# Spezialarchitekturen I (GPGPU: Architektur, Programmierung und Anwendungen)

Mario Kicherer

***Zusammenfassung*—Die vorliegende Ausarbeitung beschäftigt sich mit dem Aufbau, der Programmierung und den resultierenden Anwendungsmöglichkeiten von GPUs in Bezug auf deren Nutzung zur Lösung von allgemeinen mathematischen Problemen.**

***Index Terms*—Reconfigurable Computing, GPGPU.**

## I. Einleitung

In den vergangenen Jahren verstärkte sich der Trend die handelsüblichen GPUs für Heim-PCs neben ihrer Hauptaufgabe (der Grafikdarstellung) auch für allgemeine mathematische Berechnungen zu verwenden. Dieser Anwendungsbereich wird generell unter dem Begriff „General-purpose computation on GPU“ (kurz GPGPU) zusammengefasst. Die GPUs haben sich in ihrer Geschichte von im Prinzip einfachen Digital-Analog-Wandlern in leistungsfähige, programmierbare Prozessoren entwickelt. Inzwischen haben sie mit ihrem spezialisierten Ansatz bereits die heutigen CPUs in ihrer Rechenleistung überholt (siehe Tabelle I, Quelle [Luebke SC06]). Diesen Vorsprung haben sie aber hauptsächlich der zielgerichteten Optimierung zur Verarbeitung für Grafikdaten zu verdanken. Während CPUs möglichst universell ausgelegt sind, liegt bei GPUs der Fokus auf schneller paralleler Abarbeitung von Daten in einem speziellen Format, dadurch ist es möglich zusätzliche Chipfläche durch das Entfernen von z.B. Steuerungs- und Cachebausteinen für zusätzliche Rechenwerke zu erhalten. Algorithmen die auf einer GPU ausgeführt werden, sollten also eine hohe arithmetische Dichte aufweisen (als arithmetische Dichte bezeichnet man den Quotienten aus ausgeführten Rechen- pro ausgeführter Lese- bzw. Schreiboperation). Aufgrund dieser Einschränkungen bleibt die GPU aber nach wie vor eher ein Coprozessor für die CPU als eine eigenständige Recheneinheit.

Tabelle I
Vergleich eines 3.0 GHz Intel Core2Duo und einer Nvidia GeForce 8800 GTX

| Vergleich | 3.0 GHz Core2Duo | 8800 GTX |
|---|---|---|
| Rechenleistung (GFlops) | 48 | 330 |
| Speicheranbindung (GB/s) | 21 | 55.2 |
| Preis (Dollar) | 874 | 600 |

In Kapitel 2 wird dargestellt wie GPUs generell aufgebaut sind und funktionieren, welche Strukturen für das GPGPU-Konzept genutzt werden können und worin sich der GPGPU-Ansatz von klassischen massiv-parallelen Lösungen unterscheidet. Unter „GPGPU Programmiertechniken“ werden bestehende Programmiersprachen und Bibliotheken vorgestellt. Das vorletzte Kapitel beschäftigt sich dann mit Anwendungsbeispielen in denen diese GPGPU-Lösungen verwendet werden, bevor im letzten Teil noch eine kurze Zusammenfassung und ein Ausblick folgen.

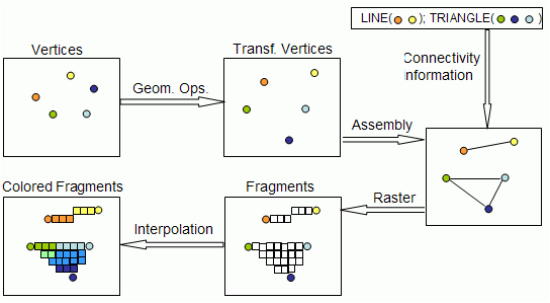


Abbildung 1. Der Ablauf der einzelnen Pipelinestufen exemplarisch dargestellt

## II. Technische Grundlagen

### A. Renderpipeline

Bis ein Bild fertig dargestellt werden kann, müssen mehrere Stufen einer Pipeline durchlaufen werden. Die Stufen einer Pipeline lassen sich grob in 2 Teilbereiche aufteilen: „Geometry processing“ und „Rasterization“. Die beiden Bereiche unterscheiden sich in der Interpretation der Daten. In der „Geometry processing“-Phase wird mit Vektoren gerechnet, während in der „Rasterization“-Phase dann mit einer Art von Pixeln gearbeitet wird.

Eine vereinfachte Beschreibung der Stufen einer gängigen Grafikpipeline (in Abbildung 1 auch exemplarisch dargestellt):

*1) Input:* Die Eingabe von Daten erfolgt meist über eine Grafikbibliothek wie OpenGL oder DirectX. Die grafischen Objekte werden dabei in ihren Einzelteilen an die verwendete Bibliothek übergeben.

*2) Transform and Light:* Die Objekte aus der Eingabephase können in relativen Koordinatensystemen übergeben werden. In der Transformationsphase werden alle relativen in absolute Koordinaten im globalen System umgerechnet und dann aus Sicht der Kamera berechnet. Man spricht auch von einer Transformation von der Weltansicht in die Bildansicht. Ausserdem wird hier noch die Beleuchtung der Eckpunkte im Bezug auf die Lichtquellen bestimmt. Während des gesamten Vorgangs werden alle Eckpunkte als unabhängig betrachtet.

*3) Assemble primitives:* In dieser Phase werden die transformierten Eckpunkte mittels eines Teils der Daten aus der

Eingabephase wieder zu einfachen Objekten zusammengesetzt.

*4) Rasterization:* Durch die Rasterisierung ändert sich nun deren Repräsentation. Die durch Vektoren beschriebenen Objekte werden in sogenannte Fragmente umgewandelt. Als Fragmente bezeichnet man in diesem Fall die Kombination aus Pixeln und zusätzlichen Eigenschaften, wie z.B. Farbe und Texturposition. Man kann diesen Schritt auch als Transformation von 3-dimensionalen in 2-dimensionale Objekte verstehen.

*5) Shading:* Aufgabe dieser Phase ist es nun aus den Fragmenten und einer oder mehrerer Texturen einen neuen Farbwert zu berechnen.

