GPU-Programmierung: OpenCL

Markus Hauschild

Seminar: Multicore Programmierung
Sommerstemester 2009

04.06.2009

Inhaltsverzeichnis

1 GPU-Programmierung

Unter GPU-Computing versteht man die Nutzung der Grafikkarte zur Berechnung allgemeiner Probleme.
Für GPU-Computing spricht unter anderem die günstige Hardware, da es sich durch die weite Verbreitung von PC-Spielen um ein in großen Mengen hergestelltes Produkt handelt. Außerdem kann man bei geeiegneten Algorithmen beziehungsweise Problemen einen Speedup vom Faktor 10 bis 100 gegenüber CPUs erreichen. [OHL+08]
So bieten moderne Grafikkarten bis zu 800 Recheneinheiten, 1300 GFlops an Rechenleistung und eine Anbindung des Speichers mit bis zu 120 GByte/s.

1.1 Entwicklung von Grafikkarten

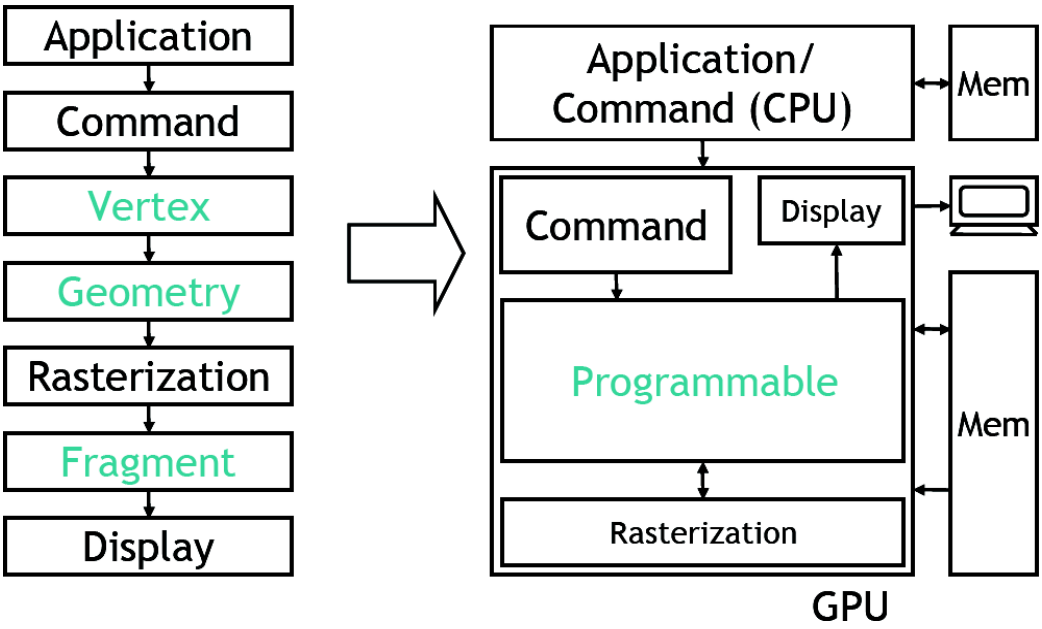

Abbildung 1: GPU-Pipeline (nach [Owe07])

Das Diagramm zeigt links eine vereinfachte Grafikpipeline und rechts die Umsetzung in Hardware.
In einer Grafikanwendung werden Geometriedaten, typischerweise Dreiecke, an die Grafikkarte übertragen und durchlaufen dabei einige Schritte bei denen sie beleuchtet und auf die Bildschirmebene abgebildet werden.
`Application` stellt hier die Grafikanwendung dar.
`Command` steht für die Verarbeitung der Befehle der Grafikanwendung.

