

Seminar Parallele Algorithmen



Paralleles Rechnen auf Grafik-Hardware

Dominik Nuszpl & Christoph Busold

Inhalt

- **Einleitung**
- Aufbau der Hardware
- Threadverwaltung
- Speicherverwaltung
- Programmierschnittstelle (API)
- Beispiel Matrixmultiplikation

Einleitung

Was ist CUDA?

- CUDA steht für Compute Unified Device Architecture.
- Zusatzbibliotheken:
 - CUFFT (CUDA Fast Fourier Transform)
 - CUBLAS (CUDA Basic Linear Algebra Subprograms)
 - CUDPP (CUDA Data Parallel Primitives)
- Integration von Direct3D oder OpenGL möglich

Einleitung

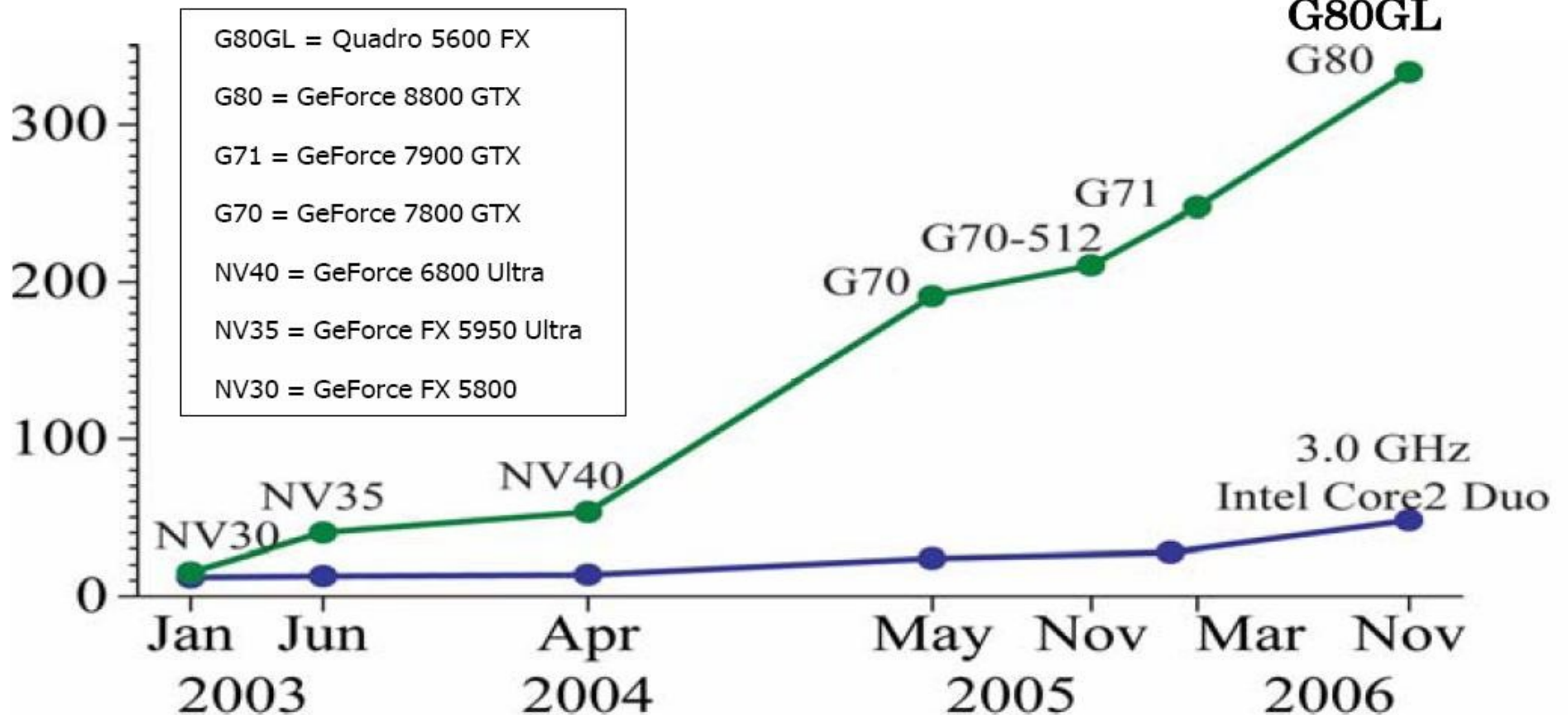
Was ist CUDA?