*6) Output:* Das fertige Bild wird, je nach gewähltem Ausgang, in elektrische Signale umgewandelt und an ein Darstellungsgerät gesandt.

### *B. Für GPGPU nutzbare Strukturen*

Die eigentlichen Hardwarestrukturen, die für die GPGPU-Funktionalität genutzt werden können, sind sogenannte Streamprozessoren. Die Programme, die auf ihnen ausgeführt werden, bezeichnet man als „Shader“ oder auch „Kernel“. Der Begriff „Shader“ wird allerdings auch oft für den Prozessor selbst verwendet.

In den gängigen GPUs sind die folgenden nutzbaren Strukturen vorhanden:

*1) Vertex processor:* Als Vertex-Prozessor bezeichnet man eine aus der „Transform and Light“-Einheit entstandenen Bauteil in der Pipeline. Mit dem Vertex-Prozessor kann man mehrere Eigenschaften von Objekten verändern, zum Beispiel den Positions- oder Normalenvektor wie auch die Texturkoordinaten. Vertex-Prozessoren sind jedoch auch in ihrer Funktionalität limitiert, so können sie zum Beispiel nur vorhandene Daten bearbeiten (sie können dieser Menge nichts hinzufügen noch entfernen) und nur eine begrenzte Anzahl von Instruktionen je Kernel abarbeiten (je nach Shader-Version, siehe Tabelle II).

Tabelle II
MAXIMALE ANZAHL DER INSTRUKTIONEN IN BEZUG AUF DIE VERTEX-SHADER-VERSION

| Shader-Version | Anzahl Instruktionen |
|---|---|
| 1.1 | 128 |
| 2.0 | 256 |
| 3.0 | 512 |
| 4 | Unbegrenzt |

Aus GPGPU-Sicht sind folgende Informationen für den Vertex-Prozessor interessant:

- Voll programmierbar (SIMD wie auch MIMD)
- Die Position der aktuellen Dateneinheit im Speicher kann verändert werden („Scatter“ Prinzip).
- Es können keine anderen Dateneinheiten außer der aktuellen gelesen werden.

*2) Geometry processor:* Der Geometry-Prozessor entspricht der „Assemble primitives“-Einheit der Pipeline und erhält als Ausgangsbasis die Daten des Vertex-Prozessor. Diese kann er jedoch flexibler verarbeiten und ist nicht so eingeschränkt wie sein Vorgänger in der Pipeline. Mittels dem Geometry-Prozessor ist es auch möglich, Eckpunkte zu einem Objekt hinzuzufügen oder zu entfernen.

*3) Fragment processor:* Zu den Aufgaben des Fragment-Prozessors gehört unter anderem die Berechnung der Farbe eines Pixels bezüglich der ihm zugewiesenen Texturen und entstand aus der Shading-Einheit. Der Fragment-Prozessor wird im Allgemeinen auch als „Pixel processor“ oder „Pixel Shader“ bezeichnet.
Tabelle III gibt eine Übersicht über die maximale Anzahl an Instruktionen, die ein Kernel umfassen darf, im Bezug auf die Shader-Version. Wichtige Eigenschaften aus Sicht einer GPGPU-Anwendung sind:

- Volle Programmierbarkeit (nur SIMD)
- Es kann von beliebigen Speicheradressen gelesen werden („Gather“-Prinzip).
- Die Speicherstelle zur Ausgabe kann nicht frei gewählt werden.

Tabelle III
MAXIMALE ANZAHL DER INSTRUKTIONEN IN BEZUG AUF DIE PIXEL-SHADER-VERSION

| Shader-Version | Anzahl Instruktionen |
|---|---|
| 1.1 | 8 |
| 1.4 | 14 |
| 2.0 | 64 arithmetisch, 32 Texturen |
| 3.0 | 512-32768 |
| 4.0 | unbegrenzt |

Mit der neuen Geforce-8-Serie führt NVIDIA die sogenannten „Unified Shaders“ im PC-Bereich ein. Diese Unified-Shader lassen sich zur Laufzeit entweder als Vertex-, Geometry- oder Pixel-Prozessoren umkonfigurieren, das bedeutet, man kann die jeweilige Anzahl optimal an die momentanen Anforderungen anpassen. Als Beispiel dient eine Szene mit Himmel und einer Wasseroberfläche im Vergleich zu einer Szene mit vielen Spielfiguren und Gegenständen. In der ersten Szene werden hauptsächlich Pixel-Shader benötigt, da wenige Polygone berechnet werden müssen, aber viel Zeit für Spiegelungen und Transparenz verbraucht wird. Im Gegensatz dazu wird in der zweiten Szene viel Rechenzeit für die Positionsberechnung der detaillierten Figuren und Gegenständen benötigt (siehe Abbildung 2 für ein grafisches Beispiel).
NVIDIA fasst jeweils 16 dieser Unified-Shader-Einheiten zu einem so genannten Shadercluster zusammen. In der 8800-GTX-GPU sind zum Beispiel acht solche Shadercluster vorhanden. Auch ATI führte diese Technik in ihrer Radeon-HD-2000-Serie ein. Hier werden jedoch jeweils fünf Shader zu einer Shadereinheit zusammengefasst und acht solche Shadereinheiten wiederum bilden ein Shadercluster. Tabelle IV zeigt eine kleine Übersicht der Shaderanzahl in den aktuellen Grafikchips.

