre. Diese Chips stellen völlig neue Konzepte dar und funktionieren völlig anders als die Grafikkarten von ATI oder nVidia.

In Abbildung 5 ist ein schematischer Aufbau des Larrabee-Prozessors dargestellt. Die einzelnen Recheneinheiten unterstützen x86 und sind durch ein Rechensystem miteinander verbunden.

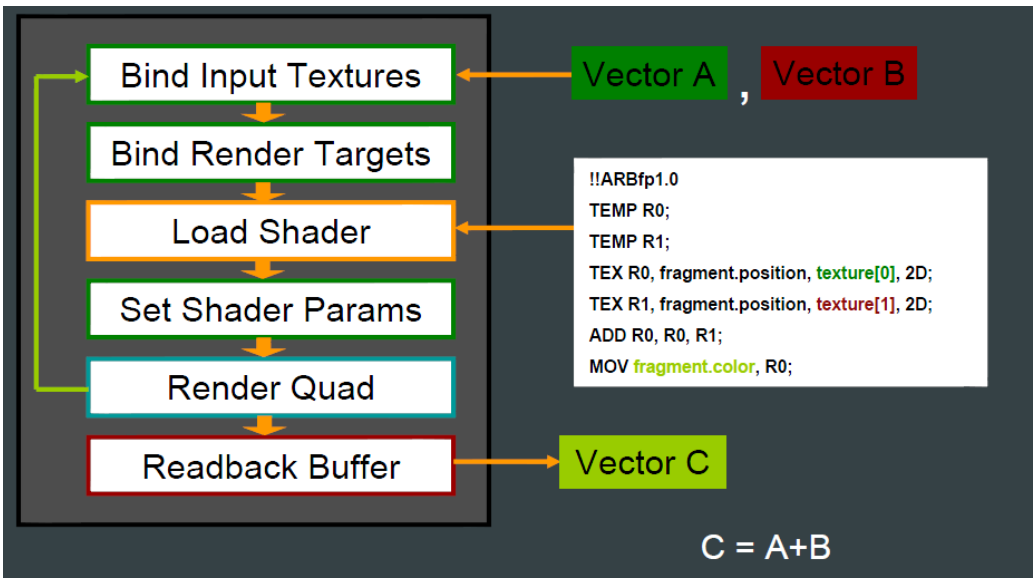

Abbildung 7: Zur Ausführung einer GPGPU-Anwendung notwendige Schritte. *Quelle: (Hou07)*

5 Programmierung

Es ist nicht möglich, Programme direkt auf der GPU ausführbar zu machen. Man musst zunächst eine auf der CPU ausgeführte Host-Anwendung schreiben und starten, die die aufzurufenden Befehle der Grafikkarte tätigt. Soll eine Berechnung mit Hilfe eines Fragment-Shaders durchgeführt werden, muss zunächst ein Fragment beispielsweise durch Rendern eines Rechtecks erzeugt werden. Die Anweisungen zum Rendern des Rechtecks mit Fragment-Shadern und den Shader-Code bekommt die Grafikkarte von der Rahmenanwendung zur Verfügung gestellt. Das Ergebnis muss dann von der Rahmenanwendung weiterverarbeitet oder gespeichert werden.

Der Ablauf einer GPGPU-Anwendung lässt sich also folgendermaßen zusammenfassen:

1. Die Host-Anwendung wird auf der CPU gestartet.
2. Initialisierung der Umgebung, zum Beispiel die Aktivierung von „Render-to-texture“, falls man das Resultat nicht schon direkt angezeigt bekommen möchte, sondern die Daten an die CPU übergeben möchte.
3. Erstellen der Texturen, die die zu verarbeitenden Daten enthalten.
4. Ubergabe des Fragment-Programms an den Grafiktreiber, der es in Maschinencode übersetzt.
5. Das Rendern eines Vollbildrechtecks starten.
6. Falls „render-to-texture“ aktiviert wurde, kann man die Output-Textur wieder zur CPU zurücktransferieren.

5.1 Befehlsätze und Datentypen

CPUs besitzen einen größeren Befehlssatz als GPUs. GPUs besitzen dafür mehr Einheiten für mathematische Berechnungen. Daher sind GPUs besonders für Aufgaben geeignet, die viele mathematische Berechnungen erfordern. Im Gegensatz zu den CPUs, bei denen der x86 im Desktop-Bereich der Quasistandard ist, gibt es einen solchen bei GPUs nicht, d.h. die Befehlssätze sind herstellerspezifisch.