`Vertex`: In dieser Phase werden die einzelnen Eckpunkte in die Bildschirmebene projiziert und durch Interaktion mit den Lichtquellen beleuchtet. Hierbei können mehrere Eckpunkte parallel verarbeitet werden.
`Geometry`: Hier können unter Umständen neue Geometrieobjekte aus vorhandenen erzeugt werden, die dann aber wieder die `Vertex`-Phase durchlaufen.
`Rasterization`: Als Rasterisierung wird der Prozess bezeichnet, bei dem berechnet wird, welche Pixel auf der Bildschirmebene von welchen Dreiecken überdeckt werden. Für jeden Pixel wird für jedes Dreieck von dem er überdeckt wird ein sogenanntes Fragment erzeugt.
`Fragment`: Hier wird jedem Fragment ein Farbwert zugewiesen, dafür können auch Daten aus Texturen (1- oder 2-dimensionalen Bildern, die auf Oberflächen abgebildet werden) verwendet werden.
`Display`: Zulezt wird das Finale Bild aus den Fragmenten zusammengesetzt. Normalerweise wird für jeden Pixel das Fragment mit dem geringsten Abstand zur Kamera genommen.
Ursprünglich waren die eingefärbten Teile der Pipeline nicht programmierbar sondern nur konfigurierbar. So konnte der Programmierer zwar jedem Vertex und der Lichtquelle eine Farbe und Position zuweisen, jenoch nicht die Art der Interaktion bestimmen.
Genau diese Programmierbarkeit ist aber für komplexere und realistischere Grafikeffekte notwendig und wird durch sogenannte Shader erreicht, die Operationen pro Vertex oder Fragment ausführen können. Neuere Geometry-Shader erlauben die Erzeugung von neuen Objekten aus vorhandenen Geometriedaten.
In diesem Beispiel sind die Shader als sogenannte Unified Shader ausgeführt sind, d.h. es gibt nur eine Hardwareeinheit die alle Arten von Shadern abarbeiten kann. Ursprünglich hatten die verschiedenen Shader getrennte Hardwareeinheiten und eigene Instruktionssätze. Der Vorteil an Unified Shadern gegenüber getrennten Shadern für Vertex, Geometry und Fragment ist, dass bei ungleichmäßiger Lastverteilung auf die Shaderarten nicht mehr die jeweilige Shadereinheit den Flaschenhals der Pipeline bildet. Der Vorteil aus Sicht des GPU-Computing ist, dass man nicht mehr manuell die Aufgaben auf die anderen Shader verteilen muss, sofern dies überhaupt sinnvoll möglich ist.
Einen tieferen Einblick in die Architektur von Grafikkarten erlauben z.B. [Owe07] oder [OHL+08]

1.2 Einsatzgebiete von GPU-Computing

GPU-Computing eigent sich allgemein für Streamcomputing und Anwendungen mit hoher Datenparallelität.

Darunter fallen zum Beispiel Anwendungen aus der Signalverarbeitung sowie die Verarbeitung von Audio, Bild und Videodaten. So gibt es zum Beispiel einen Trend zu Videoencodierungsprogrammen mit CUDA-Unterstützung. Weitere Einsatzgebiete sind Raytracing, Kryptographie sowie physikalische Simulationen und Physikeffekte für Computerspiele.

1.3 Entwicklung von GPU-Computing

Im folgenden Abschnitt soll die Entwickung des GPU-Computing anhand einiger Sprachen und APIs nachvollzogen werden.

1.3.1 Shader in OpenGL/DirectX

Zu Beginn der Entwicklung des GPU-Computing musste man auf bestehende APIs zurückgreifen, die eigentlich für die Darstellung von 3D-Grafik gedacht sind: OpenGL und DirectX bzw deren Shadersprachen. Die erste wissenschaftliche Nutzung wurde von DirectX-9-kompatiblen Grafikkarten ermöglicht, die Gleitkommazahlen in Shaderprogrammen verarbeiten können. Jedoch gehen einige Nachteile mit der Nutzung von Grafik-APIs einher: Algorithmen müssen umständlich als Pixel-Shader entworfen werden. Die zu verarbeitenden Daten müssen manuell als 1D oder 2D-Texturen verpackt und an die Grafikkarte übertragen werden. Die Texturen müssen zudem eine Größe von $2^n \times 2^m$ einhalten und dürfen je nach Hardware und Treiber meist nicht größer als 4096×4096 sein.
Um alle aufgeführten Sprachen vergleichen zu können soll die folgende Berechnung in C als Beispiel dienen:

```
for (int i=0; i<N; i++)
{
  z[i] = alpha * x[i] + y[i];
}
```