### *C. Kommunikation zwischen CPU und GPU*

Nach aktuellem Stand der Technik wird die GPU über PCI-Express (PCIe) angebunden. PCI-Express wurde entwickelt um die alten PCI- und AGP-Schnittstellen zu ersetzen. Eine erste Version von PCI-Express wurde unter dem Namen 3GIO („Third generation I/O“) von der „Arapahoe Working Group“

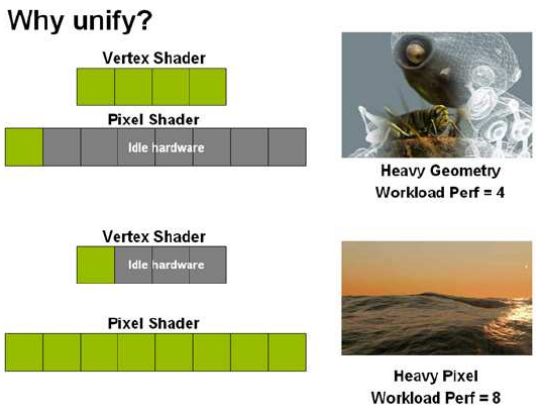


Abbildung 2. Zwei Beispielszenen mit gegensätzlichen Shaderanforderungen

(AWG) entwickelt. Diese wurde dann von der „PCI Special Interest Group“ zur heutigen bekannten Version erweitert. Im Unterschied zu PCI ist PCI-Express kein geteiltes Bus-System mit paralleler Datenübertragung sondern bietet serielle Punkt-zu-Punkt-Verbindungen. Die Datenübertragung erfolgt hier über sogenannte Lanes, die aus jeweils 2 Leitungspaaren zum Senden und Empfangen bestehen. Durch die Veröffentlichung von PCIe v2.0 konnte die Datenübertragungsrate von einer Lane im Vergleich zur Vorgängerversion auf 500MB/s in beide Richtungen verdoppelt werden. Um nun die Kommunikation noch zusätzlich zu beschleunigen ist es für ein Gerät möglich, mehrere solche Lanes gleichzeitig zu benutzen. Die Anzahl der verwendeten Lanes erkennt man an der Schnittstellenangabe eines solchen Gerätes, z.B. PCIe x16, steht für 16 benutzbare Lanes - diese Version wird speziell auch „PCI-Express for Graphics“ (PEG) genannt und hauptsächlich für Grafikkarten verwendet.

### *D. GPGPU im Vergleich zu anderen parallelen Beschleunigern*

Im Vergleich zu traditionellen massiv-parallelen Beschleunigern (z.B. von ClearSpeed) bieten die heutigen GPGPU Lösungen meist nur eingeschränkte Genauigkeit bei Fließkommaberechnungen. Konkret bieten die aktuellen GPUs mit DirectX-10 Unterstützung von AMD und NVIDIA nur 32-bit Fließkomma- und Ganzzahlendatentypen, während die traditionellen Lösungen auch 64-bit-Datentypen bereitstellen. Mit DirectX-10.1 und dem damit verbundenen Shader-Model-4.1 wird jedoch auch für die kommenden Grafikkarten 64-bit-Unterstützung Pflicht, um die Kompatibilitätszertifikation zu erhalten.

Tabelle IV
SHADERANZAHL DER AKTUELLEN GRAFIKCHIPS

| Hersteller | Grafikchip | Anzahl der Unified-Shader |
|---|---|---|
| NVIDIA | G80 | 128 |
| NVIDIA | G84 | 32 |
| NVIDIA | G86 | 16 |
| NVIDIA | G92 | 128 |
| ATI | R600 | 320 |
| ATI | RV610 | 40 |
| ATI | RV630 | 120 |
| ATI | RV670 | 320 |

AMD bietet durch den Zukauf von ATI inzwischen eine GPU mit 64-bit-Unterstützung an. Diese GPUs basieren auf dem aktuellen Kern RV670, der auch in den aktuellen High-End-Grafikkarten der HD-3800er-Serie und in der speziell als GPGPU-Lösung angebotenen FireStream-Karte arbeitet. AMD ist damit der erste Hersteller der 64-bit auch in Grafikkarten anbietet.
Ein anderer aktueller Streamprozessor ist die sogenannte „Cell Broadband Engine Architecture“. Entwickelt wurde sie gemeinschaftlich von Sony, Toshiba und IBM und ist heute wohl am bekanntesten als das Herzstück der Sony Playstation 3. Man kann den Cell-Prozessor als Bindeglied zwischen den konventionellen CPUs und den speziell angepassten GPUs sehen, da sie aus einem Verbund eines PowerPC-Kern, dem sogenannten „Power Processing Element“ (PPE), und acht SIMD Streamprozessoren, den sogenannten „Synergistic Processing Elements“ (SPE) besteht. Der große Unterschied dieser SPEs zu den Shader-Einheiten von GPUs ist die hohe Flexibilität. Während die Shader-Einheiten an die Pipeline gebunden sind und nur einen eingeschränkten Befehlssatz besitzen, können die SPEs relativ frei programmiert werden, was ihnen eine Überlegenheit in vielen Bereichen beschert.

## III. GPGPU PROGRAMMIERTECHNIKEN

GPUs wurden konstruiert um Grafikdaten zu verarbeiten. Durch diese Spezialisierung unterscheidet sich auch das Programmierkonzept deutlich von dem gewohnten Vorgehen für die CPU. „Normaler“ Code für die CPU ist daher nicht einfach portierbar, sondern muss im Prinzip neu geschrieben werden, um den Geschwindigkeitsvorteil der GPUs auszunutzen. Aber auch die Kompatibilität zwischen den Herstellern ist nicht gegeben. Daher haben sich neben den gängigen Sprachen zur Shaderprogrammierung im Grafikbereich wie Cg, HLSL oder der „OpenGL shading language“ spezielle Lösungen entwickelt, die konzipiert wurden um das Programmieren der GPU für allgemeine Berechnungen zu vereinheitlichen und zu vereinfachen.

### *A. Genereller Ablauf einer GPGPU-Anwendung*

Da GPGPU-Berechnungen nach wie vor von der CPU initiiert werden, muss zuerst eine sogenannte Host-Anwendung auf dieser gestartet werden, die dann die GPU mit den nötigen Daten versorgt. Der Ablauf einer GPGPU-Anwendung wird im folgenden grob beschrieben:

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 zurückführen möchte.
3) Erstellen der Texturen, die die zu verarbeitenden Daten enthalten.
4) Übergabe des Kernels an den Grafiktreiber, der diesen „on-the-fly“ in Maschinencode für die GPU übersetzt.
5) Das Rendern eines „Vollbildrechtecks“ starten.
6) Falls „render-to-texture“ aktiviert wurde, die resultierende Textur wieder zur CPU transferieren.