Als weitere Unterstützung von Parallelismus beherrschen die Prozessoren SIMD-Instruktionen. SIMD steht für **S**ingle **I**nstruction **M**ultiple **D**ata. Die Anwendung von SIMD-Befehlen ist bei Grafikkarten besonders sinnvoll, da nahezu jeder Datentyp ein Vektor ist (entweder 2, 3 oder 4-dimensional). Dazu gehören die Vertices, Farben, Normalenvektoren und Texturkoordinaten. Die Grafikchips können mit einem Befehl auf 4-dimensionalen Vektoren arbeiten, wodurch sich die Rechengeschwindigkeit nahezu vervierfacht, denn es muss nicht jede Komponente des Vektors einzeln betrachtet werden. Die SIMD-Instruktionen können auch für viele andere Berechnungszwecke eingesetzt werden, allerdings ist die Programmierung nicht immer intuitiv. SIMD ist auf aktuellen x86-CPUs, die die Befehlssatzerweiterung SSE unterstützen, jedoch auch möglich.

Anfangs unterstützten die GPUs nur 16- oder 24-bit Gleitkommazahlen, später wurden auch IEEE754-kompatible 32- und 64-bit Gleitkommazahlen unterstützt. Die meisten aktuellens GPU unterstützen außerdem auch Integer- und bitweise Operationen.

Die gebräuchlichste Datenform für GPUs ist ein 2D-Gitter, denn es ist für das Verarbeitungsmodell der GPU am geeignetesten. 2D-Gitter kommen meist in Form von Matrizen, Texturen und Bilddaten vor. Die Texturpuffer werden bei der GPGPU-Programmierung als Speicher benutzt, die Texur-Lookups stellen dann sozusagen Speicherleseoperationen dar. Bestimmte Leseoperationen können von der GPU auch automatisch durchgeführt werden.

Vor dem Unified Shader Model konnten die GPUs nur statische Arrays verarbeiten. Später konnten alle GPUs, die das Unified Shader Model unterstützen, können auch dynamische Arrays sowie Listen und Bäume darstellen.

5.2 Programmiertechniken

Es existieren eine Vielzahl von Programmierkonzepten und -techniken für GPUs, um die verfügbare Rechenleistung möglichst optimal auszunutzen.

Die „Map“-Operation führt für jedes Element im Stream das Fragment-Programm aus. Ein Beispiel dafür wäre, jeden Wert mit einem bestimmten Faktor zu skalieren. Für jedes Element wird dann das Fragment-Programm ausgeführt und das Resultat, welches die gleiche Größe wie die Ausgangsdaten besitzt, in den Ausgabepuffer geschrieben.

Es müssen auch nicht alle Elemente, die sich im Datenstrom befinden, bei der Berechnung berücksichtigt werden. Durch bestimmte Kriterien kann man die Elemente, die nicht betrachtet werden sollen, aus dem Datenstrom ausfiltern.

Einen Sortieralgorithmus auf einer GPU zu implementieren ist nicht ganz so trivial wie auf einer CPU. Durch die vielen Programmverzweigungen und Sprünge können die GPUs hier nicht optimal arbeiten. Die am meisten genutzt Implementierung auf GPU ist das Benutzen von Sortiernetzwerken.

Für Suchoperationen ist die GPU nicht speziell optimiert. Nach einem einzelnen Element kann meistens schneller von der CPU ausgeführt werden, jedoch ist die GPU bei mehreren parallelen Suchvorgängen wesentlich schneller.

Scatter ist ein für GPU ein großes Performance-Problem. Scatter bezeichnet die Operation, Daten von einem Prozess an alle anderen laufenden Prozesse zu senden. Da die Rechenkerne untereinander nicht direkt kommunizieren können und für die Kommunikation erst in den Speicher geschrieben und gelesen werden muss, entsteht so ein großer Performance-Verlust.

5.3 nVidia CUDA

Grafikkarten, die nVidia CUDA unterstützen, enthalten mehrere hundert SIMD-Stream-Prozessoren. Das neue an den CUDA-GPUs ist, dass sie den globalen Speicher für beliebige Lese- und Schreiboperationen nutzen können. Damit sind sie in diesem Punkt fast so flexibel wie eine CPU. Um häufigen

CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

Abbildung 8: Im Gegensatz zum CPU-Programm wird im CUDA-Programm nur der Inhalt der Schleife definiert, die GPU macht die Aufteilung auf die einzelnen Rechenkerne dann selbständig. *Quelle: (Hou05)*

Zugriff auf den globalen Speicher zu vermeiden, besitzt jeder Prozessor auf dem Chip einen eigenen kleinen Zwischenspeicher, auf den mit 4 Clock-Zyklen zugegriffen werden kann. Ein Zugriff auf den globalen Speicher benötigt hingegen 400-600 Clock-Zyklen.

CUDA stellt den Programmierern eine C-ähnliche Entwicklungsumgebung zur Verfügung und benutzt einen speziellen C-Compiler, um die Programme zu kompilieren. Dadurch werden die Shader-Sprachen durch C (mit ein paar zusätzlichen CUDA-spezifischen erweiterten Bibliotheken) ersetzt. Dadurch müssen Programme nicht mehr notwendigerweise an die speziellen Eigenheiten der Shader-Sprachen angepasst werden, wodurch die Entwicklung von GPGPU-Programmen vereinfacht und flexibler wird.