Als Shader in Cg (C for graphics) sieht das Beispiel (nach [Göd07]) so aus:

```
float4 saxpy (
  float2 coords : TEXCOORD0,
  uniform samplerRECT textureY,
  uniform samplerRECT textureX,
  uniform float alpha ) : COLOR
{
  float4 y = texRECT(textureY,coords);
  float4 x = texRECT(textureX,coords);
  return alpha * x + y;
```

```
}
```

An sich mag das oben gezeigte Beispiel noch recht harmlos aussehen, jedoch ist in der Hostanwendung (also dem Programm, das auf der CPU läuft) noch ein Grafikkontext zu erzeugen und diverse andere API-Aufrufe zu tätigen, die nichts mit dem eigentlichen Algorithmus zu tun haben. Einen guten Überblick darüber kann man in diesem GPGPU-Tutorial erlangen: [Göd07]
Man kann also sagen dass diese Art GPU-Computing ein Proof of Concept darstellt, dass sich Grafikkarten für allgemeine Berechnungen eigenen und Speedup erziehen können.

1.3.2 BrookGPU

Das seit 2003 an der Stanford University entwickelte BrookGPU bietet eine Abstraktion der Shaderprogrammierung für Grafikkarten.
BrookGPU befreit den Anwender von allen Aufrufen der Grafik-API und bietet einfache Methoden um Daten in sogenannte Streams zu verpacken. Desweiteren wird mit BrookGPU das Konzept eines kernels eingeführt. In Brook ist ein kernel eine Funktion die für jedes Element des Ausgabestreams aufgerufen wird und entsprechend Elemente des Eingabestreams und skalare Eingaben erhält.
Als Nachteil wäre aufzuführen dass BrookGPU nur eine Abstraktion der Grafik-API ist und nur Shader aus den kerneln erzeugt. Somit ist keine Nutzung von Scratchpads (lokale schnelle Zwischenspeicher) oder ähnlichen Erweiterungen der Hardware möglich. Außerdem ist nur Streamcomputing mögich, d.h. es sind keine Arrays als Parameter für kernel erlaubt und man hat somit keinen wahlfreien Zugriff auf die zu verarbeitenden Elemente. Desweiteren können Streams wie Texturen nur maximal zwei Dimensionen haben.
Das Beispiel als BrookGPU kernel:

```
kernel void k(float y<>, float x<>,
              float alpha, out float z<>)
{
  z = alpha * x + y;
}
```

1.3.3 CUDA

CUDA ist eine von Nvidia speziell für GPU-Computing entwickelte API, die eine direkte Ausführung auf der Hardware erlaubt ohne den Ballast eines Grafikkontexts. Außerdem werden statt Streams nun Arrays als Parameter