### B. Einschränkungen

Die Standardbibliotheken zur Shaderprogrammierung sind für Grafikoperationen optimiert. Dies birgt für die universelle Nutzung einige Einschränkungen. Der Fragment-Prozessor arbeitet z.B. hauptsächlich mit Texturen. Diese Texturen sind allerdings in der Größe und den Dimensionen wie auch in der Anzahl der verwendbaren Datentypen beschränkt. Ein weiterer Nachteil ist der eingeschränkte Befehlssatz bezüglich gängiger Operationen, die man von den CPUs kennt. Man sollte die Streamprozessoren auch nicht als wirklich eigenständig betrachten, denn sie unterliegen immernoch den Regeln der Pipeline, das heisst, sie haben festgelegte Ein- und Ausgabeparameter.

### C. Bibliotheken

*1) Lib Sh:* Sh ist eine Metaprogrammiersprache, die in C++ eingebettet werden kann. Sie entstand als Forschungsprojekt an der Universität von Waterloo. Inzwischen wird die quelloffene Version nicht mehr weiterentwickelt. Eine kommerzielle Version wird von Rapidmind (s.u.) vertrieben.

*2) MS Accelerator Project:* Das Accelerator Projekt besteht aus einer Bibliothek aufbauend auf .NET und ist für den nicht-kommerziellen Gebrauch kostenlos. Der Programmcode wird „on-the-fly“ entweder für den Pixel-Shader oder die CPU kompiliert. Sie unterscheidet explizit zwischen normalen Arrays und speziellen „Data-parallel arrays“, die für die Bibliotheksfunktionen benötigt werden.

*3) Brook:* Brook ist eine sogenannte „stream programming language“, die an der Universität von Stanford entwickelt wurde und unter einer Open-source-Lizenz vorliegt. Sie basiert auf der Sprache C mit einigen Erweiterungen und besteht im Wesentlichen aus 2 Teilen: dem Compiler brcc und der Laufzeitbibliothek BRT („Brook Runtime“). Bezüglich der Programmierung bietet Brook im Vergleich zu den existierenden Shaderprogrammiersprachen folgende Abstraktionen:

- Speicherorganisation über Streams
- parallel ausführbare Operationen werden durch den Aufruf einer speziellen Funktion (genannt Kernel) gestartet.

Bei Brook wurde Wert auf Plattformunabhängigkeit gelegt, daher kann OpenGL, DirectX (jeweils ab einer bestimmten Version) oder AMD's CTM als Backend benutzt werden. Es ist aber auch möglich, Brook-Code in „normalen“ CPU-Code zu übersetzen - je nachdem welche Hardwareressourcen auf der Zielplattform verfügbar sind.

*4) CTM:* CTM („Close to the metal“) ist eine Entwicklung von ATI (inzwischen aufgekauft von AMD), die im Februar 2007 veröffentlicht wurde. Sämtliche Radeon-Grafikkarten und Firestream-Streamprozessoren werden von CTM unterstützt. Inzwischen hat CTM sich in 2 Teile entwickelt, einem Geräte-spezifischen „Hardware abstraction layer“ (HAL) und einem universellen „Compute abstraction layer“ (CAL), mit dem es auch möglich ist unterschiedliche Streamprozessoren einheitlich anzusprechen.

*5) CUDA:* CUDA („compute unified device architecture“) ist die GPGPU-Lösung von NVIDIA und wurde im Februar 2007 der Öffentlichkeit vorgestellt. Die Algorithmen werden hier ebenfalls mit kleinen Erweiterungen in der Sprache C verfasst. CUDA ist allerdings beschränkt auf die Geforce-8-Serie.
CUDA abstrahiert die vorhandenen Streamprozessoren in einem System. Dazu fasst sie Verbände dieser Prozessoren in einer oder mehreren Multiprozessorgruppen zusammen. Jeder dieser Streamprozessoren unterstützt dabei 32-bit-skalare Datenpfade und Fließkommazahlen mit einfacher Genauigkeit nach IEEE 754 und kann theoretisch bis zu 96 Threads verwalten, was bei den aktuellen Grafikchips mit 128 Streamprozessoren zu einer maximalen Anzahl von 12288 Threads führt. Das Limit an Threads wird praktisch durch die Anzahl der benötigten Register pro Thread bestimmt. Um die maximale Anzahl an Threads zu erreichen, darf jeder von ihnen nicht mehr als 10 Register nutzen. Da den meisten Threads vom Compiler allerdings zwischen 20 und 32 Register zugewiesen werden, schwankt die maximale Anzahl bei den meisten Anwendungen zwischen 4000 und 6500 Threads. Die CUDA-Entwickler empfehlen eine durchschnittliche Anzahl von 5000, um die beste Performance zu erreichen.
Die Threads eines Kernels werden in bis zu 3-dimensionalen Blöcken organisiert. Die Threads eines Blocks teilen sich einen Speicherbereich. Die einzelnen Blöcke sind dagegen komplett unabhängig und können auch nicht miteinander kommunizieren. Um dieses Organisationskonzept nun auszunutzen, sollten während der Entwicklung folgende Richtlinien berücksichtigt werden:
Um die Abarbeitung von großen Datenmengen zu optimieren, sollten diese so aufgeteilt werden, dass die einzelnen Teile möglichst effizient von einem Threadblock verarbeitet werden können, d.h. unter anderem, dass der vorhandene gemeinsame Speicher gut ausgenutzt wird. Es sollte allerdings dafür Sorge getragen werden, dass der gemeinsame Speicher ausreicht, da er im Vergleich zu den Alternativen deutlich schneller ist.
CUDA bietet neben den Standardbibliotheken auch vorgefertigte Sammlungen von speziellen Funktionen, wie

- CUBLAS, eine Implementierung der BLAS-API. Sie vereinfacht die Portierung von bereits bestehenden Programmen. BLAS steht für „Basic Linear Algebra Subprograms“ und bietet eine einheitliche Schnittstelle für Programme zur Lösungen von Problemen der Linearen Algebra.
- CUFFT, eine optimierte Bibliothek für schnelle Fouriertransformationen.