- Es stehen zwei APIs zur Verfügung:
 - die CUDA Driver API
 - Präfix: cu
 - ist eine low-level API und benötigt viel Code und Lernaufwand
 - die CUDA Runtime API
 - Präfix: cuda
 - ist eine high-level API und einfacher zu Programmieren
 - setzt auf der Driver API auf
 - bietet einen Emulationsmodus

Einleitung

Wozu CUDA?

GFLOPS



Einleitung

Einsatzgebiet

- Berechnungen mit
 - vielen arithmetischen Operationen und
 - wenigen Speicherzugriffen
- hochparallele Algorithmen
 - die selbe Funktion wird auf vielen Daten ausgeführt (SIMD)
- Typische Einsatzgebiete
 - Bild- und Videobearbeitung
 - Mustererkennung, Signalverarbeitung
 - naturwissenschaftliche Simulationen

Einleitung

Vorteile

- Früher:
 - GPU-Programmierung nur über Grafik-API
 - schwer zu erlernen
 - Kein beliebiger Schreibzugriff auf Speicher
- mit CUDA:
 - allgemeine API mit umfangreicher Mathe-Bibliothek
 - leicht erlernbar, da Erweiterung von C
 - voller Zugriff auf Grafikkartenspeicher, sowohl lesend als auch schreibend

Einleitung

Nachteile

- Aktuelle Grafikprozessoren unterstützen nur einfache Genauigkeit
 - für viele Simulationen ungeeignet
- Bandbreite zwischen Host (CPU) und Device (GPU) stellt einen Flaschenhals dar
- Festlegung auf einen Hersteller (NVIDIA)
- IEEE 754 wird nicht voll umgesetzt
 - z.B. keine NaN's

Inhalt

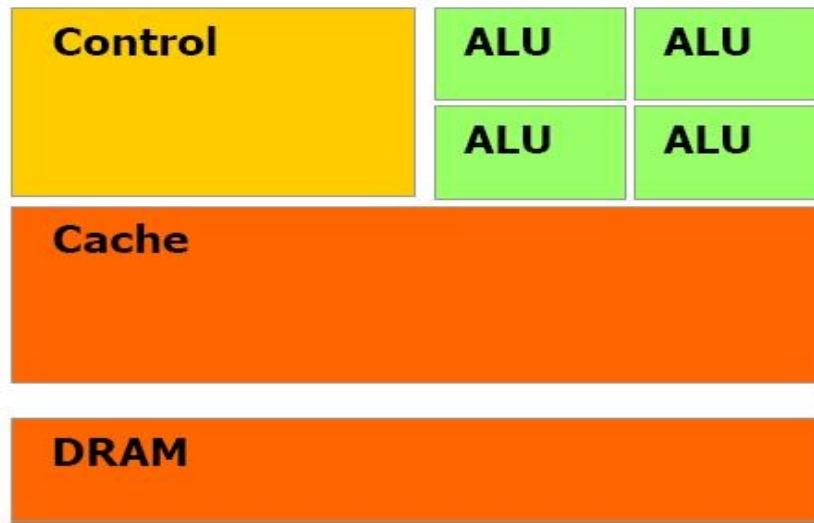
- Einleitung
- **Aufbau der Hardware**
- Threadverwaltung
- Speicherverwaltung
- Programmierschnittstelle (API)
- Beispiel Matrixmultiplikation