für kernel genutzt womit ein wahlfreier Zugriff auf Daten ermöglicht wird. Zusammen mit der Gruppierung und Indizierung von Threads, ist damit ein wahlfreier Zugriff auf die Arrays möglich, jedoch hat sich hier der Programmierer selbst darum zu kümmern welche kernel-Instanzen welche Daten verarbeiten. Die Gruppen von Threads werden dann entsprechend auf Hardwareeinheiten abgebildet, die bei regulären Zugriffsmustern auf den Speicher der Grafikkarte eine Hohe Speicherbandbreite bieten. Dadurch dass CUDA direkten Zugriff auf die Hardware hat ist auch die Nutzung des Scratchpad möglich, einem schnellen Zwischenspeicher der jeweils einer Gruppe von Threads zur Verfügung steht.
Ein Nachteil an CUDA ist, dass es nur mit Nvidia Grafikkarten funktioniert. Weiteres Probleme sind die unzureichende Fehlerbehandlung in kerneln und das eingeschränkte debugging durch Softwareemulation der Hardware.
Das Beispiel als CUDA kernel (nach [Nvi09c]):

```
__global__ void k(float* x, float* y,
                  float alpha, float* z)
{
  int i = threadIdx.x +
          blockDim.x * blockIdx.x;
  z[i] = alpha * x[i] + y[i];
}
```

2 OpenCL

2.1 Entwicklung

OpenCL ist ein offener Standard, der von der Khronos Group betreut wird, die unter anderem auch für OpenGL verantwortlich ist. Initiiert wurde OpenCL von Apple, AMD, Intel und Nvidia.
Die OpenCL working group wurde im Juni 2008 bei Khronos eingerichtet.
In der Zeit bis zur Fertigstellung der Spezifikation im Dezember 2008 sind viele weitere Firmen der working group beigetreten, wie zum Beispiel Spieleentwickler wie Blizzard oder weitere Prozessorhersteller wie IBM, Sun und ARM.
Die schnelle Verabschiedung des Standards ist dadurch zu erklären, dass viele Konzepte aus CUDA übernommen wurden.

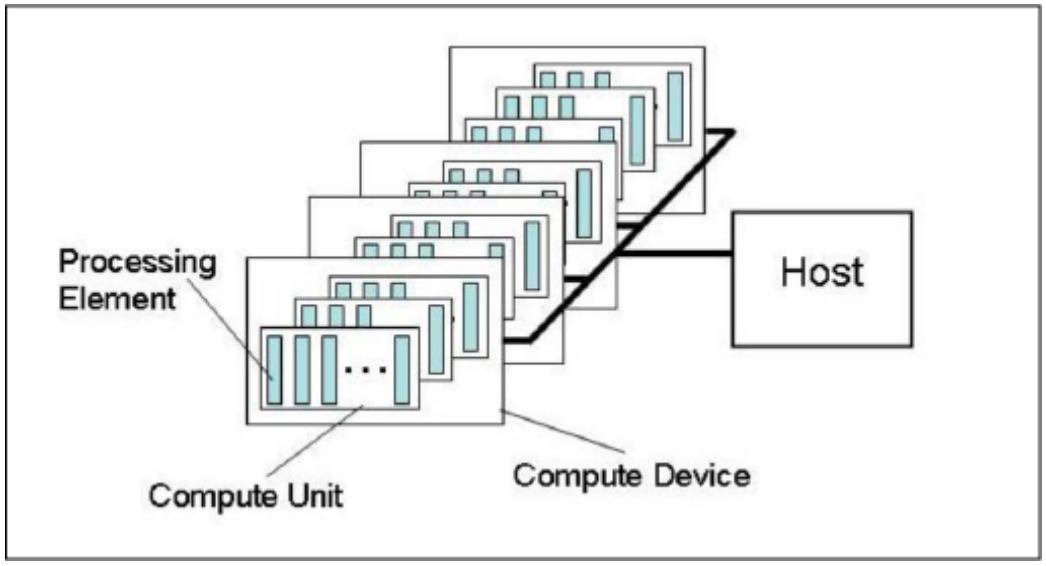

Abbildung 2: OpenCL Platform Model (nach [Khr09])

2.2 Architektur

OpenCL bietet eine Abtraktion über ein oder mehrere sogenante Compute Devices (z.B. Grafikkarten), die aus mehren Compute Units bestehen, die wiederum mehrere Processing Elements besitzen. Eine Instanz eines kernel wird dabei auf einem Processing Element ausgeführt.