Da der Quellcode einer CUDA-Anwendung aus gemischten CPU- und GPU-Code besteht, sind bei der Kompilierung im Vergleich zu normalen Anwendungen zusätzliche Schritte notwendig. Im ersten Schritt wird mittels des „EDG preprocessor“ (Edison Design Group) der CPU- vom GPU-Code getrennt und in verschiedene Dateien geschrieben. Die Dateien mit dem CPU-Code können nun mit einem beliebigen C/C++-Compiler weiterverarbeitet werden. Der GPU-Code wird dagegen an eine angepasste Version des Open64-Compilers übergeben. Open64 ist ein Open-Source-C/C++-Compiler, der ursprünglich von Intel für ihre Itanium-Architektur entwickelt wurde. Die modizifierte Version erzeugt nun sogenannten PTX-Code („Parallel Thread eXecution“). PTX ist eine Zwischensprache, die dann in den meisten Fällen erst bei

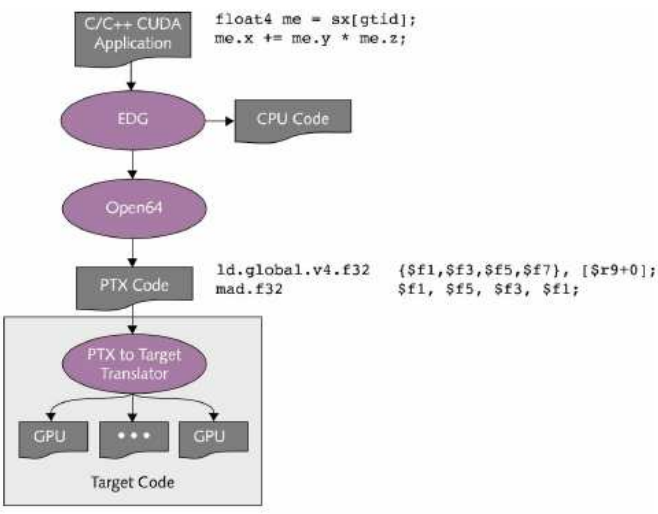


Abbildung 3. Ablauf der Übersetzung von CUDA-Quellcode

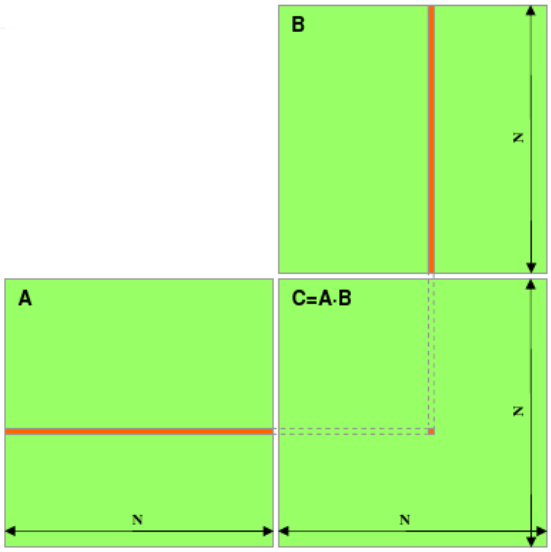


Abbildung 4. Gewöhnliche Matrixmultiplikation

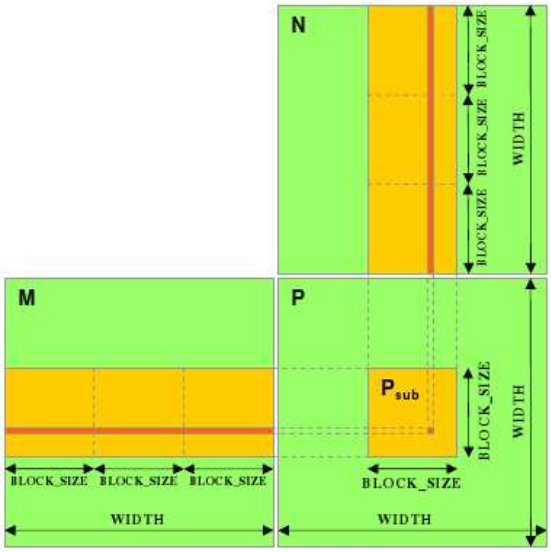


Abbildung 5. Optimierte Matrixmultiplikation

der Installation der Anwendung auf dem Zielrechner in den angepassten Code für die jeweilige GPU übersetzt wird. Durch diese Eigenschaft von CUDA ist es möglich auch „alten“ Code auf zukünftigen GPUs effizient und ohne weitere Modifikationen zu nutzen. Eine grafische Übersicht des Übersetzungsvorgangs gibt es in Abbildung 3.

*6) Peakstream:* Peakstream basiert auf einer C/C++-Bibliothek. Peakstream virtualisiert sämtliche Prozessoren in einem System, um den Entwickler vollkommen von deren speziellen Schnittstellen unabhängig zu machen.

*7) Rapidmind:* Rapidmind ist eine Entwicklungsplattform für Multicore- und Streamprozessoren. Es benötigt keinen speziellen Compiler oder ähnliches, da es sich nahtlos in C++-Code integrieren lässt.

### D. Gemeinsame Konzepte

*1) Aufteilung in Datenblöcke:* Da nicht immer alle Daten in einem Strom komplett voneinander unabhängig sind, lohnt es sich meist, die Daten in zusammenhängende Blöcke zu unterteilen, um unnötigen Datentransfer zu vermeiden. Betrachtet man zum Beispiel das intuitive Vorgehen bei einer Matrixmultiplikation (siehe Abbildung 4), so würde jedes Element der Matrizen A und B jeweils N-mal benötigt und eventuell jedes Mal neu aus dem Speicher geholt. Diese Variante wäre deutlich langsamer als die in Abbildung 5 dargestellte. Dort wird immer ein Teilbereich betrachtet, der von denselben Spalten beziehungsweise Zeilen der ursprünglichen Matrizen abhängt, die dadurch im gemeinsamen Speicher gehalten werden können und nur noch in $N/M$ Fällen aus dem globalen Speicher transferiert werden müssen.