Aufbau der Hardware

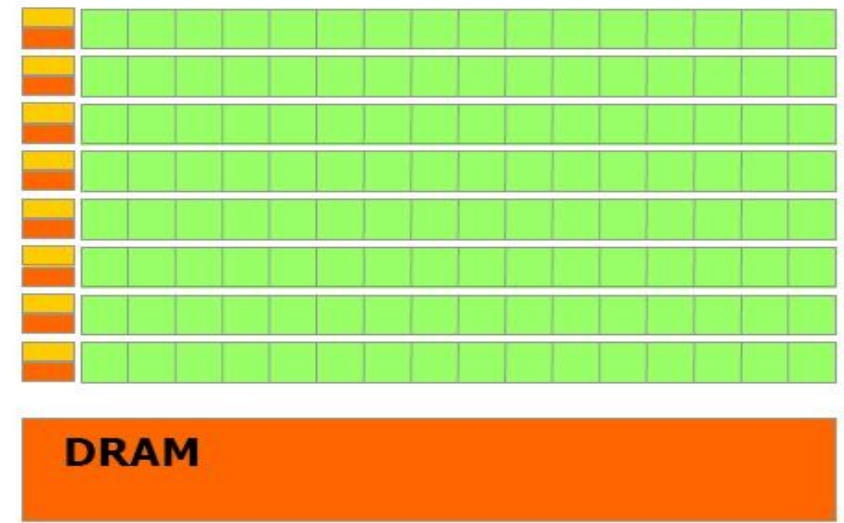
- Grafikchip besteht aus mehreren SIMD-Multiprozessoren
- Multiprozessor besteht aus
 - 8 Prozessoren
 - 16 KB Shared Memory
 - 8 KB Cache für konstanten Speicher
 - 8 KB Cache für Texturspeicher
 - 8192 32bit Registern
- Ein Multiprozessor führt auf allen Prozessoren die selbe Operation auf unterschiedlichen Daten aus (SIMD).

Aufbau der Hardware

CPU vs. GPU



CPU



GPU

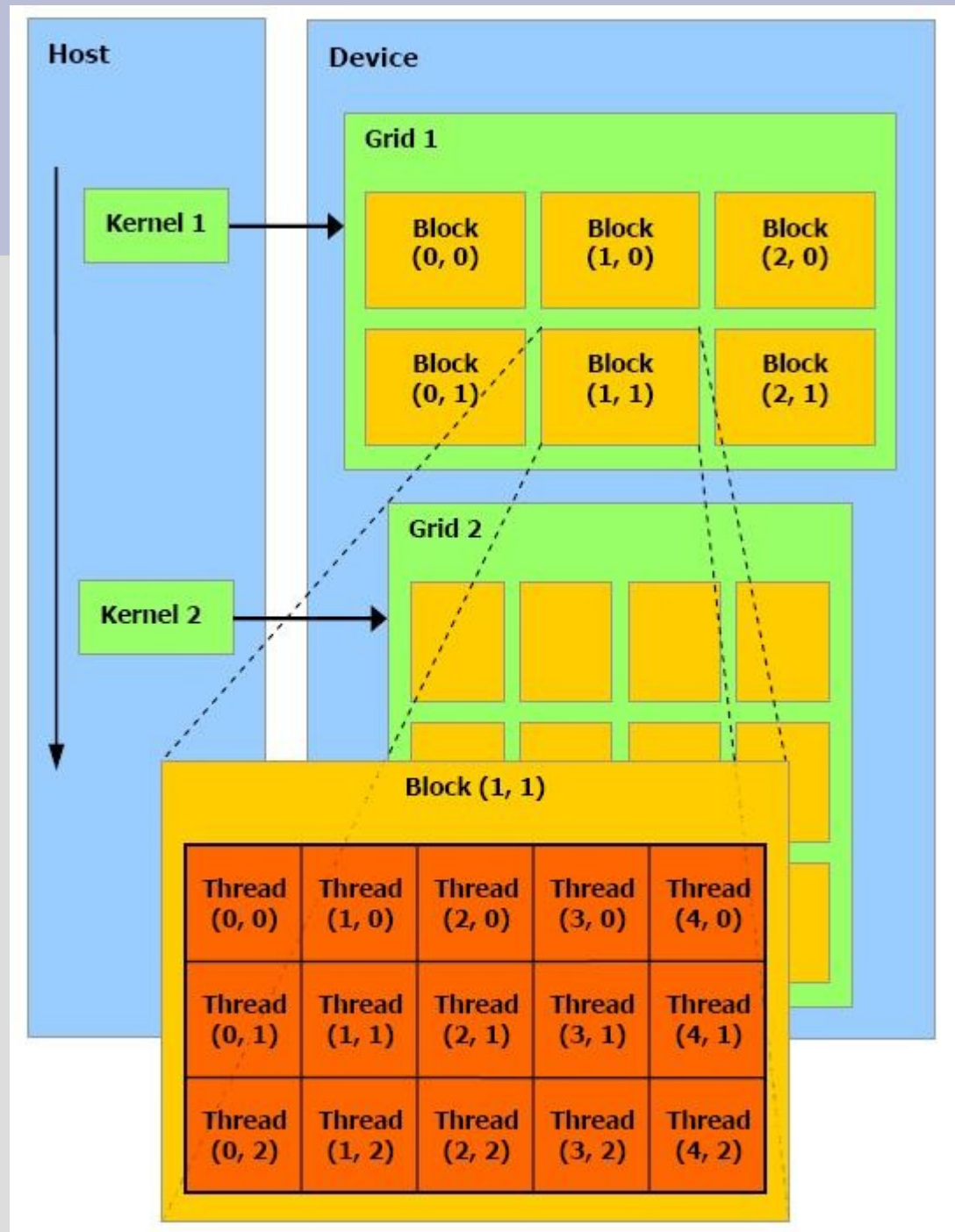
Durch die Fokussierung auf hohe Rechenleistung entfällt die Notwendigkeit für aufwändige Ablaufsteuerung und große Caches.