Bei einem CPU-Programm definiert man normalerweise die Schleifen, die durchlaufen werden sollen. Bei einem CUDA-Programm wird nur der Inhalt der Schleife definiert. Durch Makros generiert der Kompiler dann automatisch die entsprechenden Befehle, um die Berechnungen auf die vielen Rechenkerne aufzuteilen.

Im Beispielprogramm soll eine Addition von zwei Matrizen durchgeführt werden. Beim CPU-Programm geschieht dies auf gewohntem Wege durch zwei innereinander verschachtelte Schleifen. Beim CUDA-Programm werden Makros definiert, die festlegen, wie die Matrix aufgeteilt werden soll. Die Matrix wird dabei in $n \times n$ große Blöcke aufgeteilt und jedem Rechenkern wird dann ein solcher Block zugewiesen.

6 Diskussion und Ausblick

GPU können als erste für den Endverbraucher erhältlichen Parallelprozessoren betrachtet werden. Durch die hohe Verfügbarkeit und den geringen Preis hat die Verbreitung von GPGPU in den letzten Jahren rasant zugenommen. Auch durch bessere Hardware und ausgereifte Softwarebibliothen haben sich GPUs für die mathematische Problemlösung durch massiv-parallele Rechenwerke etabliert.

Es sind jedoch noch viele Verbesserungen möglich. So gestaltet sich die Programmierung von GPUs immer noch umständlicher als die von CPUs. Der Zugriff auf die Grafikkarte sollte nicht zwingend über OpenGL oder DirectX erfolgen, denn diese sind Grafikschnittstellen und eigentlich nicht für allgemeinere Berechnungen ausgelegt. Außerdem ist das Verwenden von Grafikbefehlen für allgemeine Zwecke oftmals unhandlich. Auch ein direkter Hardware-Zugriff ist ebenfalls wünschenswert.

Obwohl die PCI-Express bereits eine sehr hohe Bandbreite bietet, stellt die Verbindung zwischen CPU und GPU immer noch ein Flaschenhals dar. Die dritte Version von PCI-Express soll die aktuell verfügbare Bandbreite verdoppeln können. Zusätzlich werden auch bald CPUs auf dem Markt erhältlich sein, die einen Grafikprozessor auf dem Chip besitzen. Dadurch wird dieser Flaschenhals ebenfalls umgangen.

Zwar existieren hier bereits Standards wie OpenCL, jedoch ist hier keine Integration in bestehende Hochsprachen wie C/C++, Python oder Java gegeben. Für die Integration in Hochsprachen werden dann außerdem entsprechende Compiler benötigt. Eine Art „GCC“ für GPU ist also wünschenswert. Außerdem sollte der Datenaustausch zwischen CPU und GPU verbessert werden.

Gelingen diese Verbesserungen, könnte GPGPU seinen Siegzug gegenüber den CPU-basierten Lösungen, die zur Zeit mehrheitlich bei industriellen und wissenschaftlichen Anwendungen verwendet werden, antreten.

Außerdem nähern sich CPU und GPU immer hardwaretechnisch immer weiter an, sind aber von einer Kompabilität noch weit entfernt. Inwieweit sich die Vereinigungen Larrabee und Cell der beiden Designphilosophien CPU und GPU in Zukunft entwickeln werden, bleibt also abzuwarten.

Literatur

[Buc05] BUCK, Ian: General Purpose Computation on Graphics Hardware / nVidia Corp. 2005. – Forschungsbericht

[DMZ09] DIMITROV, Martin ; MANTOR, Mike ; ZHOU, Huiyang: Understanding Software Approaches for GPGPU Reliability / University of Central Florida, Orlando. 2009. – Forschungsbericht

[Hou05] HOUSTON, Mike: General Purpose Computation on Graphics Processors (GPGPU) / Stanford University. 2005. – Forschungsbericht

[Hou07] HOUSTON, Mike: Advanced Programming (GPGPU) / Stanford University. 2007. – Forschungsbericht

[Kic07] KICHERER, Mario: GPGPU: Architektur, Programmierung und Anwendungen. 2007. – Forschungsbericht

[Kir06] KIRCHHOFF, Marc: GPGPU Basiskonzepte. 2006. – Forschungsbericht

[LH07] LUEBKE, David ; HUMPHREYS, Greg: How GPUs work. In: *Computer (IEEE)* (2007)

[LMW07] LIU, Weiguo ; MÜLLER-WITTIG, Wolfgang: Performance Predictions for General-Purpose Computation on GPUs / Nanyang Technological University. 2007. – Forschungsbericht

[Owe06] OWENS, John: What's New with GPGPU? / University of California, Davis. 2006. – Forschungsbericht

[Per06] PERUMALLA, Kalyan S.: Parallel and distributed simulation: Traditional techniques and recend advances / Oak Ridge National Laboratory. 2006. – Forschungsbericht

[wik10] *en.wikipedia.org*. 2010

[Zib10] ZIBULA, Alexander: General Purpose Computation on Graphics Processing Units (GPGPU) using CUDA / Westfälische Wilhelms-Universität Münster. 2010. – Forschungsbericht