*2) Map Operation:* Eine Map-Operation ist definiert durch einen Datenstrom $A$ und eine Funktion $f(x)$, die auf alle Elemente $a_i$ von $A$ angewendet wird.

$$map(f, A) = f(A) = B$$

Diese Operation könnte zum Beispiel in der „Transform and light“-Stufe der Grafikpipeline verwendet werden, um die Position jedes Vektors einer Szene aus Sicht der Kamera zu berechnen.

*3) Reduktion:* Bei einer Reduktion werden alle $n$ Elemente $a_i$ eines Datenstreams $A$ mittels eines beliebigen Operators $\oplus$ miteinander verrechnet.

$$reduktion(\oplus, A) = a_1 \oplus a_2 \oplus \ldots \oplus a_n = b$$

Gängige Operatoren sind zum Beispiel $+, *, min$ und $max$.

*4) Scan:* Die Scan-Funktion verrechnet alle Zahlen $a_i$ einer $n$-elementigen Menge $A$ mit einem Operator $\oplus$ und gibt alle Zwischenwerte als neue Menge zurück.

$$scan(\oplus, A) = [a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus \ldots \oplus a_n)]$$

Diese Funktion wird für viele parallele Algorithmen verwendet, zum Beispiel für parallele Sortierverfahren oder Histogrammberechnungen, kommt aber in einer sequentiellen Anwendung fast nie zum Einsatz.

## IV. Anwendungsbeispiele

Die Verwendung von GPGPU-Lösungen können in vielen Bereichen von Nutzen sein. Dort wo bestimmte mathematische Funktionen auf viele unabhängige Daten angewendet werden müssen, lässt sich die hohe Parallelität hervorragend ausnutzen.

### A. Physikberechnungen in Spielen

Lange Zeit waren komplexe physikalische Berechnungen zu aufwendig für die Echtzeitanforderungen in Spielen. Die meisten beschränkten ihren Realismus bezüglich der Physik auf die Erkennung von Kollisionen zwischen bestimmten Objekten oder die Ballistik von einzelnen wenigen Geschossen. In den heutigen Spielen wird jedoch nicht nur Wert auf ein realistisches Aussehen der Objekte gelegt, sondern auch auf deren realistisches Verhalten gegenüber Interaktionen durch andere Objekte, wie z.B. berstendes Glas, wenn es von einer Kugel getroffen wird oder abbrechende Teile eines Autos, wenn man ein anderen Gegenstand rammt. Um nun weitere Last von der CPU zu nehmen, die in den meisten Fällen immer noch der limitierende Faktor der Spielqualität ist, wurde damit begonnen, physikalische Berechnungen ebenfalls auf die GPU zu verlagern. Betrachtet man die heutige Anzahl von Objekten in einer Szene, die im Falle von bestimmten Kollisionserkennungen oder von Partikelberechnungen berücksichtigt werden müssen, scheint die GPU mit ihrem massiv-parallelen Ansatz optimal für diese Aufgabe geeignet zu sein.
Zum Beispiel läuft eine Kollisionserkennung von mehreren Objekten wie folgt ab:

1) Als Erstes wird die neue Position jedes Objekts ungeachtet möglicher Kollisionen berechnet.
2) Nun wird geprüft, ob Kollisionen aufgetreten sind.
3) In dieser Phase werden, falls es eine Kollision geben hätte müssen, die Kollisionspunkte bestimmt und
4) im letzten Schritt wird nun die korrekte kollisionsfreie Position der Objekte berechnet.

### B. Berechnung von Partikelsystemen

Ein Partikelsystem besteht, wie der Name schon sagt, aus vielen Partikeln, die meist durch ihre Position und Bewegungsrichtung definiert werden. Solche Systeme werden zur Simulation vieler realer Phänomene wie zum Beispiel Rauch, Nebel oder Explosionen genutzt.
Die aktuelle Position beziehungsweise Geschwindigkeit ist neben den Startwerten auch vor allem von externen Parametern abhängig, wie zum Beispiel der Gravitation oder von Gegenständen im Raum des Systems.
Um nun solche Bewegungen auf der GPU zu berechnen, werden die Daten jedes Partikels in Texturen gespeichert. Da man jeweils eine Textur für Ein- und Ausgabe benötigt, werden also mindestens 4 Texturen für Positionen und Geschwindigkeiten erstellt. Für zusätzliche Informationen wie Lebenszeit wäre es noch möglich, zusätzliche Texturen zu benutzen, falls diese zur Berechnung benötigt werden. Der eigentliche Algorithmus zur Berechnung eines Partikelsystems mit einer begrenzten Lebensdauer der einzelnen Partikel läuft nun wie folgt ab:

1) Neue Partikel hinzufügen und Partikel, die ihre maximale Lebensdauer überschritten haben, entfernen.
2) Für jeden Partikel die Beschleunigung aus vorheriger Geschwindigkeit und externen Parametern berechnen.
3) Aus dieser Beschleunigung wird nun die neue Position bestimmt.
4) Die resultierenden Texturen werden nun in korrekte Vektordaten für die reguläre GPU-Verarbeitung umgewandelt und gerendert.

### C. Folding@Home

Folding@Home ist ein Projekt der Universität von Stanford, das es sich zum Ziel gesetzt hat, die Veränderung des Verhaltens von Proteinen bei einer Faltung zu erforschen. Damit Proteine richtig funktionieren müssen ihre Aminosäuresequenzen durch die sogenannte Faltung eine bestimmte Raumstruktur erreichen. Weist ein Protein nicht die für ihn korrekte Raumstruktur auf, spricht man von Missfaltung. Wenn ein solcher Fall auftritt, kommt es zur Fehlfunktion des Proteins, die ernsthafte Krankheiten auslösen kann. So führt man inzwischen die Ursachen für Alzheimer, BSE, CJD (Creutzfeldt-Jakob-Krankheit), ALS (Amyotrophe Lateralsklerose) oder Parkinson auf die Missfaltung von Proteinen zurück.
Um nun diese Verhaltensänderungen besser zu verstehen, werden viele verschiedene Simulationen der Faltungen benötigt. Gängige Computer können aber nur etwa die ersten Nanosekunden eines Faltungsvorgangs in absehbarer Zeit berechnen, während der gesamte Ablauf mehrere Mikro- bis Millisekunden dauern kann. Aus diesem Grund wurde im Zuge des Folding@Home-Projekts ein Distributed-Computing-Netzwerk entwickelt, dass es jedem Computerbesitzer ermöglicht einen Teil dieser Simulation auf seiner Hardware zu berechnen.
Neben den Clients für die gängigen Betriebssysteme zur Berechnung auf der CPU gibt es seit einiger Zeit auch eine angepasste Version die den GPGPU-Ansatz verfolgt. Laut der Folding@Home-Website lässt sich dadurch die Rechengeschwindigkeit für die Simulation auf das 20- bis 30-fache steigern. Die Website bietet auch eine aktuelle Statistik über die momentane Rechenleistung der verschiedenen Clients (siehe Tabelle V).