Inhalt

- Einleitung
- Aufbau der Hardware
- **Threadverwaltung**
- Speicherverwaltung
- Programmierschnittstelle (API)
- Beispiel Matrixmultiplikation

Thread- verwaltung

- Mehrere Threads bilden einen Block
- Mehrere Blöcke bilden ein Gitter



Threadverwaltung

Blöcke

- Threads, die auf gemeinsamen Daten arbeiten müssen, können in einem Block organisiert werden.
- Threads in dem selben Block
 - haben einen gemeinsamen Speicher (Shared Memory) und
 - können sich synchronisieren: `__syncthreads`
- Ein Block kann nur von einem Multiprozessor ausgeführt werden.

Threadverwaltung

Gitter

- Gitter bündeln Blöcke vom selben Kernel
- Threads aus unterschiedlichen Blöcken können weder
 - untereinander kommunizieren noch
 - sich synchronisieren.

Threadverwaltung

Funktionskennzeichner

- Eine CUDA-Funktion kann auf folgende Arten gekennzeichnet werden:
 - `__host__`
 - wird vom Host ausgeführt
 - `__global__`
 - wird vom Device ausgeführt
 - kann nur vom Host aufgerufen werden
 - `__device__`
 - wird vom Device ausgeführt
 - kann nur vom Device aufgerufen werden
- Beispiel:
 - `__global__ void Func(int p1, int p2);`

Threadverwaltung

Einschränkungen

- Device- und Global-Funktionen dürfen weder
 - Rekursion benutzen noch
 - statische Variablen deklarieren.
- Global-Funktionen
 - können nichts zurückgeben,
 - werden asynchron aufgerufen und
 - können keine Parameterliste größer als 256 Byte besitzen (Parameter werden über Shared Memory übergeben).

Threadverwaltung

Kernelaufruf

- Festlegen der Block- und Gittergröße

```
dim3 dimBlock(5, 3);  
dim3 dimGrid(3, 2);
```

- Kernelaufruf auf dem Device

```
Func<<<dimGrid, dimBlock>>>(p1, p2);
```

- Die übrige Threadverwaltung wird von der Grafikkarte selbst übernommen.

Threadverwaltung

Identifizierung

- Jeder Thread hat innerhalb seines Blocks eine eindeutige ID. (Thread-ID)
 - maximal 3-dimensional
 - maximal 512 Threads pro Block
- Jeder Block hat innerhalb des Gitters eine eindeutige ID. (Block-ID)
 - maximal 2-dimensional
 - maximal 65535 Blöcke pro Gitter in jeder Dimension

Threadverwaltung

Vordefinierte Variablen

- gridDim – Dimension des Gitters
- blockDim – Dimension des Blocks
- blockIdx – Index des Blocks
- threadIdx – Index des Threads

- Beispiel:
 - 2-dimensionaler Block wird auf 1-dimensionales Array abgebildet
 - `int index = blockDim.x * threadIdx.y + threadIdx.x`