2.2.1 Execution Model

Ein OpenCL Programm besteht aus einer Host-Anwendung, die sich um die Initialisierung der Daten und die Speicherverwaltung kümmert. Desweiteren enthält die Host-Anwendung alle kernel.
Ein kernel ist ein Block ausführbaren Codes, ähnlich einer C-Funktion.
Das Host-Programm führt den kernel über einem Indexram mit 1-3 Dimensionen (NDRange) aus. Eine Instanz eines Kernels wird in OpenCL-Terminologie als work-item bezeichnet. Ein work-item hat eine global eindeutige ID aus dem Indexraum. Work-items werden in work-groups gruppiert, die eine eindeutige work-group ID haben. Außerdem hat jedes work-item eine innerhalb einer work-group eindeutige ID. Über die IDs ist es dem Programmierer überlassen die Arbeit unter den Threads aufzuteilen, indem jeder Thread je nach ID einen anderen Teil der Daten bearbeitet.

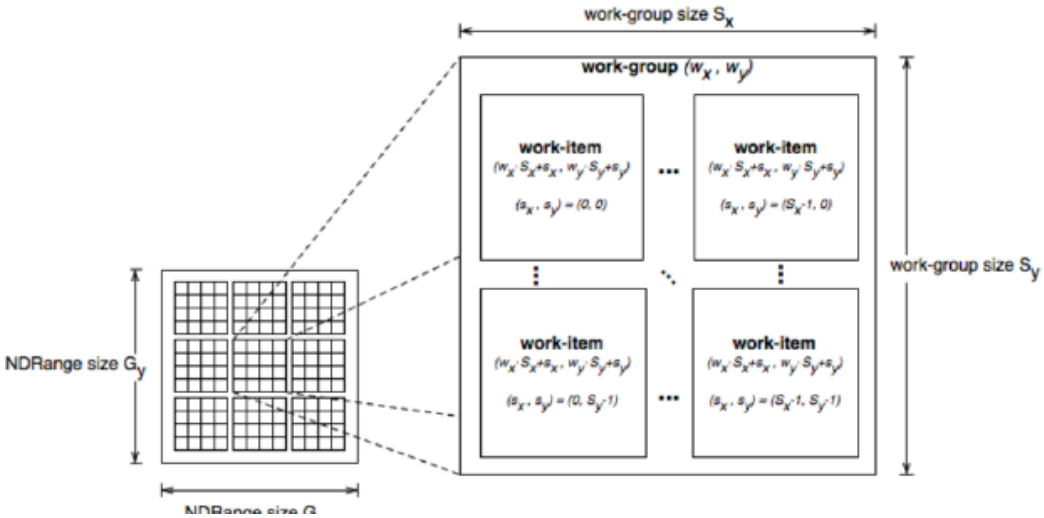

Abbildung 3: OpenCL Execution Model (nach [Khr09])

2.2.2 Command-queue

In OpenCL werden einige Befehle über die Command-queue abgearbeitet. Dies sind Befehle zur Ausführung eines kernel, Befehle für den Datentransfer zwischen Hauptspeicher und dem Speicher des Compute Device. Desweiteren können Synchronisationsbefehle eingereiht werden, die Beispielsweise einen Datentransfer auf die Ausführung eines kernels warten lassen.

2.2.3 Speichermodell

In OpenCL gibt es folgende Adressräume:
Private für Daten die zu einem work-item gehören. Local für Datren die zu einer work-group gehören. Global für Daten auf von allen work-items aus allen work-groups aus zugegriffen werden kann. Sowie constant für read-only Daten.
Die jeweilige OpenCL-Implementierung versucht diese Hierarchie so gut wie möglich auf die jeweilige Hardware abzubilden.
In OpenCL ist die Konsistenz der Daten im Speicher nicht immer garantiert. So sind innerhalb einer work-group nach einer Barriersynchronisation Daten aus `local` und `global` Bereichen konsistent, jedoch wird für `global` zugreifbare Daten keine Konsistenz über die Grenzen einer work-group hinweg garantiert.