Tabelle V
Aktuelle Rechenleistung der Clients (Stand 20. Januar 2008)

| Client | Mom. TFLOPS | aktive CPUs | Gesamt CPUs |
|---|---|---|---|
| Windows | 175 | 183746 | 1904487 |
| Mac OS X/PowerPC | 7 | 9299 | 110664 |
| Mac OS X/Intel | 20 | 6600 | 35080 |
| Linux | 41 | 23901 | 267383 |
| GPU | 38 | 641 | 5027 |
| PLAYSTATION®3 | 784 | 31631 | 411364 |

### D. GPUTeraSort

GPUTeraSort ist ein neuer Sortieralgorithmus, der für sehr große Datenbanken optimiert wurde und dafür unter anderem auf die GPU als Coprozessor zurückgreift. GPUTeraSort verschiebt dazu einen Großteil der rechen- und speicherintensiven

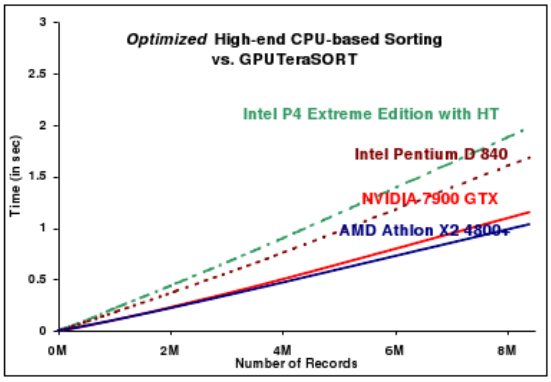


Abbildung 6. Vergleich CPU-basierter Sortieralgorithmen mit GPUTeraSort

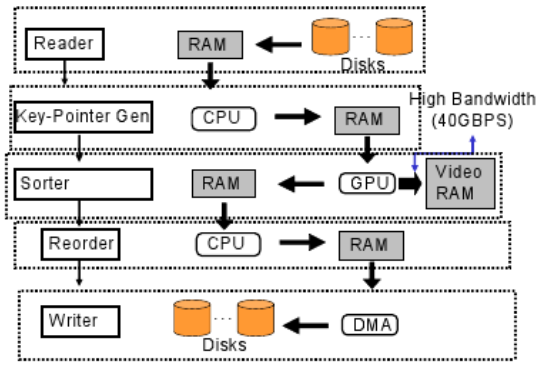


Abbildung 7. Ablaufdiagramm des GPUTeraSort-Algorithmus

Aufgaben auf die GPU, um deren Parallelismus und die schnelle Speicheranbindung auszunutzen.
Der Sortiervorgang von GPUTeraSort ist in die folgenden Stufen gegliedert (Fig 7 zeigt dasselbe in einem Ablaufdiagramm):

- **Reader**
  Der Reader liest die Eingabedaten von einem Speichermedium wie der Festplatte in einen Buffer im Hauptspeicher.
- **Key-Generator**
  Der Key-Generator erstellt die Schlüssel- und Zeigerpaare aus den Eingabedaten.
- **Sorter**
  Diese Stufe ist der rechenintensivste Teil. Die Schlüssel- und Zeigerpaare werden zur GPU übertragen, die dann den eigentlichen Sortiervorgang durchführt.
- **Reorder**
  Aus dem Ergebnis der vorherigen Stufe werden nun die eigentlichen Eingabedaten im Hauptspeicher sortiert.
- **Writer**
  Der sortierte Buffer wird nun wieder auf das Speichermedium geschrieben.

Die Entwickler haben ihre Implementation auf einem 3.0 GHz Pentium 4 mit einer NVIDIA 7800 GT GPU getestet und haben eine Performance erreicht, die vergleichbar ist mit einem 3.6 GHz Dual Xeon Server (weitere Vergleiche sind dem Diagramm in Abbildung 6 zu entnehmen). Konkret erreichten sie eine Speicherbandbreite von 50 Gigabyte pro Sekunde und führten 14 Gigaoperationen in der Sekunde auf der GPU aus.

### *E. Raytracing*

Raytracing nennt man ein Verfahren zur Berechnung von 2-dimensionalen Bildern aus 3-dimensionalen Objektbeschreibungen. Dazu werden für jedes Pixel des resultierenden Bildes Strahlen „ausgesandt“ und verfolgt, ob sie ein Objekt der Szene schneiden, und dann der resultierende Farbwert an dieser Stelle bestimmt, was wiederum ein Aussenden eines Strahls bewirken kann, falls das Objekt zum Beispiel eine spiegelnde Oberfläche besitzt (siehe auch Abbildung 9). Raytracing gilt als ein sehr aufwendiges Verfahren dieser Kategorie, allerdings