Threadverwaltung

Warps

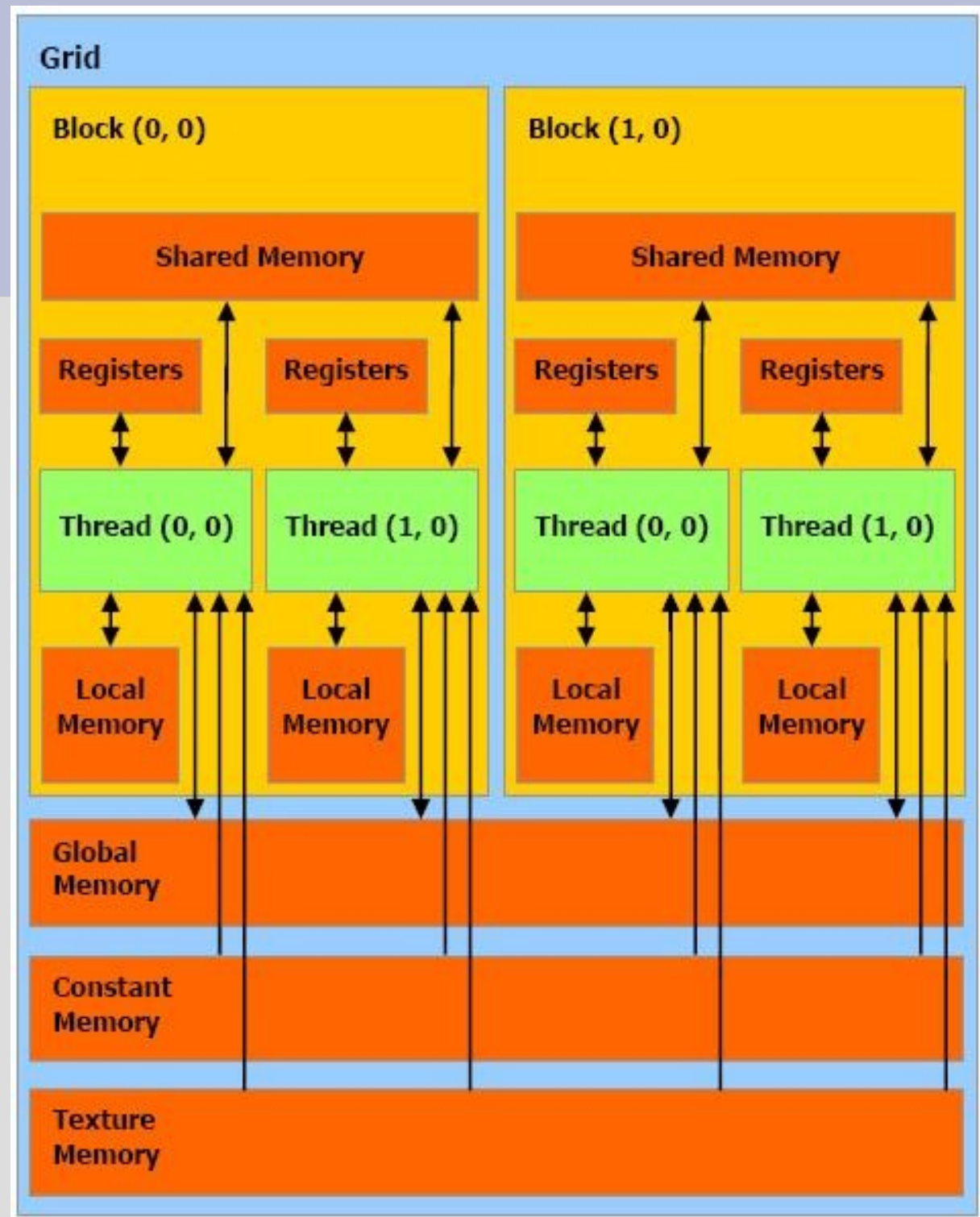
- Die Threads innerhalb eines Blocks werden vom Device zusätzlich in Warps unterteilt.
- Bei der Ausführung eines Blocks schaltet der Thread-Scheduler zwischen den Warps um.
- Der Entwickler hat keinen Einfluss auf die Reihenfolge der Ausführung.
- Die Anzahl der Threads pro Warp ist 32.

Inhalt

- Einleitung
- Aufbau der Hardware
- Threadverwaltung
- **Speicherverwaltung**
- Programmierschnittstelle (API)
- Beispiel Matrixmultiplikation

Speicherzugriff

- Constant und Texture Memory können von den Threads nicht verändert werden.



Speicherverwaltung

Speicher kennzeichner

- Eine Variable kann auf folgende Arten gekennzeichnet werden:
 - `__device__` wird im globalen DRAM der Grafikkarte abgelegt und kann von allen Threads benutzt werden
 - `__constant__` wird im konstanten Speicher abgelegt und kann nur gelesen werden
 - `__shared__` wird im On-Chip Speicher eines Multiprozessors abgelegt und kann nur von Threads aus dem entsprechenden Block benutzt werden

Inhalt

- Einleitung
- Aufbau der Hardware
- Threadverwaltung
- Speicherverwaltung
- **Programmierschnittstelle (API)**
- Beispiel Matrixmultiplikation

Programmierschnittstelle

Speicherverwaltung

- `cudaMalloc` legt Speicher auf Device an.
- `cudaFree` gibt den Speicher wieder frei.