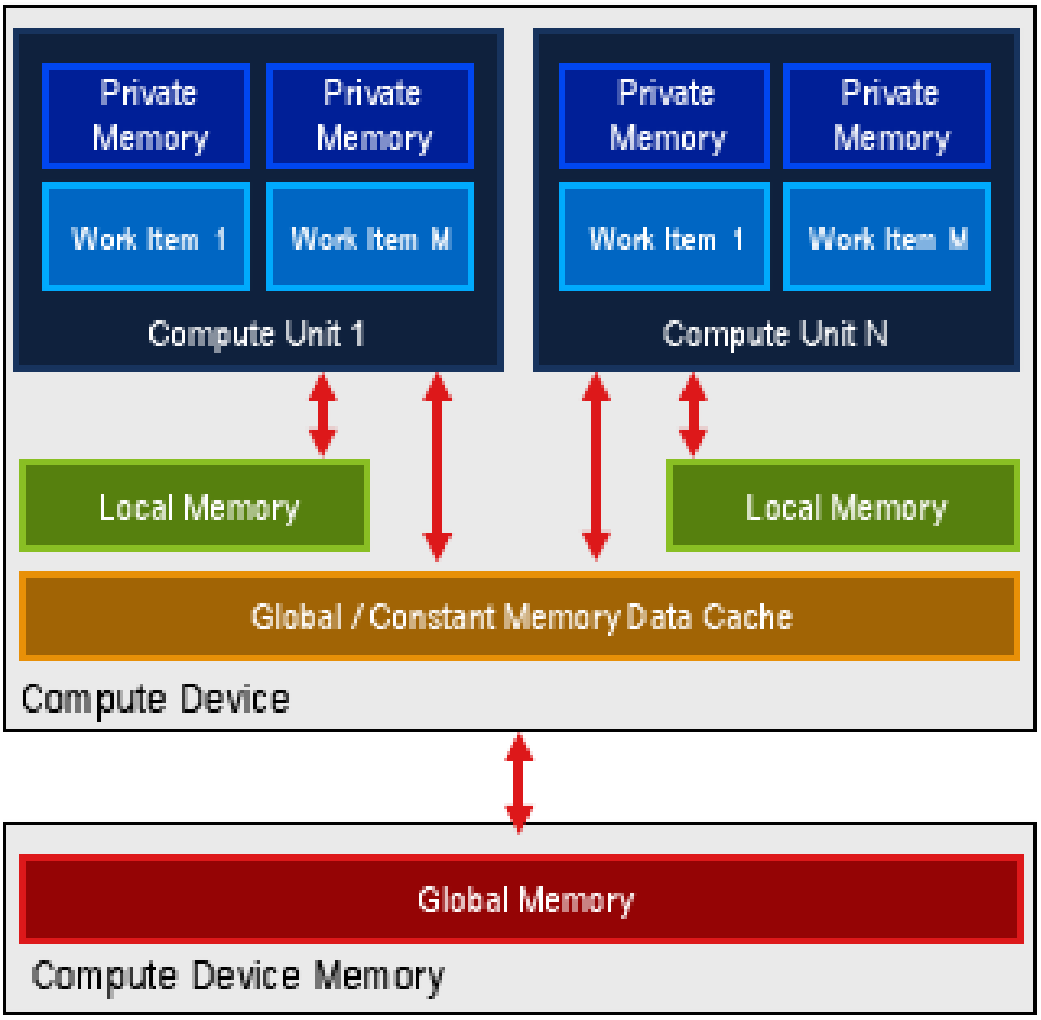

Abbildung 4: OpenCL Memory Model (nach [Khr09])

2.3 Spracheigenschaften

OpenCL erlaubt innerhalb der kernel im wesentlichen C99 mit einigen Ausnahmen. So sind Funktionspointer und rekursive kernel-Aufrufe nicht erlaubt. Ebenso werden Arrays variabler Länge und structs nicht unterstützt.
Einige Features sind in OpenCL nur optional über Erweiterungen verfügbar, sofern diese von der Hardware unterstützt werden. Darunter fallen unter anderem Gleitkommazahlen mit doppelter Genauigkeit sowie atomare Funktionen wie increment.
Weitere technische Einzelheiten finden sich in den Folien zur Vorstellung von OpenCL ([Khr09]).