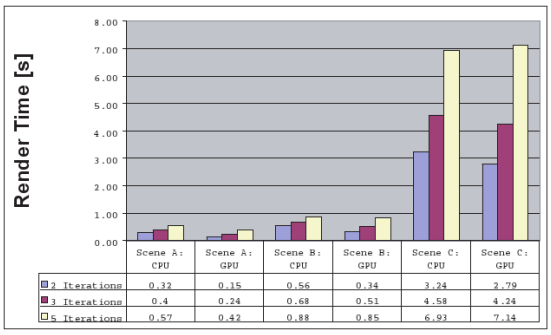


Abbildung 8. Vergleich der Laufzeit auf einem 2 GHz Pentium 4 gegenüber einer NVIDIA GeForce 6800 Ultra

auch als eines mit den realistischten Resultaten.
Da eine korrekte realistische Berechnung einer Szene bis heute in absehbarer Zeit nicht möglich ist, werden immer neue, zum Teil mit Qualitätsverlusten behaftete, Optimierungen entwickelt um den Vorgang zu beschleunigen. Ein Beispiel einer verlustbehafteten Optimierung wäre die Begrenzung der Rekursion. Durch die Begrenzung der Rekursion wird ein Strahl nicht weiter verfolgt, da eine maximale Anzahl an Rekursionen erreicht wurde und die resultierende Wirkung einer weiteren Strahlverfolgung nur minimal wäre.
In [Christen DA07] wurde nun ein solcher Raytracing-Algorithmus für eine GPU programmiert und die Ergebnisse evaluiert. Dabei wurde die benötigte Rechenzeit des Algorithmus auf der CPU mit der auf einer GPU unter Verwendung verschiedener Szenen verglichen und es stellte sich heraus (vergleiche Abbildung 8), dass in diesem Fall durch die zu dieser Zeit bestehenden Beschränkungen der GPU-Programmierung kein nennenswerter Vorteil gegenüber eines rein CPU-basierten Algorithmus besteht.
Einen aktuellerer Ansatz wird in [RSB Hybrid] verfolgt. Neben weiter optimierten CPU- beziehungsweise GPU-Verfahren wird beim „Hybrid Raytracing“-Verfahren die Arbeit zwischen CPU und GPU aufgeteilt. Ein Vergleich der Verfahren mit unterschiedlichen Testszenen auf einem AMD Athlon64 3500+ mit einer NVIDIA GeForce 7800GT Grafikkarte wird in Tabelle VI dargestellt. Die Entwickler erreichten in den meisten Fällen eine 2-fache Beschleunigung der Rechenzeit.
Verwendet man nur das sogenannte „Raycasting“-Verfahren,

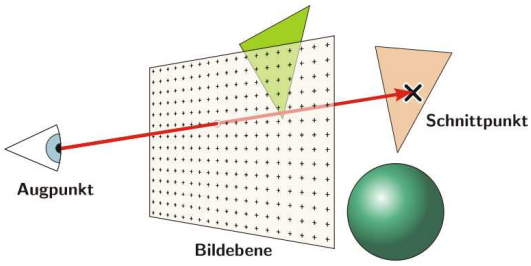


Abbildung 9. Raytracing-Prinzip (Quelle: Wikipedia)

wird der Geschwindigkeitsvorteil einer GPU-basierten Berechnung noch deutlicher (beim „Raycasting“-Verfahren findet keine Rekursion statt). Hier lässt sich eine Beschleunigen um mehr als Faktor 10 beobachten (siehe Tabelle VII).

Tabelle VI
ANZAHL BILDER PRO SEKUNDE BEI TRADITIONELLEN UND HYBRIDEN RAYTRACING-ALGORITHMEN

| Szene | A | B | C |
|---|---|---|---|
| Traditionell | 0.9 | 0.6 | 0.1 |
| Hybrid | 1.6 | 1.0 | 0.1 |

Tabelle VII
ANZAHL BILDER PRO SEKUNDE BEI CPU- UND GPU-BASIERTEN RAYCASTING-ALGORITHMEN

| Szene | A | B | C |
|---|---|---|---|
| CPU | 8.0 | 8.5 | 3.0 |
| GPU | 112 | 36 | 18 |

## V. ZUSAMMENFASSUNG UND AUSBLICK

In den vergangenen Jahren hat sich das GPGPU-Konzept durch die Entwicklung flexiblerer, programmierbarer Hardware im hart umkämpften Consumer-Markt und darauf aufbauenden angepassten Softwarebibliotheken zu einer ausgereiften Alternative gegenüber den etablierten Produkten im Bereich der mathematischen Problemlösung durch massiv-parallele Rechenwerke entwickelt. Einige Projekte setzen inzwischen auf die GPGPU-Technik und Nvidia bietet auch komplette Server an, die mit mehreren ihrer Grafikchips ausgestattet sind. Ob GPGPU allerdings in der breiten Masse bei wissenschaftlichen und industriellen Anwendungen mit den etablierten Produkten konkurrieren kann, muss sich in den nächsten Jahren zeigen.

## LITERATUR

[Luebke SC06] „General-purpose computation on graphics hardware“, David Luebke, NVIDIA, Supercomputing 2006.
[Latta GDC04] „Building a Million Particle System“, Lutz Latta, Massive Development GmbH, Game developer conference 2004.
[Owens SC07] „Data-parallel Algorithms and Data Structures“, John Owens, UC Davis, Supercomputing 2007.
[Houston SG07] „High Level Languages for GPUs - Overview“, Mike Houston, Stanford University, SIGGRAPH 2007.
[Harris SG07] „Introduction to CUDA“, Mark Harris, NVIDIA, SIGGRAPH 2007.
[Green SG07] „GPU Physics“, Simon Green, NVIDIA, SIGGRAPH 2007.
[GPGPU.org] „GPGPU.org“, http://www.gpgpu.org
[Folding@Home] „Folding@Home“, Stanford University, http://folding.stanford.edu
[GPUTeraSort] „GPUTeraSort: high performance graphics co-processor sorting for large database management“, Naga Govindaraju and Jim Gray and Ritesh Kumar and Dinesh Manocha, Chicago, IL, USA
[Christen DA07] „Ray Tracing on GPU“, Diploma Thesis, Martin Christen, University of Applied Sciences Basel (FHBB)
[RSB Hybrid] „Hybrid Ray Tracing - Ray Tracing Using GPU-Accelerated Image-Space Methods“, Philippe C.D. Robert, Severin Schoepke and Hanspeter Bieri, 2007, Institute of Computer Science and Applied Mathematics, University of Bern
[Halfhill CUDA] „Parallel processing with CUDA“, Tom R. Halfhill, www.MPRonline.com