- **Beispiel:**

```
int size = 5 * sizeof(float);  
float *data;  
cudaMalloc((void**) &data, size);  
...  
cudaFree(data);
```

Programmierschnittstelle

Speichertransfer

- `cudaMemcpy` kopiert Speicher vom
 - Host zum Device (`cudaMemcpyHostToDevice`),
 - Device zum Host (`cudaMemcpyDeviceToHost`).
- **Beispiel:**
 - kopiert `size` Bytes von `A` auf dem Host nach `B` auf dem Device
 - `cudaMemcpy(B, A, size, cudaMemcpyHostToDevice);`

Programmierschnittstelle *Threadverwaltung*

- Kernelaufrufe sind generell asynchron.
- `cudaThreadSynchronize` wartet, bis alle Threads des laufenden Kernels beendet sind.

Programmierschnittstelle

Deviceverwaltung

- `cudaGetDeviceCount` gibt die Anzahl der verfügbaren CUDA Geräte zurück.
- `cudaGetDeviceProperties` gibt Informationen zu einem Gerät zurück.
- `cudaSetDevice` legt fest, welches Gerät bei nachfolgenden Kernelaufrufen benutzt wird.

Programmierschnittstelle

Richtlinien

- Für beste Performance sollte man auf folgendes achten:
 - Problem in so kleine Teile zerlegen wie möglich
 - jeder Thread sollte nur eine atomare Aufgabe erledigen
 - je mehr Threads, desto besser können die Prozessoren ausgelastet werden
 - große Anzahl Threads pro Block, am besten ein Vielfaches der Warp Size
 - Verzweigungen (if, switch, goto) in den Threads vermeiden

Programmierschnittstelle

Richtlinien

- Für beste Performance sollte man auf folgendes achten:
 - Datentransfer zwischen Host und Device gering halten
 - Daten der Threads möglichst im Shared Memory halten, Zugriff auf globalen Speicher vermeiden
 - Datentransfers von allen Threads gemeinsam ausführen, um besten Datendurchsatz zu erhalten

Inhalt

- Einleitung
- Aufbau der Hardware
- Threadverwaltung
- Speicherverwaltung
- Programmierschnittstelle (API)
- **Beispiel Matrixmultiplikation**