2.4 Vergleich mit CUDA

Bei einem ersten Vergleich mit CUDA fällt auf, dass sich viele API-Befehle ähneln, jedoch unterschiedliche Parameter benötigen. So müssen zum Beispiel in OpenCL alle Pointer mit einem `adress space qualifier` versehen werden, der angibt wo sich das referenzierte Datum befindet. Desweiteren werden alle Speichertransfers in OpenCL über die Command-Queue abgewickelt. Ein weiterer Unterschied besteht darin, dass OpenCL die kernel erst zur Laufzeit für die entsprechende Platform kompiliert.

2.5 Beispiel

Das Beispiel als OpenCL kernel:

```
__kernel void k(__global const float* y,
                __global const float* x,
                __global const float alpha,
                __global float* z)
{
  int index = get_global_id(0);
  z[index] = alpha * x[index] + y[index];
}
```

2.6 Besonderheiten

Verglichen mit anderen Sprachen und APIs im Bereich des GPU-Computing hebt sich OpenCL besonders dadurch hervor, dass neben neueren Grafikkarten von AMD/ATI und Nvidia diverse weitere Platformen unterstützt werden. So soll es Treiber für den Intel Larrabee, den Cell und andere CPUs geben. Desweiteren enthält OpenCL ein Profil für embedded und mobile Devices, das unter anderem eine geringere Genauigkeit bei Gleitkommaoperationen fordert.

2.7 Verfügbrkeit

Da OpenCL noch relativ neu ist, sind zur Zeit noch keine Treiber verfügbar. Von AMD/ATI ist ein Treiber für das zweite Halbjahr 2009 angekündigt. [ATI09]
Von Nvidia sind Beta-Treiber für registrierte Entwickler seit 20. April 2009 verfügbar. [Nvi09a] Nvidia hat die Treiber bereits zur Zertifizierung bei der Khronos Group eingereicht. [Nvi09b]

Literatur

[ATI09] ATI. A Brief History of General Purpose (GPGPU) Computing. Website, 2009. Available online at `http://ati.amd.com/technology/streamcomputing/gpgpu_history.html`; visited on June 2nd 2009.

[Göd07] Dominik Göddeke. GPGPU::Basic Math Tutorial. Website, 2007. Available online at `http://www.mathematik.tu-dortmund.de/~goeddeke/gpgpu/tutorial.html`; visited on May 28th 2009.

[Khr09] Khronos Group. OpenCL - The Open Standard for Heterogeneous Parallel Programming, February 2009. Available online at `http://www.khronos.org/developers/library/overview/opencl_overview.pdf`.

[Nvi09a] Nvidia. NVIDIA Releases OpenCL Driver To Developers. Website, April 2009. Available online at `http://www.nvidia.com/object/io_1240224603372.html`; visited on May 28th 2009.

[Nvi09b] Nvidia. NVIDIA SUBMITS OPENCL 1.0 DRIVER TO KHRONOS FOR CONFORMANCE CERTIFICATION FOR WINDOWS AND LINUX. Website, April 2009. Available online at `http://www.nvidia.com/object/io_1242238985095`; visited on May 28th 2009.

[Nvi09c] Nvidia. OpenCL JumpStart Guide, 2009. Available online at `http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf`.

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.

[Owe07] John Owens. GPU Architecture Overview. SIGGRAPH, 2007. Available online at `http://gpgpu.org/static/s2007/slides/02-gpu-architecture-overview-s07.pdf`.