Matrixmultiplikation

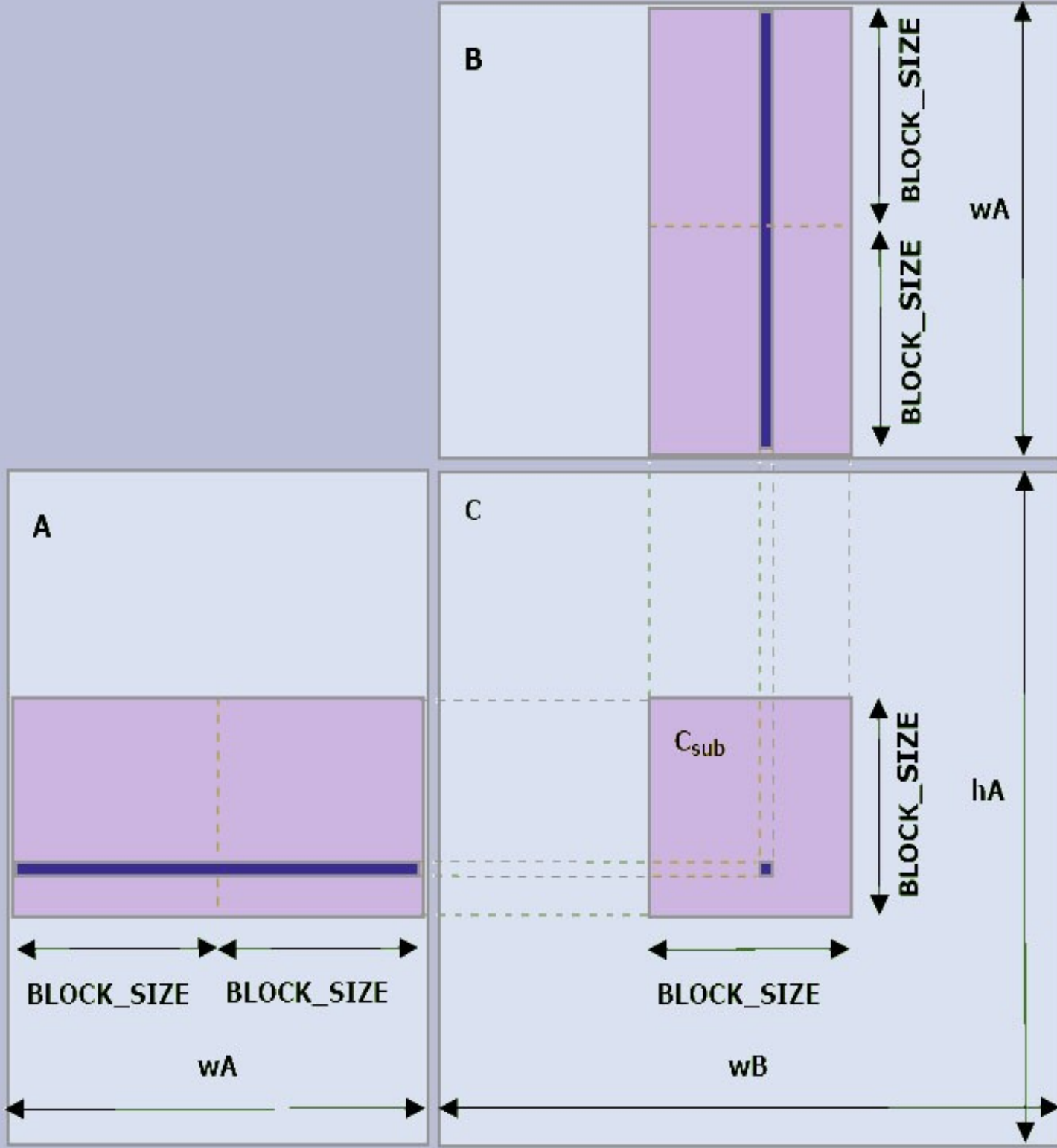
Problemstellung

- Zwei Matrizen beliebiger Größe sollen miteinander multipliziert werden.
- Für optimale Geschwindigkeit soll der Shared Memory verwendet werden.
- Es muss eine Möglichkeit gefunden werden, die benötigten Daten passend im Shared Memory abzulegen.

Matrixmultiplikation

Lösungsansatz

- Die Lösungsmatrix wird aufgeteilt in Submatrizen, die von je einem Block berechnet werden.
- Innerhalb jedes Blocks berechnet je ein Thread ein Element der Submatrix.
- Die benötigten Rechteckmatrizen der Ausgangsmatrizen werden in Blockgröße aufgeteilt, um in den Shared Memory zu passen.



Alternativen *GPGPU*

- Brook
 - Stanford University Projekt
 - Erweiterung von ANSI-C
 - Open Source (GPL und BSD-License)
- SH
 - University of Waterloo Projekt
 - Open Source (GNU Lesser GPL)
 - wird nicht weiter entwickelt

Alternativen *GPGPU*

- PeakStream
 - kommerzielle Variante von Brook
 - inzwischen aufgekauft von Google
 - erlaubt Programmierung von x86, FireStream und Cell
- Rapidmind
 - kommerzielle Variante von SH
 - erlaubt sowohl Programmierung von ATI- und NVIDIA-Karten, als auch Cell Prozessor

Alternativen *FireStream*

- entwickelt von AMD/ATI
- Hardware Interface CTM/CAL
 - unabhängig von Programmiersprache
 - direkter Zugriff auf Prozessoren und Speicher
- Programmiersprache Brook+
 - für ATI Karten optimierte Variante von Brook
- ebenfalls Probleme wie CUDA, z.B. nur einfache Genauigkeit

Quellen

- NVIDIA
 - CUDA Zone
(http://www.nvidia.com/object/cuda_home.html)
 - Developer Zone
(<http://developer.nvidia.com/page/home.html>)
 - insbesondere: Programming Guide
- University of Illinois
 - Vorlesungen "Programming Massively Parallel Processors" von Wen-mei W. Hwu
(<http://courses.ece.uiuc.edu/ece498/a11/Syllabus.html>)
- Wikipedia

ENDE

Vielen Dank für Ihre Aufmerksamkeit!

Noch Fragen?